ACID Compliant Distributed Key-Value Store

Lakshmi Narasimhan Seshan[#1], Rajesh Jalisatgi[#2], Vijaeendra Simha G A[#3]

[#1]lsimhan.cse@gmail.com

[#2]rajeshjalisatgi@gmail.com

[#3]vijaeendra@gmail.com

Abstract

Built a fault-tolerant and strongly-consistent key/value storage service using the existing RAFT implementation. Add ACID transaction capability to the service using a 2PC variant. Build on the raftexample[1] (available as part of etcd) to add atomic transactions. This supports sharding of keys and concurrent transactions on sharded KV Store. By Implementing a transactional distributed KV store, we gain working knowledge of different protocols and complexities to make it work together.

***Keywords*— ACID, Key-Value, 2PC, sharding, 2PL Transaction**

I. Introduction

Distributed KV stores have become a norm with the advent of microservices architecture and the NoSql revolution. Initially KV Store discarded ACID properties from the Database world. Nowadays with a lot of applications requiring transactional properties (ex.Bitcoin) and also to take advantage of the distributed nature of the systems, a lot of Databases are designed with ACID properties with the underlying store being a distributed KV Store. CockroachDB, TinyKV are few examples. [2, 3]. We wanted to explore what it involves to implement such a system and learn from that. We have organised this paper with the following topics : Architecture,Transaction,Performance,Evaluation,Learning and Future Work..

II. Architecture

Our ACID compliant Distributed Infrastructure Architecture is inspired by Spanner[4], CockroachDB, TinyKV. At a high level[fig1], a 2PC Coordinator (Transaction Manager) is distributed and backed by Raft Consensus protocol. The Key range is split into different shards and are replicated in as many servers (called as Replica Server) as available. Raft consensus protocol is run between the shards and it elects the leaders for each shard. In our model, each transaction is carried out by the respective Shard leaders. An Alternative model is to send the transactions to all members of the shards and once the majority of the cohorts for that shard returns, decide on the outcome of that transaction.[5]. We decided to go with the "shard leader" model as we thought that would be easier to implement. All components run http and grpc endpoints. Http endpoints are for debugging/configuration. Grpc Endpoints are used for communication between components.

Replica Manager (RM): RM is the service discovery part of the system. RM is initiated with each of the Replica Servers available in the system. Users have to instantiate the RM with the number of shards that the user wants the key to be split into. Currently we cannot dynamically change while RM is running. New Replica Servers cannot be added or removed from once RM is up and running. However Replica Servers can go down and come back and RM can detect this. Each Shard Leader updates the information to RM, while the DTM leader updates its information to the RM. Clients and DTM query for the DTM leader, Shard leader respectively.

Distributed Transaction Manager(DTM) : DTM is responsible for driving the Two Phase Commit. Clients interact with the RM to get information about the DTM leader. A DTM leader can process multiple transactions concurrently. DTM runs raft among its peers to elect the leader, log the 2PC messages. We can dynamically add or remove transaction managers while the system is running.

Store : Each replica server runs a store process. It is responsible for receiving shard information from RM, handling transactions from a DRM leader and to run Raft protocol for each of the shards that is deployed in the replica server.

III. ACID Transactions

Our implementation of ACID took inspiration from spanner and cockroach db. Two Phase Commit (2PC) provides atomicity and two phase locking (2PL) is used for Isolation. We rely on clients for consistency of the data and Durability is provided by storage of the RAFT example.

Raft leader is the transaction coordinator for the transaction and relies on the leader to make changes to the state of transaction. Raft ensures replication of transaction state to peer nodes. 2PC inherently has

performance overhead so we did some tweaks to simplify implementation without compromising on consistency. We follow the regular PREPARE phase and once outcome is decided for the transaction, write it into a raft based log and immediately return the transaction result to the client without waiting for the writes to be reflected in the store. Modifying the final state of a transaction on cohorts is done by worker thread. Multiple dedicated worker threads are configured to abort/commit transactions. We implement PrC (Presumed Commit) optimization of 2PC[6]. We write

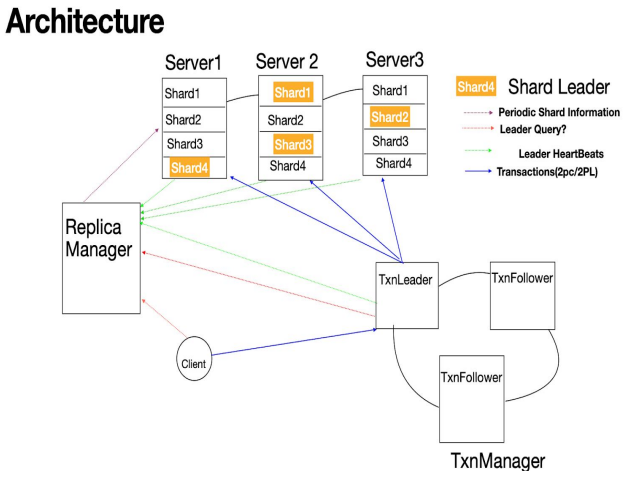

Fig 1: System architecture

two log records, one initiation record and second commit record. On COMMIT of transaction, deleting entry from the record reduces log footprint.

2PL[7] is implemented with adding write intent for each key along with read-write locks for concurrent access. Accessing read-write lock is mandatory for accessing any fields of value. A write transaction takes the write lock and stages the value on write intent in the PREPARE stage if it's empty. If write intent is occupied, the transaction goes into conflict resolution. The current transaction queries the state of the staged transaction with a gRPC call to the transaction coordinator. If the outcome of a staged transaction is a) PENDING, would abort the current transaction since a concurrent inflight transaction has taken lock on it. b) COMMIT/ABORT, current transaction would execute the outcome on kvstore and stage itself onto write intent. This process termed as resolving transaction conflict. Figure 3 shows the resolving transaction conflict in action. We think this additional overhead is caused in few of the transactions as DTM worker threads clear the intents in background and concurrent transactions are expected to take performance hits due to the nature of it. If the background thread sees an already committed/aborted transaction(conflict resolve might have happened before) it will return success to DTM without changing any state in the KV store.

Crash recovery in the coordinator is handled with a timer for each transaction. On recovery, the leader at that term would go over all the pending transactions and upon elapsed time greater than timeout aborts the transactions. Currently we have transaction timeout set to 5s. At cohort, on crash inflight transactions will be cancelled and relies on RAFT to recover the store state. We take snapshots of transaction records at DTM and key-value stores to reduce the time for recovery.

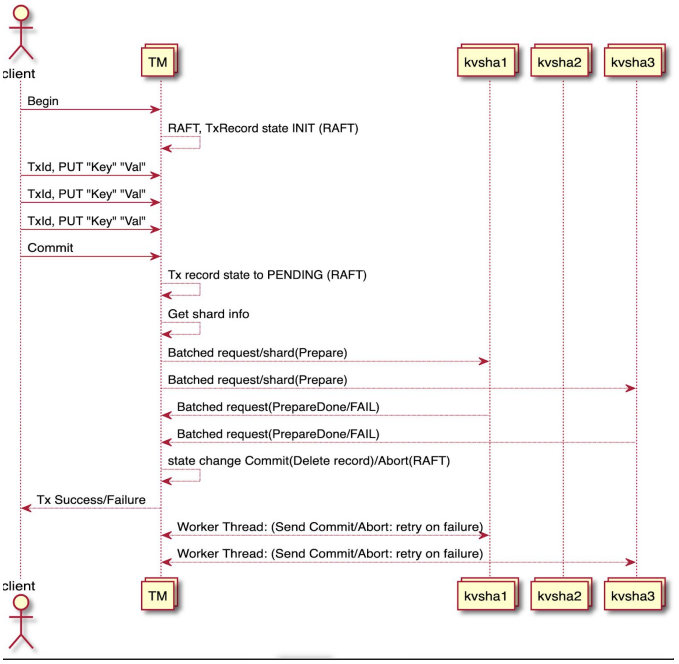

Fig 2: Simple write-only transaction

Key-value store supports read-write, write-only and read-only transactions. All types of transactions are handled by the store leader. Figure 2 goes over the flow of simple write only transactions without conflicts. This assumes DTM already has information about shards involved in the transaction from the replica manager. DTM establishes a gRPC connection with required shards while executing operations. In read operations with conflict, we check the transaction ID of the staged transaction with the current transaction. Since we use

monotonically increasing IDs. If a current transaction ID is greater than staged transaction ID then it would call transaction conflict resolve to decide the result else it

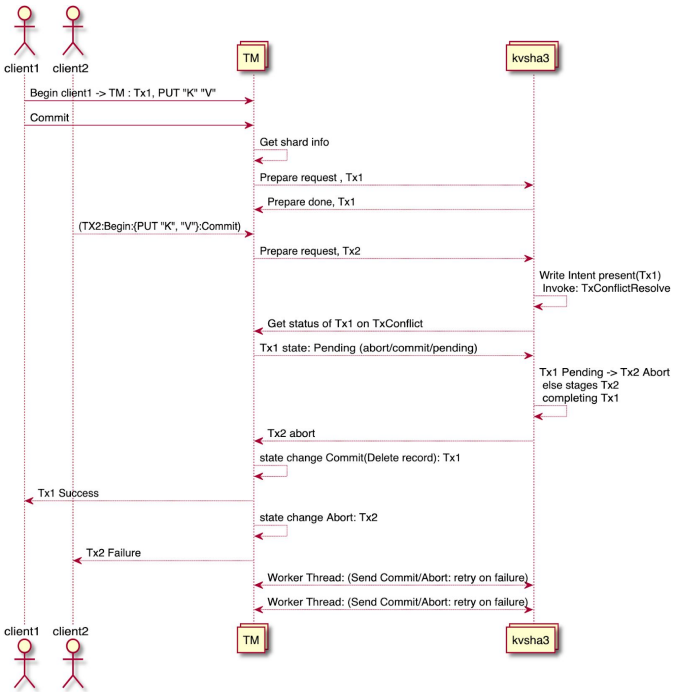

Fig 3: Concurrent transactions, resolving transaction conflict

returns the last committed value. If a transaction tries to read the previously written value in the same transaction, it returns the last committed value. The datastore does not support multiple writes of the same key in a transaction.

IV. PERFORMANCE EVALUATION

We deployed the system in Amazon EC2 instances. All the instances were in the same region. We were able to scale up to 7 Replica Servers with 3 TMs and 1 Replica Manager. In all the test cases, keys are numbers and values are their english words.

A.Throughput

Throughput is calculated as the number of transactions per Second. Clients are run for 1000 txns as go-routine(for concurrency) with txn per routine. We varied the number of operations per transaction over 3,5,7 Replica servers and 1,3 TM with 3,5,7 Shards.

As the number of operations per transaction increased, the throughput decreased for both read and write transactions [Table 1] as the keys in each operation had to be committed across multiple shards. As a future work, we should implement reads by reading from any shard member. This would improve in read performance.

TABLE 1 AVG TXNPER SECOND FOR 3 KV 3TX 5 SHARDS

op/txns	Without logs	
	Read	Write
3	290	250
10	130	120
20	90	80

Varying the number of shards we saw a throughput impact with 7 shards(Fig 4). However when we disabled the logs, the same was not observed. Probably with logs disabled, we might observe the same behaviour at higher number of shards.This is something that needs to be investigated further.

B.Latency

Read/Write Latency is measured in a 3 Replica Server setup with 1 Tx managers with 3 Shards for 500 txns sequentially. We are measuring latency as time taken to get a response from the Transaction Manager for an outcome.

Write Latency ranged between 18 - 22 ms per transaction.On a non-leader crash, transactions are not impacted significantly. [Fig 5]. If we crash a replica-server,which is the KV leader for all the shards in the system, we saw that 2% of the transactions resulted in ~200 ms latency[Fig 6] and those 2% transactions failed. We observed latency on leader crash depending on how the shard leaders are chosen. If shard leaders are different for different shards, then one or two transactions are affected. On killing one shard leader we saw the write latency to be 9000 ms. Read Latency ranged between 15 - 20ms per transaction.[Fig 7]

Number of TM affects the latency as well. We increased the TransactionManagers(TM) to 3 TM and we saw that the write latency went up to 30 ms. [Fig 8] On crashing the TM Leader, around 30% of the transactions failed. We did our experiment with 2000 txns and saw more than 600 txns fail [Fig 9] Crashing the non-leader TM or bringing up the crashed leader did not impact the transactions.

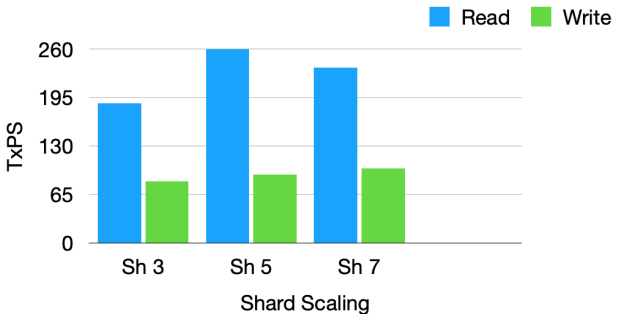

Fig 4. Throughput with logs enabled.

V. Learnings

Initially we overlooked the need for service discovery. We could have used something like etcd/consul for discovery. Using these services could have helped us to focus on adding other features to the system instead of writing our own RM. Client interactions are done via http client. We believe performance would have improved if we had used gRPC. Implementation of 2PC is more challenging than expected in crash recovery and leader change.

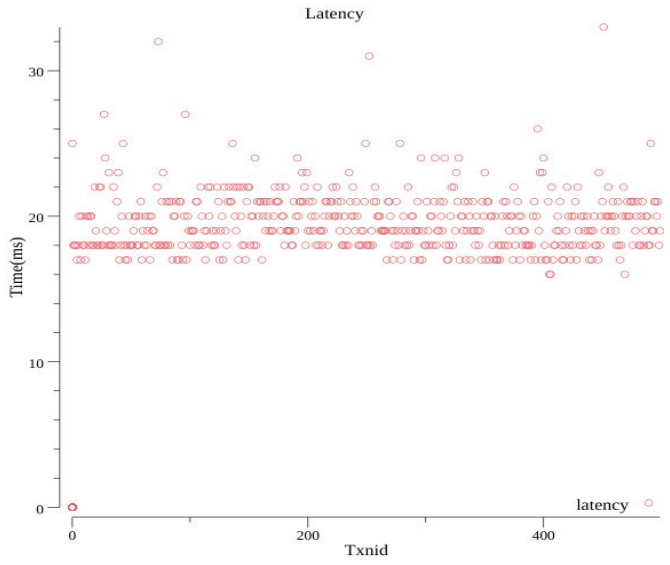

Fig. 5 Write Latency with Non Shard Leader Crash

VI. Future work

This project was a good experience to understand various components of a distributed system and the challenges involved in implementing them. We would like to explore the possibility of using MVCC for isolation and compare the performance with the TWo Phase Locking. We would like to try a unified consistency and atomicity model[5] and compare the performances. To improve read performance, we can implement reading from any shards. We want to use consistent hashing as a future extension and enable support for dynamic rebalancing of the nodes. Performance numbers from nodes across geographical regions are not evaluated in our current project. Potentially that will change the performance numbers. Also, we would like to collect more performance measurements such as tail latencies.

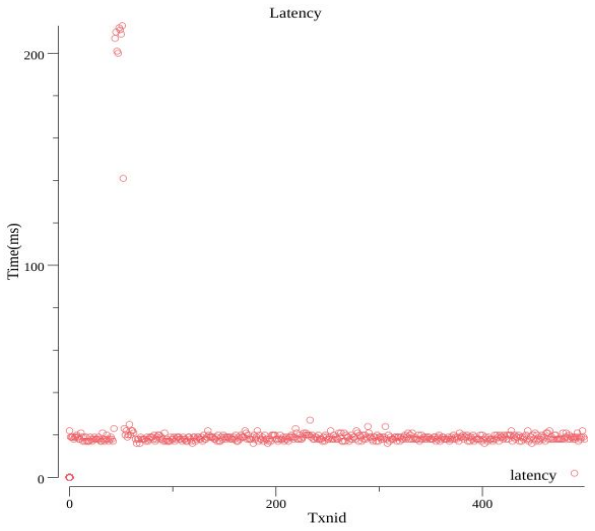

Fig 6 Latency with Shard Leader Crash

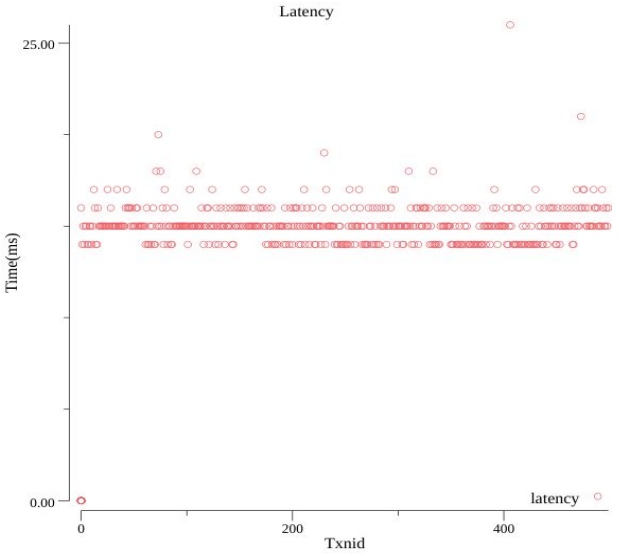

Fig.7. Read Latency

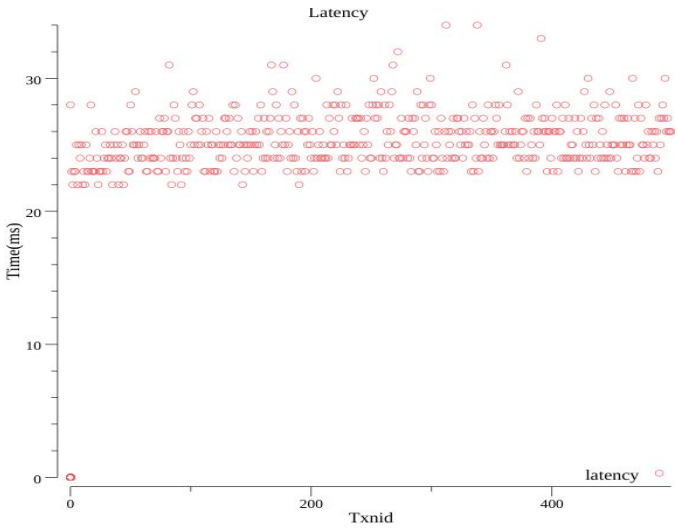

Fig 8- Write Latency with 3 Txn Manager

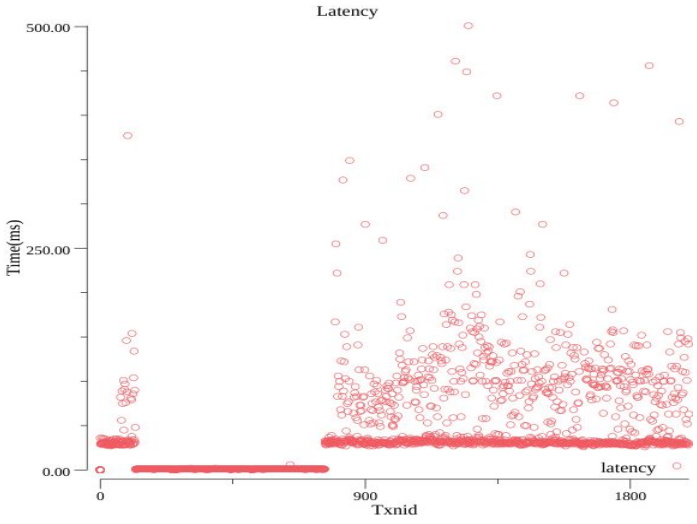

Fig 9-Write Latency with leader crash 3 Txn Manager

VII. Conclusions

We were able to implement an in-memory KV store which is distributed. We were able to add transactions on top of the KV store. In the process of implementing such a system we understood different moving parts in a typical distributed system. We learnt how availability and scaling affects the performance. Network performance plays a role in application performance. We got a better understanding of some of the concepts such CAP vs ACID, consistency and different kinds of isolation.

References

[1] https://github.com/etcd-io/etcd/tree/master/contrib/raftexample

[2] Matt Tracy, How CockroachDB does distributed atomic transactions, September 2, 2015

[3] https://github.com/pingcap-incubator/tinykv

[4] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford Google, Inc. Spanner: Google's Globally-Distributed Database

[5] Sujaya Maiyya, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi. Unifying Consensus and Atomic Commitment for Effective Cloud Data Management. PVLDB, 12(5): 611-623, 2019. DOI: https://doi.org/10.14778/3303753.3303765

[6] Butler Lampson and David Lomet, *A new Presumed Commit Optimization for Two Phase Commit*, 19th VLDB conference, Dublin Ireland, 1993.

[7] N.B. Al-Jumah, H.S. Hassanein, et. al Implementation and modelling of two-phase locking concurrency control- a performance study, Information and Software Technology, 1999