Regina and the Plastics Distribute Key-Values (Plastic DB)

proj2_e9t9a_l8m0b_r9t9a_s1c9
Seerat Sekhon, Shavon Zhang, Winnie Wong, Brendon Wong

Introduction

A distributed key-value store is a key-value store hosted in multiple instances or separate datastores. The purpose is to prevent a single point of failure, be able to have high availability, and have eventual consistency.

Overview

What problem can we solve?

- Consistency: the same data will be returned after no updates occur for a period of time
- Availability: fault-tolerant, able to withstand failures

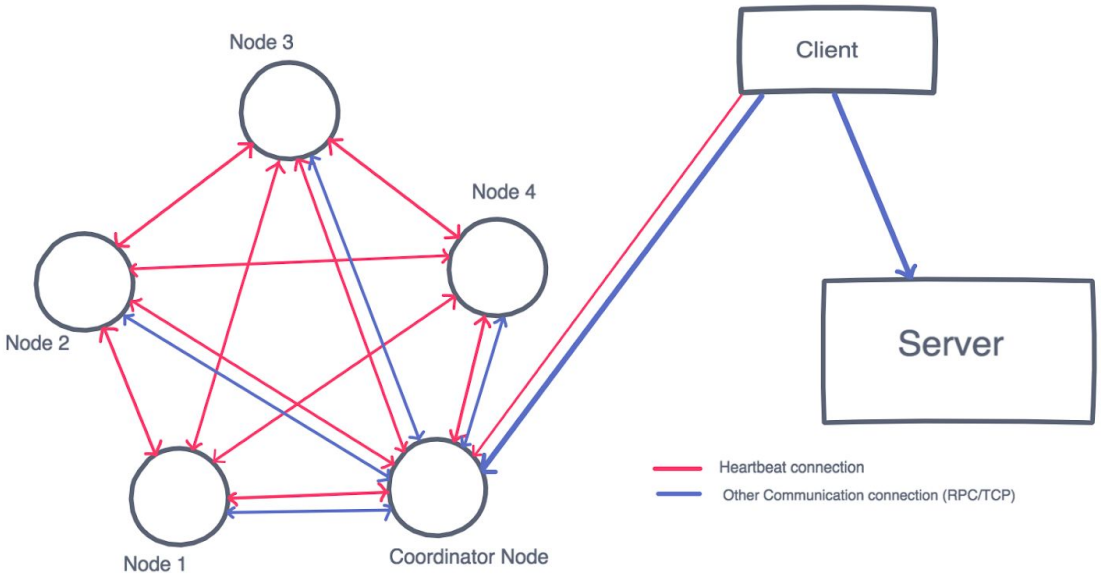
Assumptions & Constraints

- The server is always online, does not fail, and does not misbehave.
- The server is assumed to be correct and fault free
- A timeout of TIMEOUT seconds can be used to detect failed nodes.
- Assume client and nodes know the address of the server
- Assume that up to 10 nodes/instances can be connected at a time
- Assume trusted domain ie: everybody trusts everybody else
- We will have the restriction such that key-value pairs must be alphanumeric strings

Design

Architecture

There are three types of nodes in our system: up to 10 network nodes (one of which is the coordinator node), some number of client nodes, and a single server node. Each network node is aware of every other existing network node through a heartbeat connection (all-to-all, fully connected network). This allows us to detect and handle node failures quickly. The client connects to the server to retrieve information about the set of nodes in the system and subsequently connects to the coordinator node. The coordinator node is responsible for retrieving data from and updating data in all of the individual nodes as a result of the client requesting read or write access to the database. The diagram below showcases the connections in a failure-free system:



Server. The server is responsible for the registration of new network nodes and storage of all the addresses of the existing nodes as well as their online/offline status. Nodes can connect and disconnect from the server, but the server needs to continue to maintain communication with the elected Coordinator node (*See Coordinator-Node*) throughout the lifespan of the network. Through the Coordinator node, the server is periodically updated with the statuses of all of the nodes. This allows the server to return valid information about the existing nodes upon registration of another node.

Coordinator-Node. This node is the master-node and handles requests by clients, among other tasks. The remaining nodes act as replicants, each also holding the complete set of key-value pairings (*See Network-Nodes*). A node will be appointed the Coordinator role in the case that it is the first node to connect to the server, or if the previous coordinator fails.

The coordinator is responsible for retrieving or updating rows of the table on the remaining online Network-Nodes when it receives a read or write request respectively from a client.

As mentioned above, the coordinator must maintain a connection with the server at all times. See *Dealing with Failures* on ways to resolve a failure to do so.

Network-Nodes. The first node to connect to the server will automatically be appointed coordinator. However, it should be noted that all other network-nodes are capable of taking on the coordinator role, and may be elected to do so if the current coordinator fails (see *Dealing with Failures*).

Nodes will know the the address of the server prior to joining. Using this information, they will establish a connection with the server to register themselves. Upon successful registration, the server will return the address of the coordinator, as well as the addresses of all the other nodes in the network. The coordinator will assign each new node a unique ID. Each newly-joined node will have an ID higher than all the nodes that are alive in the network.

This newly-joined node will then proceed to establish connections with all of the nodes in the network, sending periodic heartbeats to maintain connection. Nodes who are not the coordinator can decide to disconnect from the server after registration.

Clients. Clients will know the address of the server prior to connecting to the network. They will first communicate with the server to get the address of the coordinator. After establishing a connection with the coordinator, a client will periodically send heartbeats to it to maintain its connection. A client is then able to read/write/update key-value pairings through the coordinator. If the connection between the client and the coordinator fails at any time, the client will be required to communicate with the server again, to get the address of the coordinator. This is necessary because a new coordinator may have been elected during this client's blackout period.

Replication & Conflict Resolution

We will be performing primary-backup replication. All nodes will contain a copy of each of the key-values. Due to transmission errors, delays, and failures, it is possible that some nodes will have inconsistent data. In such cases, we will resolve conflict through peer voting.

Peer Election/Voting

Coordinator Election: To elect a coordinator after a previous coordinator node's failure is noticed, each node will tell the server the node of which it knows to have the lowest ID. The server will tally the votes received within the voting period and determine the new leader by majority vote. If a tie ensues, then the Server will choose from one of the ties randomly. The server will let the Network know who is the new Coordinator. The new Coordinator will reach out to the Server and maintain a connection.

Node Failure Voting: *See Dealing With Failure: Node fails.*

CRUD Voting: *See Coordinator-Client API: Write/Update, Read*

Dealing with Failures

Network-Node Failure

Case: NodeB cannot connect to NodeA but Network can.

NodeA will only be deemed dead if the coordinator receives reports from a *quorum* number of nodes regarding the disconnectivity of NodeA.

Case: Node fails.
A neighbouring NodeB detects a failure in NodeA and reports the failure to the Coordinator. The Coordinator will set a timer of *voting* seconds to listen for other nodes reporting failure of nodeA. If *quorum* number of responses is reached within the voting period, NodeA is considered dead. The death of NodeA is broadcasted to the Network and each node in the Network will break the connection between itself and NodeA and update its list of available nodes. The Coordinator also updates server on the death of NodeA. If quorum is not reached within the voting period, voting ends and regular activity continues (NodeA is considered still alive).

Case: Node fails and comes back online.
Total failure of a node will be acknowledged by its neighbouring nodes through the lack of heartbeats received. Once a neighbouring node detects failure, it will forward this information to the coordinator. If this node was to come back online, it would have to re-register itself with the server, and join the network as a new node.

Coordinator Node Failure

Case: Network can connect to Coordinator but Coordinator cannot connect to Server.
In the case that the coordinator is unable to connect to the server, but can still maintain a connection to the rest of the network, it will attempt to re-establish a connection with the server *numRetry* number of times. If this connection continues to fail, the coordinator will decide to step down from it's role. It sends out a broadcast to all network nodes to begin vote for a new coordinator.

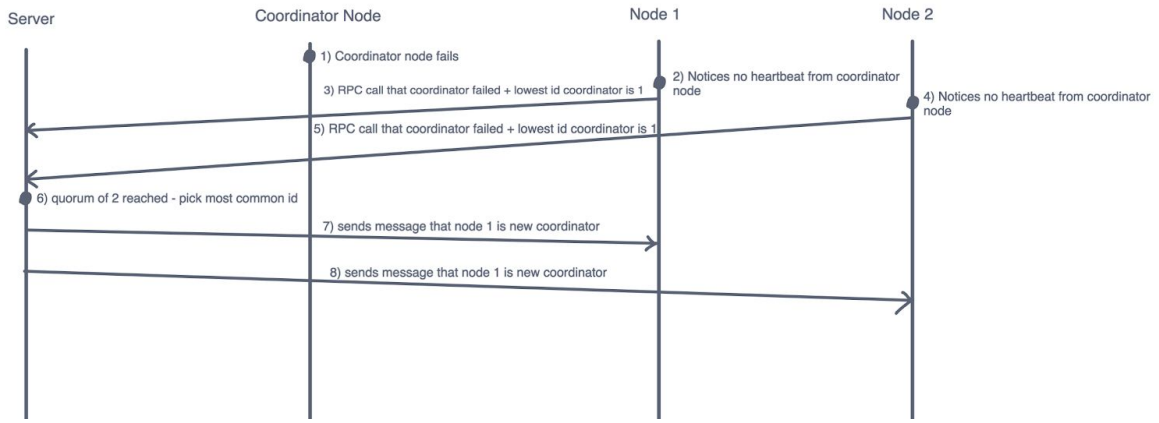
Case: Network cannot connect to Coordinator.
After detecting the failure of the coordinator, the nodes of the network will all individually establish a connection with the server to issue a vote as to who they believe the new coordinator should be. The server must then tally up the votes, and broadcast the coordinator who received the most votes to all of the nodes in the network. This is done through TCP to ensure receipt of the message on the network side. This also allows for consistent knowledge throughout the system. If there is a tie of votes, the server will choose one among the ties at random.

Case: Node fails, then Coordinator fails.
In such case, we should handle the coordinator failure first. This means, the network should vote, and appoint a new coordinator before dealing with the node failure. By doing this, we can increase the uptime and increase availability.

Case: Coordinator fails, Client cannot make a call. Coordinator Election is underway.
The Client probes the status of the Coordinator through heartbeat calls. When the Client realizes the lack of heartbeats, it will disconnect from the Coordinator, query the Server for the

address of the Coordinator, as the Coordinator may have failed. The address returned by the Server may be the same as the past Coordinator or a new address. If a Coordinator election is underway and there is temporarily no Coordinator, the call will block until a Coordinator has been elected and the Coordinator is connected to the Server. When the Client receives an address response, it will connect to the given Coordinator and make its calls as before.

The diagram below illustrates this process. If the client makes a request anytime between steps 1 and 6, the server will have no coordinator id to respond with and will block until the server finishes step 6 (picks the new coordinator).



Client Node Failure

If a Client fails at any point in the system, when it rejoins, it must contact the server and request the address of the Coordinator. Any non-idempotent operation called prior to failure will be persistent in the key-value store, assuming the operation is successful.

APIs

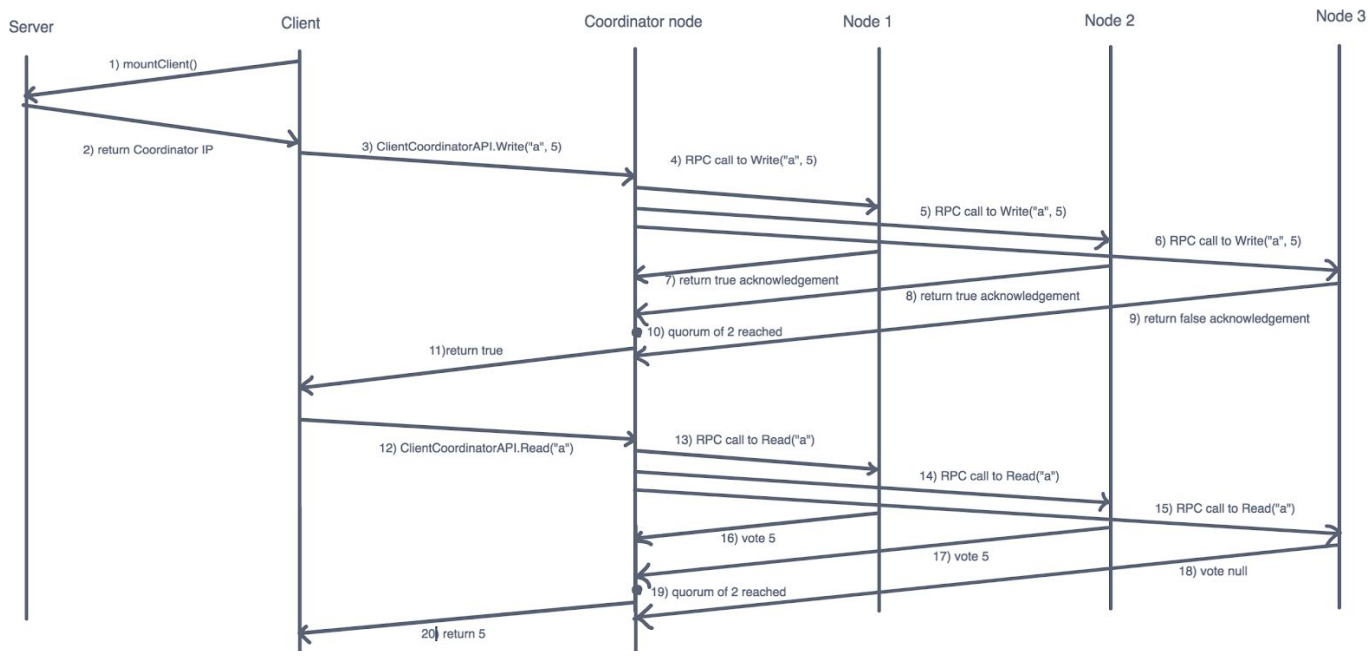
Server-Node

1. **err, coordinatorIPAddr ← register(nodeAddr)**
   To join the network, a node will first have to register itself with the server. After the server has registered this node, it will return the address of the coordinator node to the node to connect to. It will also return the addresses of the nodes currently in the network.

2. **voteNewCoordinator(nodeID)**
   If a node is no longer receiving heartbeats from the coordinator, it will send a message containing it's vote on who it believes should be the new coordinator (node with the lowest ID) to the server. When the server first receives a vote, the voting begins. The server waits until *quorum* number of messages containing the same ID is received from other nodes in the network or until voting period ends. If *quorum* number of messages

were received then server will broadcast this ID to all of the nodes in the network as the new coordinator.

Coordinator-Client API



1. **err ← Write/Update(key, value)**
   When coordinator receives a write request with its corresponding key and value, it will validate the request (confirm that key-value strings are alphanumeric) and broadcast the write request to all the other nodes. This broadcast is shown in the diagram above, where once the coordinator node receives a write request from the client, it broadcasts the request to all nodes (3 nodes in this case). Once it receives a successful response from a *quorum* number of nodes (2 in our case), the coordinator will return acknowledgement of this operation to the client.

   If the coordinator does not receive responses from a *quorum* number of nodes within a certain *responseTime*, it will return a failure to the client.

2. **value ← Read(key)**
   When the coordinator receives a read request of a given key, the coordinator will query all the nodes in the network. This is shown in the diagram above where the coordinator makes 3 RPC calls to the 3 nodes in the network upon a read request. When a *quorum* number of responses for the same value is returned from the nodes, that value is returned back to the client.

In the case that there is a tie on multiple values, a random value among the ties will be selected as the correct one. The correct value will replace the other values in the replicas, and will be returned to the client.

3. **err ← Delete(key)**
   When the coordinator receives a delete request with a given key, the coordinator will forward this request to the given nodes. An acknowledgement will be returned back to the client once a *quorum* response is received from the node. If this does not occur before *timeout*, the coordinator will return failure.

Node-Node

1. **err ← heartBeat(id)**
   Nodes must send out heartbeats with their ID to all other nodes of the network every *heartbeat* seconds. This node will be assumed dead if *quorum* number of nodes do not receive heart beats from the node. If the disconnected node is not the coordinator, report of this disconnected node is forwarded to the coordinator to handle. Otherwise it is reported to the server.

Azure Use

The Azure cloud consists of several data-centers world-wide. We will use the Azure environment to deploy our server and network nodes. The clients of our system may be deployed from anywhere. This allows us to test our system on a large scale.

Testing

We shall add unit tests for core functionality of the project (such as API capabilities, including reading, writing, deleting). These tests shall include error cases such as writing of invalid keys or values, reading of non-existent values, and deleting non-existent values as well as correct cases.

Testing scripts and manual analysis of behaviour shall be employed for a large portion of the project. We shall be observing single node, two nodes, and N node cases. For single node cases, we shall test basic behaviours such as writing a value and then attempting to read the value. For two and multi-node cases, we shall test cases such as: the failure of a coordinator and the election of new coordinators, whether reads, writes, and deletions are observable after multiple nodes attempt actions on the same value, and redistribution of keys upon node failures.

Stress tests shall also be performed manually by invoking large clusters of node and clients to observe any degradation in performance and to see if any functionality is compromised.

Additional testing may be done with GoVector and ShiViz to observe and validate the communication structure of our system.

Timeline

* Dates in red are key project dates

Date	Event
March 16	Set up server code - Shavon Allow nodes to register and set first node as coordinator node - Winnie Allow clients to connect to Server and Coordinator node - Seerat Send heartbeats between nodes - Brendon Send heartbeats from client to coordinator node - Brendon Draft API between client and coordinator node - Winnie Draft Replication protocol (Node additions and failures) - Seerat
March 22	Add Consistent Hashing protocol - Seerat Develop Client-Coordinator API (Writes, Reads, Consistent Hashing) - Winnie, Shavon Add tests for client READ/WRITE requests - Shavon Add replication protocol - Brendon, Seerat
March 23	Group meetings with designated TA
March 30	Add tests for node failure detection - Seerat Add Network-Node Failure protocol - Shavon Add Coordinator Node Failure protocol (Elections) - Winnie, Brendon
April 5	Complete any unfinished tasks from previous weeks - all members Write the report - all members Test the system thoroughly - all members Ensure that all points in the proposal are met - all members
April 6	Code and final reports due

SWOT Analysis

Internal

Strengths

- Team members are all familiar with each other and have worked together in Project1
- Team members are highly available for communication, we respond to team messages within the hour

- Team members have a diverse background (internships, engineering clubs experience) with various tools and technologies
- Each team member has familiarity with coding in Golang through previous assignments as well as side-projects

Weaknesses
- Writing tests in Golang - no one is familiar with this as we didn't do it in any assignments or Project 1
- Team has had issues meeting deadlines in Project 1
- Team has had communication issues, this can be solved by meeting in person/on skype more often instead of exchanging information over text-based methods
- Being too ambitious. With the limited time we should be realistic on what our Minimal Viable Product is.

External

Opportunities
- Many tutorials and resources are available for creating boilerplate KV stores
- Libraries exist for election protocols which can help us figure out our implementation

Threats
- We are concerned that the gossip protocol may be a better than our current plan to have all-to-all connections for nodes. If all-to-all doesn't work, it could set us back.
- Assignments, exams and club commitments can hinder the completion of this system
- Abundance of resources may cause confusions in design and implementation
- Theory of implementation may not equate to real-life implementation, resulting in unforeseen problems and blockers
- Golang is a relatively new language, changes in the language may cause deprecation issues for our system

References

1. Castro-Castilla, A. (April, 2015). Building a Distributed Fault-Tolerant Key-Value Store. http://blog.fourthbit.com/2015/04/12/building-a-distributed-fault-tolerant-key-value-store accessed Mar 3, 2018.
2. Halterman, J (Jan, 2017). How to Create a Distributed DataStore in 10 Minutes. http://jodah.net/create-a-distributed-datastore-in-10-minutes accessed Mar 1, 2018.
3. Cassandra Architecture. Apache Cassandra Documentation. http://cassandra.apache.org/ accessed Mar 1, 2018.
4. DeCandia, G., et al. Dynamo: Amazon's Highly Available Key-value Store. www.allthingsdistributed.com/2007/10/amazons_dynamo.html accessed Mar 1, 2018.