Distributed Key-Value Database using RAFT Consensus Algorithm for Consistency Control

(Article: https://raft.github.io/raft.pdf)

Benjamin Gee - v5e0b
Jiyoon (Yoony) Ok - p7x8
Kevin Mienata - b9c9
Jonathan Kim - p2v9a

Introduction

1.1 Overview

This project we aimed to design a distributed key-value store database. The objective of a distributed key-value store is to multiple copies of the database distributed across the network. The data in each one of the copies should remain consistent at all times. For example, if a client writes to a store, that information needs to be replicated to all other stores such that if another client reads from that key from another store it should return the written value.

1.2 Problem

The major problem we aim to solve in this project is how to keep data consistency through the store network. Consistency in database systems means that the data across all stores in the network should be the same. Any writes to one store should somehow reach every other store in the network and any reads to any store should return the same value. This means that our transactions must follow ACID database principles. In this project, we aimed to follow the atomicity, consistency and isolation principles by following the solution outlined in the RAFT consensus algorithm (https://raft.github.io/raft.pdf).

1.3 Solution

Atomicity

Atomicity states that a transaction is an “all or nothing” process, meaning that if a part of a transaction fails, the entire transaction should not be performed and the database state should not change. This is accomplished with our 2-phase commit and logging implementation (see Section 4.2). In brief, when a transaction is performed, it first undergoes a process of priming the network to receive a transaction by logging it as “uncommitted”. Once the network responds and is ready, the transaction is logged as “committed” and written into the database.

Consistency

In order to keep data across all stores in the network consistent, we used RAFT’s leader nomination and rollback mechanism. A leader is a store who is nominated to be the single point of entry and exit for all transactions. This means that any read (except fast read) or writes to the database must retrieve a value or write a value to this leader store first respectively. This leader store is responsible for disseminating the transaction to all other stores in the network so that each store remains consistent with respects to the leader. The rollback mechanism is used to replicate the logs with respects to a new leader whenever a leader nomination process is triggered. More detail about leadership nomination and rollback mechanism in Section 3.2.

Isolation

Isolation states that any concurrent transaction will render the same database state as if the transactions were performed sequentially. Any concurrent transactions is solved by our 2-phase commit and logging mechanism. The logging mechanism of first uncommitted and then

committed transactions creates a serialization of operations. Therefore, only transactions who have been committed into the logs will be written to the database.

System Topology and Nodes

2.1 System topology

Refer to Figure 1 on page 5.

2.2 Server Node

There is a single server running independently in our system. For each store that connects, it will register themselves with the server and the server will keep track of their addresses in a map. This map is provided to each store as a reply after they register in order for the stores in the network to establish a bidirectional connection with each other. The store registration process to the server is broken down into two phases. The first phase provides the connecting store with the leader store address in which the logs will be obtained from. The connecting store makes a connection with the leader store and a copy of the leader's logs will be given to the connecting store. These logs are used to update the stores database to be consistent with the network. In the second phase of registration, the connecting store receives a map of all other stores in the network to which it can then establish a bidirectional connections to each store. When a client connects to the network it will first connect to the server and the server will try to provide the most up-to-date map of stores it has to the client. The client will perform read and writes by referring to this map of stores for whom to make a connection to. The store network tries its best to keep the server up-to-date with any disconnections or leadership changes by relaying that information whenever it occurs of RPC.

RegisterStoreFirstPhase RPC

Arguments:

StoreAddress address of the store that wants to register

Results:

LeaderInfo information of the leader it should connect to

Receiver Implementation:

1. If the store array is empty, the new store registering will become the new leader.
2. LeaderInfo will be the new store's own information.

RegisterStoreSecondPhase RPC

Arguments:

StoreAddress address of the store that wants to register

Results:

ListOfStores information of all other stores

Receiver Implementation:

1. Server will update its own store array with the information of the store registering.
2. Returns its own store array to the store to connect with other stores.

2.3 Store Node

The bulk of our system relies on the store nodes in the network to perform. The store nodes are responsible for storing and maintaining an in-memory hash table that serves as our key-value database. Whenever a transaction (read or write) is performed by the client, the store must provide the correct information to the client. To reduce multiple points of entry and multiple points of inconsistency, all but one of the transactions must be performed out of a single store node known as the leader. The leader store is initially selected as the first store to join the network. This store remains the leader and will be the single source of truth for the entire store network meaning all logs and hash tables must be consistent with this leader across the network. The other stores are referred to as follower stores. Any read or writes that the client performs will return the value based on the leader's hashtable or writes to the leader's logs and hashtable first respectively. The leader is responsible for keeping a consistent state across the network and this is accomplished by our logging mechanism for writes (Section 4).

The leader store maintains its "leadership" by sending a heartbeat every 2 seconds to the follower stores. Each follower store checks every 3 seconds (extra second to counteract latency in the network) for a heartbeat. At any given time, if a leader store

can disconnect or one of the follower stores does not receive a heartbeat within 3 seconds then a new leader is selected from the network of followers. A follower is deemed the new leader based on a majority voting system described in Section 3. Again, like the previous leader, the new leader is now the single source of truth and all logs and entries in the hash table has to be consistent with this new leader.

2.4 Client Node

The client nodes emulates the transactions a user of the database would perform. The client will initially connect to the server to receive a map of the most up-to-date stores in the network. Whenever the client performs a read or write, it will pick a random store to initial request to. If that store is not the leader, an error message will appear on the client's console and the client is force to send the request again to the leader store. The client is exposed to an API that includes functionalities such as fast read, default read, consistent read, write and refresh. Described below are the corresponding RPC calls for the calls the clients are allowed to perform.

FastRead RPC

Arguments:

Key key where the value is read from

Value reply back to the client with corresponding value

Results:

Returns the value corresponding to the key

Receiver Implementation:

1. Returns DisconnectedError if store is disconnected
2. Returns KeyDoesNotExistError if key was not previously written to

DefaultRead RPC

Arguments:

Key key where the value is read from

Value reply back to the client with corresponding value

Results:

If store is leader, returns the value corresponding to the key, otherwise returns NonLeaderReadError

Receiver Implementation:

1. Returns DisconnectedError if store is disconnected
2. Returns KeyDoesNotExistError if key was not previously written to
3. Returns NonLeaderReadError if store is not leader, contains leader address in error message

ConsistentRead RPC

Arguments:

Key key where the value is read from

Value reply back to the client with corresponding value

Results:

If store is leader, sends RPC to rest of network to obtain majority answer and returns the value corresponding to the key, otherwise returns NonLeaderReadError

Receiver Implementation:

1. Returns DisconnectedError if store is disconnected
2. Returns KeyDoesNotExistError if key was not previously written to
3. Returns NonLeaderReadError if store is not leader, contains leader address in error message

Write RPC

Arguments:

WriteRequest contains the key and value to be written

Results:

If leader, undergoes 2-phase commit logging and writes the key and value into hashtable, otherwise returns NonLeaderWriteError

Receiver Implementation:

1. Returns DisconnectedError if store is disconnected
2. Returns NonLeaderWriteError if store is not leader, contains leader address in error message

RetrieveStores RPC

Arguments:

StoreInfo array of stores connected to server

Results:

Returns an array of the stores currently connected to server

Receiver Implementation:

1. Returns an array of stores currently connected to server

Leader Election

3.1 Election Process

The system can be prone to errors and crashes of nodes, but the most important node is the store node that is appointed leadership role. The store node with the leadership role takes on sole responsibility of all write operation therefore if it is down, the system must deal with how to appoint a new one.

3.2 Leadership Nomination

The first store to connect to the server will be designated as the leader. This store will remain the leader and will receive all write requests until its connection is dropped (See figure 2 on page 5). The leader store have an heartbeat that pings all other stores in the network indicating that it is alive. If at any point a store in the network does not receive the heartbeat from the leader, it will request a re-election for a new leader. Each store in the network will increment current term by one, and send RequestVote RPC to other stores asking for a vote based on 1. number of committed entries they have or 2. the length of log entries they have. If the store that receives RequestVote has equal number of committed log entries, then the store compares the length of the entire log. If the store that receives request has less than or equal number of log entries than the store that sends it, then the recipient store replies with one vote. Otherwise the store replies with zero vote. Each store can only vote once for a given term to ensure that there is at most one leader in a term. If the recipient's store's term is greater than the term of the requesting store's, then recipient store replies with zero vote. Also, each store will have a randomized election timeout (150-300 ms) period, which will make split votes a rare case. If no leader is chosen for a term, then the candidate store increments term and starts the election process again. Once any of the stores collect majority of votes based on currently connected stores, it designates itself as a leader and starts sending out heartbeats to rest of the network. Other stores that receive new leader store's heartbeat stops its election process.

Once a new leader is selected, it has to ensure all stores in the network has the same log and hashtable information as itself. We will compare the leader's log file with each other store's log file, find the latest common entry in the same index, and sync up from that point forward. Any newly committed requests we add to the store's log will be written to the hashtable. This is our rollback mechanism. All of write, defaultRead and

RequestVote RPC

Arguments:

CandidateInfo Information of the candidate who is requesting vote including currentTerm, length of log, and number of logs committed

Results:

Returns zero or one vote

Receiver Implementation:

1. Candidate gets a vote if 1. requester's length of log is greater or equal than recipient's or 2. requester's number of committed logs is greater or equal to recipient's.

consistent read requests from clients during election process is paused, and is resumed once a leader is elected.

Consistency

4.1 Keeping Consistency

A key component of database maintenance for our system is log replication. This is where consistency is achieved throughout the system. Leader node is responsible for propagating the most up to date log status to other store nodes. Please see section below for log replication algorithm.

4.2 Log Replication:

To keep consistency between our network of stores, we designate the leader store to handle all WRITE operations. For each WRITE operation the client will indicate which key and what value it wants to write to. Once leader receives write operation, the following steps occur:

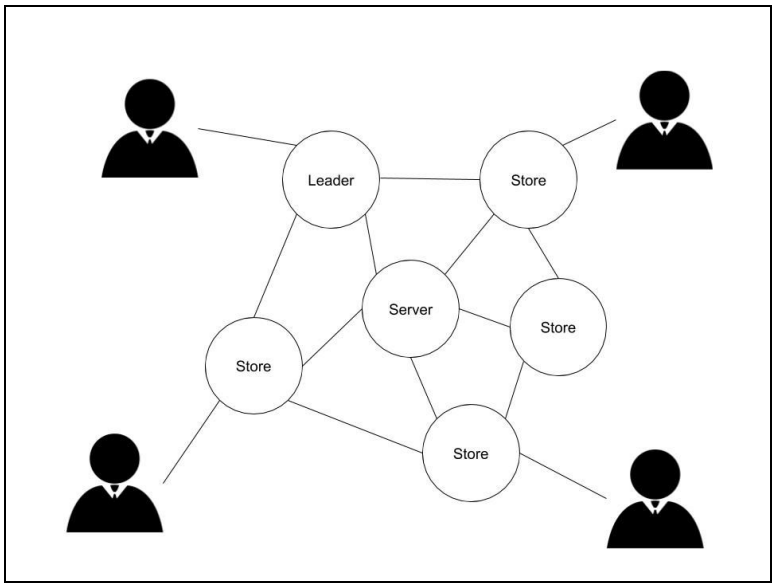

Figure 1. Application topology. A single server serving a network of stores bidirectionally connected to each other. Client nodes make connections to the stores to perform transactions

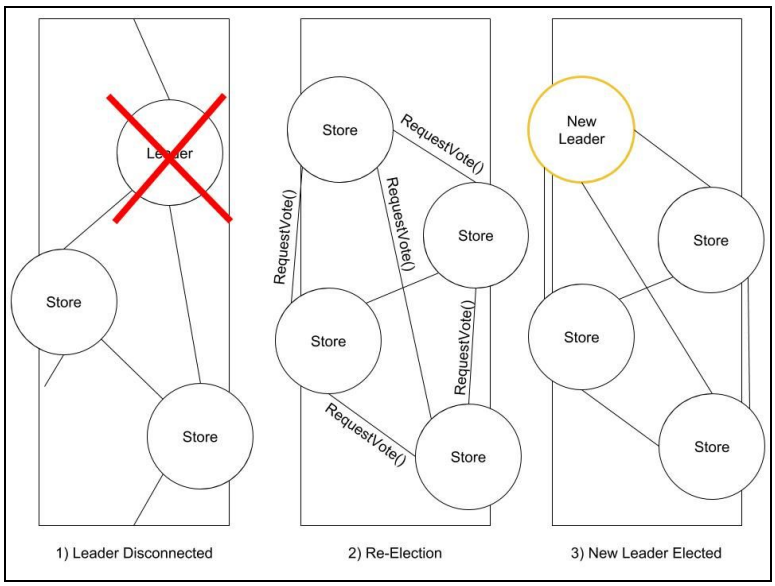

Figure 2. Steps in a leader nomination. The first step shows the disconnection of the leader node in the network. The second is the action that other store nodes take after the leader has disconnected. They will call RequestVote() on all the other store nodes to start the re-election. The last step reveals that one store node has retrieved the majority of the votes and will now start a new term as the new leader.

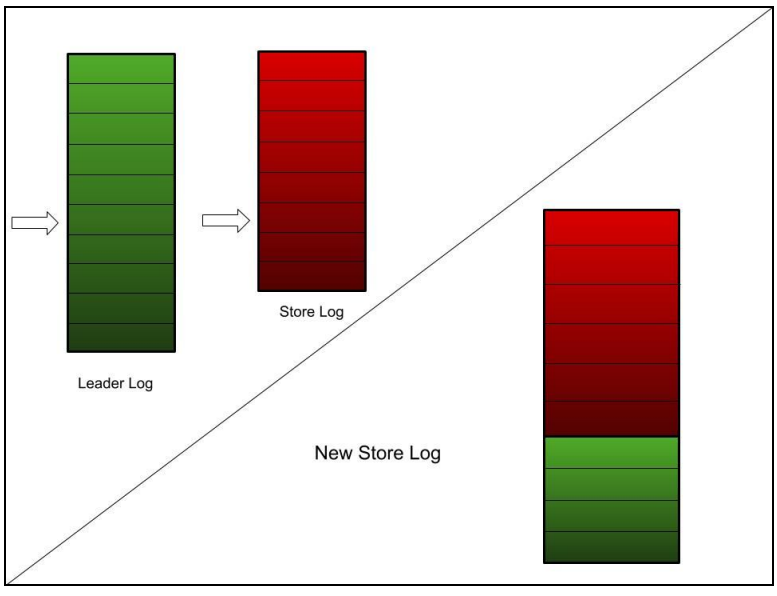

Figure 3. Rollback mechanism for synchronizing store logs with leader logs once leader nomination process ends and a new leader is selected.

1. The leader makes a new uncommitted transaction log entry in its logs with the following fields: {Current term, Index, Key, Value, IsCommitted}.
2. Leader sends WriteLog RPC to all other stores in the network to log this request in its logs as uncommitted and waits for acknowledgement. Recipient store will not acknowledge if the log request's term is smaller than its current term, or if the preceding log entry of the incoming log entry does not match with the last log entry of the recipient's store. Otherwise other stores acknowledge.
3. Once leader receives a majority of ACKs, it creates a new log for the request as committed and updates the hashtable with new key-value pair. Leader sends another RPC to all other stores with this new log entry and waits for acknowledgement. Recipient stores will again ack only if the term of incoming log entry is no less than the current term, and if the preceding entry is same as the last entry in its logs.
4. Once leader receives a majority of ACKS, then write operation is finished.

WriteLog RPC

Arguments:

LogEntries New log entry (uncommitted) and a preceding log entry

Results:

Acknowledgement of logging

Receiver Implementation:

1. If term of incoming log entry is less than recipient's current term, set acknowledgement reply to false.
2. If recipient's last log entry is not the same as the preceding entry of the incoming entry, set acknowledgement reply to false.
3. Otherwise, set ack to true and add the log entry into its logs

UpdateDictionary RPC

Arguments:

LogEntries New log entry(committed) and a preceding log entry

Results:

Ack Acknowledgement of logging and updating of dictionary

Receiver Implementation:

1. If term of incoming log entry is less than recipient's current term, set ack reply to false.
2. If recipient's last log entry is not the same as the preceding entry of the incoming entry, set ack reply to false.
3. Otherwise, set ack to true, add the log entry into its logs and update the dictionary with new key-value pair

If the leaders drops before the request was committed, then it will be lost when a new leader is selected. If at least one other store commits the request, it will be selected as the new leader and with our rollback mechanism, all other stores will receive the committed request.

Configuration and Cluster Changes

5.1 Cluster Membership Changes

When store nodes fail, outside of the leader node failing or when a new store node connects to the system somehow the connected components in the system must be aware of these changes. This can be referred to as configuration changes in the system.

5.2 Configuration Changes

If a new store connects to the server and there is already a leader, the new store will have to register with the client through 2 phases. The first phase includes a call to the server in which the server will reply with the address of the leader. The store will then store this leader information (making a connection, ip address) and create a bidirectional RPC connection with the leader. The store will ask the leader to obtain their hash table along with the log file in order to make the new store consistent and up to date. After the first phase, it will move on to the second phase, where it will call the server once more to notify it has all the updated logs and hash table. The server will then add this store into its store map and reply with a map of all the other stores so that the new store can create a bidirectional RPC connection with them. After these

two phases are done, the store will act just like an existing store in the network.

If a store disconnects from the network and a client just so happens to call it for a READ or WRITE request, we would receive a disconnect error prompting the client to automatically retry to a different store remove that disconnected store on its local map. Same would happen when a store makes an RPC call to a disconnected store, it will receive an error and remove that disconnected store on its local map. The store will then make an RPC call to the server to update it with the disconnection of a store node in its network. Refer to RegisterStoreFirstPhase RPC and RegisterStoreSecondPhase RPC in section 2.2.

DisconnectStore RPC

Arguments:

StoreAddress address of the store that disconnected

Results:

Ack acknowledgement that server received this information

Receiver Implementation:

1. Updates server's store array and adjusts the index of the array.

Conclusion

7.1 Conclusion

In conclusion, our project implements the Raft Consensus Algorithm that is described in the article. The main principles of this project is to encompass the ACI properties of database transactions and ensure that write and read operations satisfies these properties.

- **Atomicity**: Through the use of two-phase commits, we have established that the system will either be all-in (when it receives a majority of ACKs from other nodes) or nothing.
- **Consistency**: Each write operations are disseminated by the leader node throughout the network to ensure consistency with each database.
- **Isolation**: Using two-phase commit and logging, we can ensure that each operation will happen as though it were requested sequentially.

Our implementation tries to solve 3 main important parts of the Raft Consensus Algorithm, which are:

- **Log Replication**: This is done through the use of a leader node (receives all write operations) and then disseminates its new log throughout the network.
- **Leader Nomination**: A candidate store node must have the majority of votes from the network in order to become the new leader. This happens when the old leader has disconnected and a new term begins.
- **Configuration Changes**: Our project handles new incoming and disconnecting store nodes by using the heartbeat from the leader. The leader will then acknowledge this event and tells everyone in the network (including the server) of a node failure.

By implementing these major problems and properties written in the Raft article, we end up with a key value database store which allows users to manage replicated logs efficiently and securely.

References

8.1 References

Ongaro, Diego, and John Ousterhout. "In Search of a Understandable Consensus Algorithm." *Raft.github.io*, raft.github.io/raft.pdf.