RAFT based Key-Value Store with Transaction Support

Chen Chen, Varun Kulkarni, Supriya Premkumar and Renga Srinivasan
Stanford University

Abstract

In this project, we present a highly consistent, fault tolerant distributed key-value store. It adapts the raft consensus algorithm broadly in the system and supports concurrent transactions across shards. Each operation on the store is handled by a set of coordinators as a raft group. The keys in the store are partitioned across shards and each shard maintains multiple replicas forming their own raft groups. The distributed transaction across shards is achieved using two-phase commit protocol with two-phase locking to guarantee atomicity and serializability.

1. Introduction

Highly available, consistent, fault-tolerant key-value stores are at the center of modern applications and distributed database systems. Most NoSQL systems [1,2] inspired by Amazon Dynamo [1] does not guarantee strong consistency in favor of availability due to the trade-off described in the CAP theorem [3] or the more generalized PACELC theorem [4]. However, more recent systems, such as Spanner [5], CockroachDB [6] and TiKV [7], employ the mechanism of using consensus protocols to enforce consistency across multiple replicas of the data and mitigate the non-availability.

Motivated by the idea, this project focuses on building a distributed key-value store that is strongly consistent and fault tolerant [1]. It uses the raft consensus algorithm [8] to ensure fault tolerance in both coordination and data replicas. In addition, we adapt two-phase commit (2PC) [9] and two-phase locking (2PL) [10] to support multi-shard transactions that guarantee atomicity and serializability.

The paper is structured as follows. Section 2 presents the system design and section 3 describes the implementation. Section 4 evaluates the performance of the key-value store and section 5 concludes the paper.

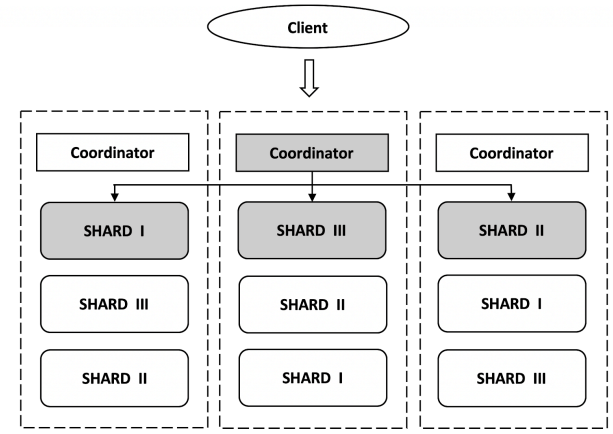

Figure 1: Architecture of raft based key value store

2. System architecture

In this section, we present the architecture of the system as shown in Figure 1, consisting of the following components: coordinator, data store and client.

2.1. Coordinator

The coordinator acts as a transaction manager to the key-value store. It is responsible for executing client requests and exposes the operations on the data store to the clients via APIs described in section 3.1. To ensure fault tolerance, the system manages $2f + 1$ coordinator replicas forming a raft group, where f is the number of failures that a raft group can tolerate. The raft consensus algorithm [8] ensures that coordinators are always available as long as the majority of the replicas in the clusters are available. For every request sent by clients, it reaches the leader of the coordinator. The coordinator then maps the request to the corresponding shards using a simple hash function, which equally distributes the keys across shards. The coordinator leader is also responsible for figuring out the leader of each shard

[1] https://github.com/SCPD-Project/raft-kv-store

and relays the request. In addition, the finite state machines of transaction are replicated among the coordinator replicas using raft.

Besides, the coordinator helps coordinate transactions on the key-value store described in section 2.2. When the coordinator leader receives a transaction request, it initiates the communication following the 2PC protocol [9] to the relevant shards and drives the transaction until its completion. In the case of coordinator leader failure during the transaction, a newly elected leader will help recover the transaction. This ensures that the 2PC never gets stuck and every transaction gets a closure eventually.

2.2. Data store

In the data store, the key-value pairs are partitioned into a configurable number of shards. Each shard consists of *2f+1* replicas as a raft cluster. Within each replica, the data are stored in an in-memory map and being periodically persisted to disk using Boltdb [11] for efficient snapshot and recovery.

In the current design, we implement a customized concurrent hash map (section 3.8) that has keys of string data type and values of integers. However, this could be easily extended to support any datatype. Similar to the coordinators, the raft consensus algorithm is also used to guarantee the strong consistency of the key-value states among the replicas.

2.3. Client

Client provides a user interface to interact with the coordinator. It is also responsible for routing to the coordinator leader. The details are described in section 3.2.

3. Implementation

In this section, we focus on the implementation details of the key-value store.

3.1. Overview

The client communicates with the leader of the coordinator through a HTTP protocol and the coordinator leader communicates with the data store using RPC protocol. The data are encoded as protocol buffer [12] for efficient transmission. Golang's native `net/http` and `net/rpc` packages are used to build the transport channels for the system.

For raft implementation, we evaluated several popular raft libraries in Go. Eventually, we zeroed in on the `hashicorp/raft` [13], that provides neat abstractions and makes it easy to build applications on raft. The raft library provides a callback function that gets called once raft entries are committed. At that point, the commands can be executed on the finite state machine. The library also provided callbacks for periodic snapshots and log compaction that allows the key-value store to be persisted on disk via these callbacks periodically.

3.2. Client implementation

Client stores the metadata of the coordinator leader in cache based on the last successful request. In case the leader changes and client reaches one of the coordinator replicas, the replica will send its leader metadata along with a 421 HTTP status to indicate a leader redirection. Client then updates its own leader information and retry its request with the new coordinator leader. In case of all other errors showing the leader is not available, client iterates its known list of coordinator nodes in a round-robin manner until it finds the correct leader or gets redirected. This process could be extended and optimized via automatic service discovery, which is outside the scope of the current implementation.

Below we present the supported operations in the client implementation.

GET, with syntax `get <key>`, fetches the value of a provided key. It returns the value of the key if the key already exists. Otherwise, an error of "key not exists" is returned to the client.

SET, with syntax `set <key, value>`, sets or overrides the key with the provided value. In some cases, an error may be sent back to the client if the coordinator finds the key is locked by other clients before timeout. Besides, we also support conditional set, `set <key, value, condition>`, so the write is executed only if the `condition` is satisfied. It is useful in ADD/SUB, XFER described later in this section and coupled with optimistic locking described in section 3.6 in practice.

DEL, with syntax `del <key>`, removes the given key from the key-value store . Note that we make this an idempotent operation, so it does not complain even if the key does not exist.

TXN, with syntax `txn <commands> end`, sends a set of simple operations including GET, SET and

DEL in a single transaction block to the coordinator. Based on what is provided in `commands`, it applies optimization rules to remove the redundant commands before sending to the coordinator. For example, if a client gives two SET operations on the same key in a single transaction, the first SET operation will be annihilated.

ADD, with syntax `add <key, value>` is used to increase the value of the current key by a certain amount. Internally, it splits the operation into two stages, including the simple operation GET and a conditional SET. In the first stage, it gets the current value of the given key. In case the key does not exist, it shows the error to the client. Otherwise, it follows with a conditional SET operation in the second stage that guarantees that the write is executed only if the value remains the same as the one obtained in the first stage.

SUB, with syntax `sub <key, value>` is an alias of `add <key, -value>`.

XFER, with syntax `xfer <fromkey, tokey, amount>`, transfers a specified amount from source key to destination key. It is similar to ADD/SUB operations, but the keys involved can be across multiple shards. It uses two TXN operations as basic building blocks, one read-only transaction followed by two conditional SETs. In the first read-only transaction, the client requests values for both the keys. Besides, we check the value of the source key is no less than the transfer amount specified to imitate a real transfer between two bank accounts. If the validation fails, the transfer is then aborted. Otherwise, two conditional SETs are sent in a single transaction to update the values of both of the keys. More details will be described in section 3.6.

3.3. Simple operations

We classify SET, GET and DEL as simple operations as they involve a single key. When the coordinator leader receives the request, it finds the destination shard, discover the leader of the shard and relays the request. Except the GET request, the data store then tries to commit the entry in the raft log.
Once the log entry committed, i.e. it is successfully replicated across replicas, the key-value store executes the corresponding command on the in-memory map.

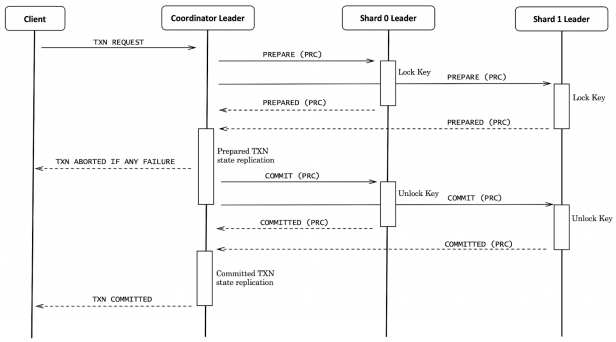

Figure 2: Two-phase commit diagram of a transaction request

3.4. Transactions

For each transaction request, the coordinator generates a unique transaction ID, `txid`. The coordinator maintains a map to keep track of all the transactions indexed by `txid`. The map acts as a finite state machine for the coordinator, which tracks the states of each transaction (*prepared*, *committed*, *abort*) as shown in Figure 2. Every operation on the map is replicated by raft across the coordinator replicas.

The transaction requests are handled by the coordinator with 2PC [9] and 2PL [10] across multiple shards. As in a typical two-phase protocol, the *prepare* phase is to ensure the shard leaders (cohorts) are available and ready to execute the transaction. The shards respond to the coordinator with the *prepared* message if it 1) acquired all the locks for the keys in the transaction (section 3.8), and 2) satisfied the conditional checks on the operation, if exist (section 3.6). Once the coordinator receives *prepared* messages from all the cohorts, it sends a *commit* message to commit the transaction. Otherwise, it sends an *abort* message to all the cohorts and releases the locks.

3.5. Read-only transactions

A read-only transaction is a transaction that consists of only GET operations on the key value store. This can be on a single shard or across shards. Since the GET operations does not alter the key value store, there is scope for optimization. In our implementation, the coordinator figures out if the transaction is read-only and marks it in the *prepare* message to the cohorts. The cohorts return the values of GET operations in the response. Once the coordinator receives responses from all the cohorts, it skips the

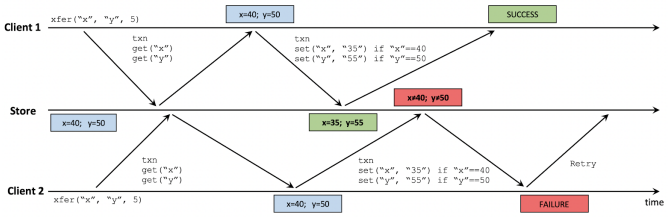

Figure 3: An example how optimistic locking works when two clients send XFER requests simultaneously

commit phase and replies back to the client immediately. This might result in stale reads on the client side. However, in our implementation the read-only transactions are only used for compound operations like ADD, SUB and XFER that support conditional set operations. This helps alleviate the problem of stale reads and the client can retry to operate on the latest data.

3.6. Optimistic locking

As described in section 3.1, compound operations, XFER, ADD and SUB, consist of a read-only transaction followed by a conditional SET transaction. In order to ensure compound transactions are atomic and externally consistent, the second transaction must always operate on the latest data. In our implementation, this is achieved by means of coupling conditional SET with optimistic locking [14]. It can be also treated as a distributed *CompareAndSwap* operation. During the *prepare* phase of 2PC, the condition is validated by cohorts when fetching locks as described in section 3.8. The cohort agrees to execute the transaction and sends back *parepared* message to the coordinator only if the condition holds.

Figure 3 demonstrates an example that two clients simultaneously request for a transfer operation on two keys. In this case, Client 1 succeeds because it passes the conditional SET. Client 2 lags behind and fails in the conditional SET but will eventually succeed on retry.

3.7. Transaction recovery and garbage collection

As discussed in section 3.4, coordinator replicates its state using raft to all its replicas. When a new leader is elected, it can recover any pending transactions by consulting its state. In our implementation, we listen for leadership changes in a separate background thread. When the raft notifies leadership changes, the new leader scans its map and recovers all transactions that are not in *committed* state by sending appropriate *abort/commit* messages depending on the transaction state.

In addition, the same background thread garbage collects the data of all the committed transactions to prevent the coordinator finite state machine from growing infinitely [9].

Waiting Time	Single lock	Fine grain locks
1 ns	2,339 ns/op	364 ns/op
1 µs	7,577 ns/op	443 ns/op
1 ms	1,214 µs/op	408 ns/op

Table 1: Performance benchmark comparison

3.8. Concurrent map

We implemented a concurrent map to both support concurrent writes and 2PC transactions. To minimize the waiting time of locking, we adapt the fine grain locking strategy, that is, one global lock for the entire map and one local lock for each key-value pair respectively, all of which are readers-writer locks. The idea is, once the local locks are obtained, the global lock should be lifted at the earliest possible time to unblock other transactions. Note that the waiting time of 2PC is not negligible, in that it is spanned across two phases, the *prepare* and *commit*/*abort* including one round-trip data transmission between the coordinator and cohorts. Table 1 shows a performance benchmark comparison of concurrent writes between our concurrent map implementation and a map with a single lock only. It is observed that while the writes running time grows exponentially as the waiting time increases in the naive implementation, it remains the same in the fine grain implementation. Even if the network latency is as low as 1 *ns*, it shows significant improvement in efficiency, ~ 6.4 times.

`TryLocks` and `Abort` are two important interfaces implemented to serve the purpose of 2PC. `TryLocks` tries to acquire the global and local locks with two configurable timeouts, `GlobalTimeout` and `LocalTimeout`. The new keys, if there are any, are created with a temporary tag with locks attained. Besides, all the keys in the transaction are also tagged with the `txid`. `TryLocks` is determined to be successful when 1) all the locks are acquired before timeouts, and 2) all the conditional checks described

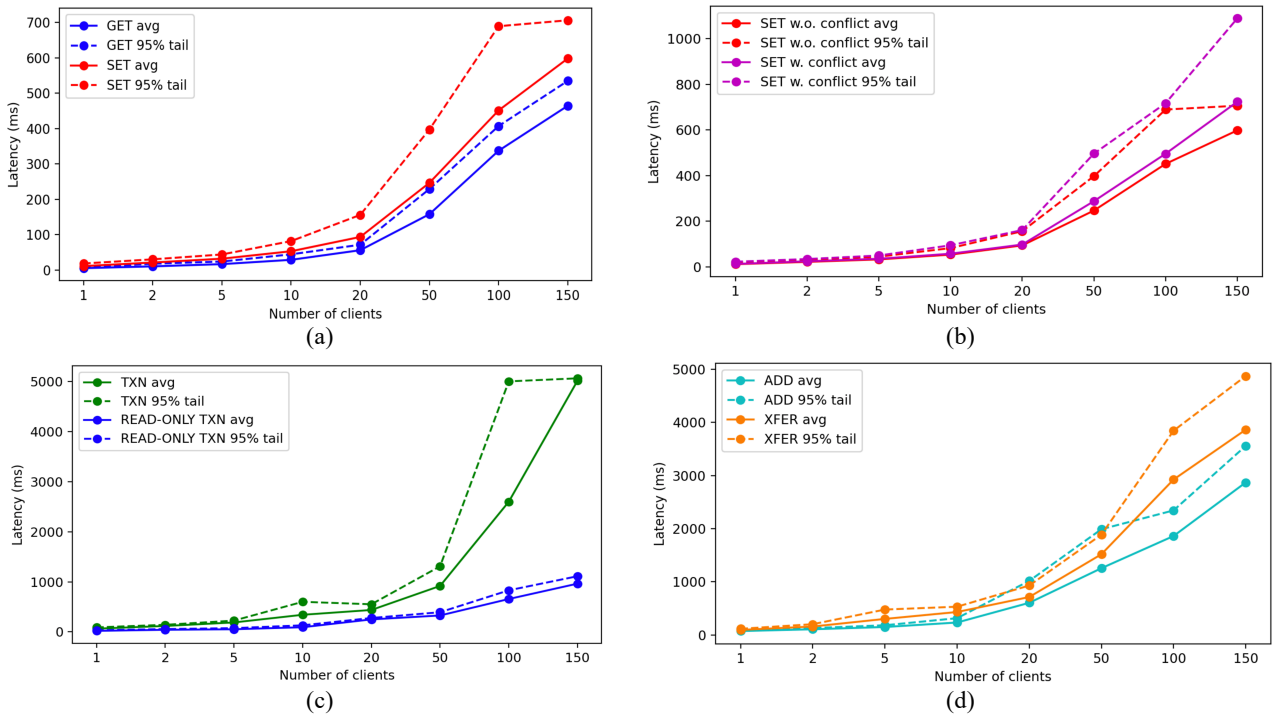

Figure 4: Latency of each operation

in section 3.6 are satisfied. Otherwise, the map and relevant locks are rolled back to the state before `TryLocks` was called. In either case, the global lock is lifted as soon as possible. `Abort` is called when the coordinator decides to abort the transaction. It uses the temporary tag to distinguish the new and pre-existing keys. The former ones are removed, and the later ones are unlocked only if its attached `txid` matches with the transaction. This check is necessary as it guarantees that the locks are only lifted by the same transaction that attained them.

We started off with both the `GlobalTimeout` and `LocalTimeout` as constant but found that it can lead to deadlock when multiple transactions for the same keys happen to attain mutually exclusive locks across shards. To resolve the issue, we set `GlobalTimeout` as a fixed constant but `LocalTimeout` as randomized duration, for each transaction, in the same order of one round-trip data transmission time. In this case, even if the deadlock occurs in the first place, once the shorter ones time out, the transaction with the longer timeout can eventually acquire the locks and proceed to the next phase.

4. Performance

This section presents an evaluation that establishes a baseline system latency and a fault tolerant analysis that explores the impact of leader failure. For experiment, we launch our system with three docker containers to simulate the server nodes in practice. Each container consists of three processes, one for the coordinator and the rest for two different shards. In addition, there is one more container for the clients to request to the system.

4.1. Latency

To measure the latency of each operation supported, a fixed number of clients are set up and each of them generates new requests to the coordinator once it receives the feedback of the last request for a duration 30 *s*. After it finishes, the latencies of each request are gathered across clients over the duration to form a distribution of latency. We examined the case of 1, 2, 5, 10, 20, 50, 100 and 150 concurrent clients to see the impact of the number of clients on the latency.

Figure 4(a) illustrates the latency of GET and SET operations where no same keys are requested at the same time on the servers. It is seen that the request latency grows for both GET and SET as the number of clients increases. The SET operation takes longer to respond than GET, as is expected with readers-writer locks mentioned in section 3.8. Besides, the gap between the average and 95th percentile response time is larger for SET than GET, indicating a heavier tail in

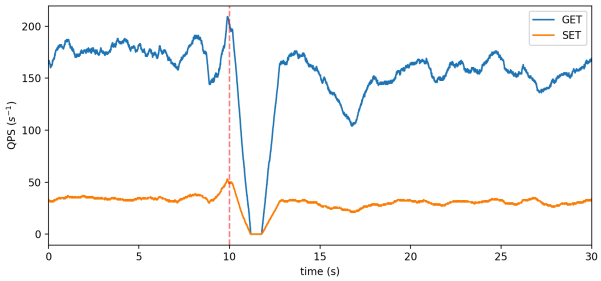

Figure 5. Fault tolerant test showing recovery after failure

the latency distribution for writes operations. Figure 4(b) shows the impact of multiple clients sending SET requests on the overlapped keys. Although the average and tail response time increase when key conflicts happen, the increment is not as large as the difference between GET and SET.

Besides, we also investigated the latency of read-only vs a typical 2PC transaction. Both types of the transactions are guaranteed to be across two shards in this experiment. As shown in Figure 4(c), 2PC transaction puts a great burden due to its coordination algorithm compared to the read-only transactions that only involves *prepare*/*prepared* phase. This is, however, indeed a tradeoff between the availability and richness of transaction semantics as expected. The tendency becomes more obvious when the client size grows. When the number of clients approaches 150, both of the average and 95 tail latency reach 5 *s*, which is the HTTP request timeout set for the clients. We also explore the latency of operations ADD and XFER that uses optimistic locking as shown in Figure 4(d). Given that both of the operations consist of two transactions internally and require multiple rounds of communication as mentioned in section 3.4, even the ADD operations that involve a single shard are relatively expensive to support.

4.2. Fault tolerant

In this experiment, 5 clients are launched concurrently, of which 4 clients keep sending GET requests and the rest one keeps sending SET requests. We stop the docker container containing the coordinator leader 10 s after the experiment starts in order to simulate a node failure. Figure 5 shows the traffic rate before and after the container stops. With the stop of the container, both GET and SET traffic drop down to zero for less than a second and then recover to the normal state, which is as anticipated for a fault tolerant system. The slight decrease of traffic rate after recovery can be explained by the fact that the shard leaders happened to be in the same container as the coordinator before the failure but spread across containers after the leaders election.

5. Conclusion and future work

This paper presents a strongly consistent, fault tolerant transactional key value store using Raft and 2PC. A few improvements in the areas of service discovery to find leaders of raft groups, consistent hashing to support equal redistribution of keys during configuration changes can improve the efficiency and robustness of the system. Our performance tests also indicated that 2PC with presumed commit optimizations is still slow. To further improve performance, we can extend the system to support Google's Percolator transaction model [15] which is a variant of 2PC. Overall, the project gave us an opportunity to understand the trade-offs and challenges involved in implementing a scalable distributed system.

Acknowledgement

We would like to express our thanks to David Mazières and Jim Posen for constant guidance and support throughout the quarter.

References

[1] Sivasubramanian, Swaminathan. "Amazon dynamoDB: a seamlessly scalable non-relational database service." Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. 2012.

[2] Cassandra, Apache. "Apache cassandra." Website. Available online at http://planetcassandra. org/what-is-apache-cassandra 13 (2014).

[3] Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." Acm Sigact News 33.2 (2002): 51-59.

[4] Abadi, Daniel. "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story." Computer 45.2 (2012): 37-42.

[5] Corbett, James C., et al. "Spanner: Google's globally distributed database." ACM Transactions on Computer Systems (TOCS) 31.3 (2013): 1-22.

[6] Taft, Rebecca, et al. "CockroachDB: The Resilient Geo-Distributed SQL Database." Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020.

[7] https://tikv.org/docs/3.0/concepts/overview/

[8] Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." 2014 Annual Technical Conference (14). 2014.

[9] Lampson, Butler, and David B. Lomet. "A new presumed commit optimization for two phase commit." VLDB. Vol. 93. 1993.
[10] Bernstein, Philip A., Vassos Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. Vol. 370. New York: Addison-wesley, 1987.
[11] https://github.com/boltdb/bolt
[12] https://developers.google.com/protocol-buffers
[13] https://github.com/hashicorp/raft
[14] Halici, Ugur, and Asuman Dogac. "An optimistic locking technique for concurrency control in distributed databases." IEEE Transactions on Software Engineering 7 (1991): 712-724.
[15] Peng, Daniel, and Frank Dabek. "Large-scale incremental processing using distributed transactions and notifications." (2010).