Fault tolerant Distributed Key Value Store

CompSci 510: Spring 2018
Final Report

Waqar Aqeel
waqar.aqeel@duke.edu

Sahiti Bommareddy
sahiti.bommareddy@duke.edu

Yang Yuxi
yuxi.yang@duke.edu

Abstract

We wanted to explore what is involved in designing a system that can be used in production. With that goal, we have integrated Raft based replication and consensus into an existing distributed key value store to make it fault tolerant. The design of the system is based on optimistic concurrency control in a distributed key value store, inspired from Thor[9]. We have added Raft[2] consistency control algorithm into each store to increase availability and reliability. We test this implementation with various scenarios and see how the system responds to different load configurations and failures. We also discuss the advantages, and limitations of this system implementation and ways to overcome them to make it good for production .

1. Introduction

Database system has become a vital component in many service oriented businesses and organizations, such as banks, airlines, and hospitals[1]. In database system, the main concerns of a Database Management Systems (DBMS) are to optimize performance, availability, and reliability of the system. The two main DBMS functions then are: security of the data and correctness of the data.

There are two popular ways to ensure correctness: first is to use two-phase commit to do atomic transactions, second is to use replicas to prevent the crash of a single Database node from bringing down the entire system.

The rest of this paper is organized as follows: Section 2 discusses related work, Section 3 explains the optimistic concurrency algorithms used in the distributed system and the Raft consensus algorithm. Then, Section 4 discusses our system design and Section 5-7 demonstrate the different situations we considered in this system and how the system reacts to them.

2. Related Work

There has been a lot of previous research about the concurrency control[1,10] and consistency control mechanism[2].

Dynamo[7] is interesting as it is a highly available key-value storage system. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. This is because it is not possible to guarantee that there will not be a network partition, and in case of a partition, Dynamo makes it a priority to keep the system available. So, consistency is sacrificed, but only for a short time i.e., eventually consistent until the other replica databases catch up. It makes extensive use of object versioning and application-assisted conflict resolution. Cassandra[6] is a distributed storage system that provides highly available service with no single point of failure. It uses optimistic concurrency control and resembles Dynamo in a lot of ways.

The consensus algorithms in distributed systems usually use message passing to support communication and synchronization. Through these synchronization mechanisms, varying degrees of consistency are supported[8]: strong consistency, eventual consistency, consistent prefix, bounded staleness, monotonic reads, and read my writes. Following is the previous research based on strong consistency using message passing. In 1989, Leslie Lamport first introduced Paxos[4], a strong consensus algorithm that had strong similarities with Barbara Liskov's 1988 ViewStamped Replication[5] work. ViewStamped Replication used a primary node to process the request from the client,and kept a group of state machine replicas to prevent the crash of the primary node from bringing the entire system down. Consensus for the primary node designation and other operations is simply based on agreement among a majority of replica nodes. Then comes the Raft consensus algorithm[2], which makes it a priority to simplify the protocol. Similar to the previous two algorithm, Raft provides strong consistency and provable safety. Raft will be described in detail in Section 3.2.

3. Algorithms

The first stage of the project involves an existing distributed key value store. This distributed store is designed for an environment with low to medium contention so that the number of conflicts remains low. Hence, the design choice of optimistic concurrency control[9]. The second stage involves bolstering this extremely unreliable system by replication, and log recovery through Raft, and further extending it to improve balance load.

3.1 Concurrency Control

In absence of locking we need an algorithm at database side providing the same guarantees that locking is capable of. This is achieved by the database through a history line to track transactions during their two phase commit and a validation algorithm. When a transaction makes a write request and succeeds in phase 1 during our validation, we log this into history line with status as `prepared`. Other incoming concurrent transactions are validated against these `prepared` transactions which the system has already promised to commit. Hence, in case of a conflict we abort the current transaction. Once a transaction has passed the second phase of 2PC, we change its status to committed, and reflect the changes in KV store where we increment the key's version number and delete it the transaction from the history line.

The validation algorithm being used in this system is as follows:

Threshold Check

```
If T.ts < Threshold then
Abort T
```

Checks Against Earlier transactions

```
For each uncommitted transaction S in history line
such that S.ts < T.ts
If (S.MOS ∩ T.ROS is NOT NULL) then
Abort T
```

Current- Version Check

```
For each object   in T. ROS
If each object’s version in T.Ros is not same as version in database server or t.ts<rstamp then
Abort T
```

Checks Against Later Transactions

```
For each transaction S in history line
such that T.ts < S.ts
If (T. ROS ∩ S.MOS is NOT NULL) then
Abort T
```

Ts-timestamp
MOS-write set /keys that are modified
ROS- read set/ keys that are modified
T-transaction being validated
S-Other transaction already in the history line

3.2 Consensus - Raft

Raft is an algorithm that achieves consensus via an elected leader. The leader is responsible for log replication to the followers. It regularly informs the followers of its existence and liveness by sending a heartbeat message at regular intervals. Each follower has a timeout usually slightly longer than heartbeats interval during which it expects to receive a heartbeat message from the leader. Each group of replicas, henceforth called cluster, has one and only one elected leader who is fully responsible for managing log replication on the other servers of the cluster. It means that the leader can decide on new entries placement and establishment of data flow between itself and the other servers without consulting the other servers first. A leader leads until it fails or disconnects, in which case a new leader is elected.

The safety of raft system is kept with the following principles:

- The leader will never overwrite an existing log entry. The only operation it can do to logs is to append at the end. In addition, the log can only be added by the leader and copied to the follower, never in reverse.
- If a follower’s log is newer than the candidate’s, the follower will refuse the VoteRequest, and will not vote for this candidate.
- After ensuring that a log entry has been safely copied to a majority of followers, i.e.the AppendLog message triggers a success response on a majority of DB servers to the leader, the leader concludes that the log entry has been successfully committed and updates its log index.

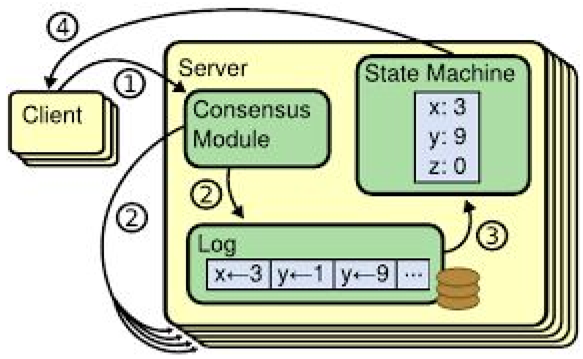

Figure 1: Transaction execution flow through which safety of system is preserved.

4. Distributed Key Value Store (Existing architecture)

We harvested an existing system from one of our previous projects. This system implements two phase commit over a set of stores and has no replication or fault tolerance at all. If a single database fails, the entire system fails, bringing the throughput to almost zero.

4.1 Client

A client consists of the application and a clerk library attached to it. The clerk library is capable of dividing up a transaction into multiple servers based on a mapping that is served by the coordinator. The clerk then sends out 2PC `prepare`, `commit` or `abort` messages for the generated transaction. The client is always connected to the coordinator, which can serve mapping updates to the client if there are any. It also caches these server mappings and object values that it has read.

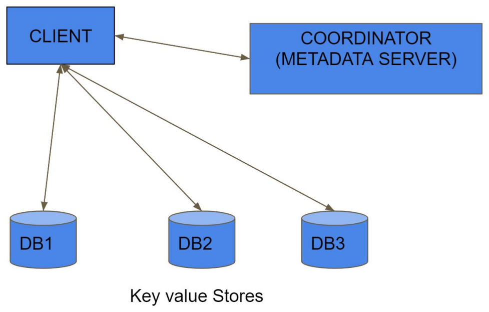

Figure 2: Architecture of Distributed key value store

4.2 Coordinator

Our initial implementation of coordinator is inspired from GFS Master in Google File System[3].Each object has a globally unique id called `oid`. Every object has a database server cluster as its primary responsible for the object. The oids are distributed among the databases by coordinator and the object-cluster mapping is stored on the coordinator. The coordinator, upon receiving a WhoServes message from client (with a list of oids) replies to the clerk with a map of primary servers for the oids requested.

The coordinator is also responsible for generating timestamps which are needed by clients just before the start of phase 1 of two phase commit. These timestamps ensure that transactions that are divided among multiple clusters are ordered properly. They are used to synchronize the history lines of multiple server clusters.

4.3 Database Server

The database server is a key value store which is equipped with the above history-line-based validation algorithm and capable of performing a 2PC. The entire keyspace is sharded on to multiple database servers.

5. Architecture Enhancements and implementation

Raft algorithm has four parts - the replica state machine, the leader election algorithm, the log replication, and security. We discuss our implementation of these features with some changes and trials in our system.

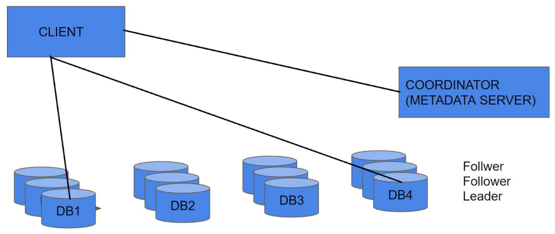

Figure 4: Our enhanced architecture

5.1 Replica State Machine

Our coordinator is configured to set up the clusters and to do keyspace sharding. The coordinator has parameters that can be used to configure the number of clusters, the size of each cluster and key sharding techniques. It sets up clusters and updates its own cluster maps, indicating which oids are maintained by which clusters. It also populates the routing maps in each database server with members of the cluster. It also sets the random timeouts of each database and heartbeat interval.

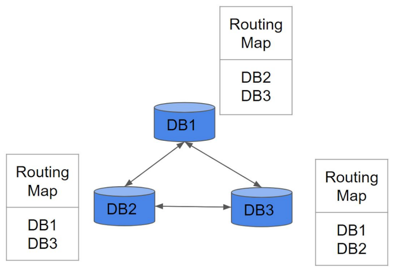

Figure 3: Our cluster design

5.2 Leader election

Our coordinator starts each server in follower mode. Since each server has a random timeout, one of the servers times out first, changes its role to candidate, and sends out VoteRequest messages to members of its own cluster. The members validate that this request is for a term greater than the current one, and that the latest log index included in the VoteRequest message is equal to or greater than the one in their log. If both these conditions are satisfied, the server for the candidate by sending a Vote message. Once the candidate determines that it has received a majority (> cluster-size/2) votes, it changes its role to leader and sends out a heartbeat with new term number to all members of its cluster. Upon receiving the heartbeat message from the leader, the followers reset their timers and any candidates or previous term leaders step down to become followers.

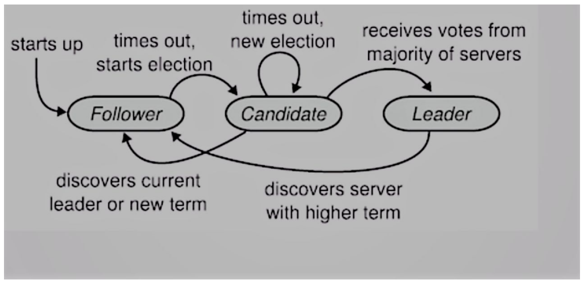
starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 5: State transitions

5.3 Logging and Recovery

During transaction execution, the leader performs transaction validation and decides whether to commit or abort a transaction, depending upon whether the transaction passes the validation tests. If the transaction is committed, the leader informs all cluster members through a LogAppend message. Followers append to the log and acknowledge the same after checking its own log.

If a follower, upon receiving a LogAppend message, finds out that the log index of the current transaction is more than one greater than the length of its own log, it concludes that it has missed some LogAppend message. If a database server falls behind like this, it sends LogRecoveryRequest message to the cluster leader with the help of its routing map. This request includes latest index present in the follower's own log. The leader then replies with a set of transactions that the follower has missed based on the index that the follower sent. Upon receiving a LogRecovery reply, the follower appends received entries to its log and performs the transaction writes in its KV store. Once it catches up, it starts executing the next LogAppend messages normally.

5.4 Configuration changes handling

In case an the current leader goes down or there is a network failure, followers stop receiving heartbeat messages and , after a timeout, an election is triggered. A new leader is elected based on the methodology described above. After gathering enough votes, the new leader notifies the coordinator of the change. The coordinator in turn sends mapping updates to clients. This prevents the unnecessary transaction aborts due to out of date map cache in clients.

6. Implementation details, tests, logs and code

We realised the system in Python using the Thespian actor library. This is a library that supports the Actor model along with its idioms. Thespian strictly adheres to the concept that there should be no `ask` among actors to maximize parallelism. This replicates real world scenario as the different components involved i.e. the client, database servers, and coordinator can be modeled as actors communicating with one another through messaging passing. Each actor is spawned as a separate Python process because of Python's limited multithreading capabilities, and communicates with other actors by sending TCP segments. This has the added advantage that no actors share memory and it becomes a truly distributed system capable of running on completely separate machines.

Since there is no `ask` between actors, we ended up using the scatter-gather pattern to coordinate among multiple clusters and multiple servers inside a cluster. This means that a server never blocks unless its message queue is empty and thus system utilization is as high as possible. The heartbeat timeouts are random timers that are initialized when each server is started as an independent process. The different types of messages that these servers and the coordinator send to each other are detailed below:

Messages during election between members of cluster and coordinator-

1. Heartbeat(term) - The leader sends heartbeat message to followers to inform them of its liveness. The heartbeat is sent every t seconds, where t is as configured. When a candidate or follower sees a heartbeat with higher term number, they will become followers and update.
2. VoteRequest(term, candidate, latest log index) - When follower times out without receiving heartbeat, the followers turn into candidates automatically and sends out vote request with incremented term number and its last log index. A candidate becomes a leader when it receives more than half of the votes.
3. Vote()- The servers send Vote() messages to the candidates they vote for. This is decided by a First-Come-First-Serve(FCFS) algorithm and only if leader is up to date
4. LeaderChange(cluster,leaderName) - After a leader has been elected, it sends this message to the Coordinator to inform it of the new leader, and the Coordinator, upon receiving this message informs clients of the change.

Messages during recovery between cluster members-

1. RepairLogReq(latest log index) - when a member wants to catch up on missed transactions, it sends this message to the leader of the cluster with the help of its routing map.
2. RepairLog(entries) - On receiving the RepairLogReq message, the leader checks its own log and send the missing entries to the requester.

Messages during normal operation between leader and follower-

1. AppendLog(index, entry) - When the leader commits a transaction, it sends the this message to the followers in its cluster along with the log entry thatt it has just created.

In our implementation, we chose to provide the leader with complete control over log, in the sense that it not only appends to log but is also capable of rewriting the history.

We simulated situations to test the system's resilience. When there are multiple candidates in an election, this contest is resolved as followers vote on first come first serve basis. Also, in case of a failed election, there is a term increment and re-election happens. To test the failure of a database server, we made a member sleep for certain duration and then wake up. When it wakes up, it starts receiving heartbeat messages, so it discovers its leader. Upon receiving an AppendLog message, it realises that it is lagging behind based on the index, and sends out a LogRepairRequest to the leader. The leader replies with missing transactions, the server catches up and resumes normal behavior. We also checked that the clients' cached mappings are updating when leader change occurs. We are also able to configure the co-ordinator to direct read request to non leader member of clusters but write requests are exclusively handled by the leader. We also started experimenting as to how much of consistency we can give up and increase availability of the system. The logs of system behaviour in these scenarios uploaded along with the source code on this git repo:

https://gitlab.oit.duke.edu/510-project/510-replication

7. Future scope

Future directions of this work could include replicating the coordinator itself so there is more fault tolerance. Based on our current implementation, the leader in each cluster is likely to get overloaded since its responsible for both client communication and log recovery. The leader could delegate log recovery to one of its up-to-date followers so as to reduce its own load. The coordinator could also be configured to monitor utilization on the leaders so an election is triggered preemptively before the leader gets overloaded. Since a cluster is unavailable during a leader election, this could increase availability.

References

1. Concurrency control and recovery, Michael J. Franklin, SOGMOD, 1992, chapter 3 pp1.
2. D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm.," in USENIX Annual Technical Conference, 2014, pp. 305–319.
3. S. Ghemawat, H. Gobioff, and S.-T. Leung, The Google file system, vol. 37. ACM, 2003.
4. L. Lamport, "Paxos made simple," ACM Sigact News, vol. 32, no. 4, pp. 18–25, 2001.
5. B. Liskov and J. Cowling, "Viewstamped replication revisited," 2012.
6. A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," ACM SIGOPS Operating Systems Review, vol. 44, no. 2, pp. 35–40, 2010.
7. G. DeCandia et al., "Dynamo: amazon's highly available key-value store," in ACM SIGOPS operating systems review, 2007, vol. 41, pp. 205–220.
8. Replicated Data Consistency Explained Through Baseball, Doung Terry, MSR Technical Report, 2011
9. "Efficient optimistic concurrency control using loosely synchronized clocks" by Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. In ACM Conference on Management of Data (SIGMOD), (San Jose, CA), June 1995, pp. 23-34
10. B. Liskov, M. Day and L. Shrira, Distributed Object Management in Thor, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, USA, 1993.