# 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 pure and true distributed system (or network), as it is simply a key-value database hosted across multiple independent hosts (or nodes) such that each host contains a local key-value store. This project was chosen because of its clear distributed properties, which we believed to be a good challenge of our distributed systems knowledge. Our team wanted to experience how daunting it would be to convert a single centralized datastore to a distributed one. Because data on centralized stores is stored on a single host, if the host fails, all data will be lost. A distributed store, however, has decentralized copies of the data and must ensure that upon failure of a node, the system is able to recover without loss of data. With multiple nodes, we need to solve consistency and communication issues across the independent stores. With this in mind, our team aimed to develop a system that would be highly available and eventually consistent.

## Problem Description

As mentioned above, failure of a centralized key-value datastore results in data loss as there is no redundancy of data. If businesses used centralized key-value datastores to hold their transactions or other important data, a system failure would be catastrophic, as all information would be lost.

A distributed key-value store solves this issue by ensuring that there is no single point of failure that could destroy the system. If at any point there is a failure within our system, our information would still be intact, as we have multiple copies of the data. This allows for a safe and easy recovery of our system.

## Procedure and Design

### Design

Inspired by Cassandra and DynamoDB, we have replicant nodes and a coordinator node. Many of the stores we studied comprise of said nodes and clients. This made the project a larger undertaking, so we decided to add a server node to monitor network state. Doing so gives greater visibility into the system and eases some responsibilities from the coordinator node. The server also makes insertion of new nodes into the system easier.

We decided to have a primary store (that being the coordinator) which would house all the data, and have all other nodes function as primary replicas to the coordinator. This helps ensure high availability because if one node fails, we still have the data stored on the other nodes. Should the coordinator fail, any other node can become the new coordinator and primary store.

All nodes within the network establish RPC connections to all other nodes. This allows us to easily keep track of which nodes are alive in the system. The coordinator itself is the only network node that is continuously connected to the server. This allows us to update the server with the most current network state.

## 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 5 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

## Architecture and code overview

There exist 4 different entities in our system: the server, the client, the coordinator, and an optional set of network nodes.

The Server is used as the initial point of communication for nodes who wish to join the system and for clients who wish to access the key-value store. The server stores the addresses of all existing nodes in the system and keeps track of the current coordinator. When a client connects to the server, the server relays the coordinator's address so that the client can make requests. Similarly, when a node joins the system, it connects to the server to retrieve a list of active nodes in the system. The coordinator updates the server every *UpdateServerInterval* seconds with a list of currently active nodes. The server also handles the election of a new coordinator in the event that the current coordinator fails. This protocol is defined in *Functionality*.

The Coordinator is the master-node of our system. The Coordinator handles read, write, and delete requests from clients by propagating the requests to the network nodes and determining the quorum response. The Coordinator is also responsible for handling the decisions in the network-node failure protocol of our system. This protocol is defined in *Functionality*.

Network nodes are used as replicas for the key-value store. They receive requests from the Coordinator for reads, writes, and deletes. Network nodes maintain heartbeat connections to all other nodes in the system (including the Coordinator). In the event of failures, they will report the failure to the correct failure handling entity.

Clients which connect to our system perform write, read, and delete requests through the Coordinator to modify values in the key-value store. They receive the result of their request or a failure when the request could not be completed.

Upon starting, the server listens for incoming connections from nodes and clients at a known tcp port.
Server - Node API

- Settings, err ← RegisterNode(Address)
  - Registers a new node with a unique ID and address into the network. If this is the first node to join or if no coordinator in the network exists, this node is appointed Coordinator. The Coordinator periodically updates the server on existing nodes in the network.
- addrSet, err ← GetAllNodes()
  - Returns address of all existing nodes in the network. Once the caller node has retrieved all existing nodes, it is free to disconnect from the server.
- MonitorCoordinator()
  - When there is no Coordinator present and no Coordinator election is underway (ie: no alive nodes in network), the server will erase its record of alive nodes. The next node that joins the network will be appointed Coordinator.
- Err ← CoordinatorFailureReport(coordinatorNode, newCoordinator)
  - If a network node fails to receive a heartbeat from the coordinator within a given period, it sends a failure report for the coordinator and votes for a new coordinator to the server. A node will always vote for the node they believe has the lowest ID. For the first failure report, the server will open up a voting period for incoming failure reports from other existing network nodes. If a quorum number of failure reports have been received during this time window, an election begins. See *Coordinator Election* below.

Server-Client API

- Addr ← GetCoordinatorAddress()
  - For a client to join the network, it must first retrieve the address of the Coordinator from the server. It will then connect and communicate with the Coordinator.

DKV API

- CNodeConn, err ← OpenCoordinatorConn(addr)
    - Upon receiving the Coordinator address from the server, the client will begin communicating with the network using the dkvlib library.
- value, err ← Read(key)
    - If the client cannot connect to the coordinator, this call returns an error. Otherwise, this will forward the read request to the Coordinator. See *Coordinator-Node API.*
- Reply, err ← Write(key, value)
    - If the client cannot connect to the Coordinator, an error is returned. If the key does not consist of alphanumeric characters, InvalidKeyCharError or InvalidValueCharError is returned. Otherwise, the request is forwarded to the Coordinator. See *Coordinator-NodeAPI.*
- err ←Delete(key)
    - If the client cannot connect to the coordinator, an error is returned. Otherwise, it will forward the delete request to the coordinator. See *Coordinator-Node API.*

Coordinator - Node API

- value, err ← CoordinatorRead(key)
    - The Coordinator asks all network nodes for their value of the key. If quorum nodes agree on a value, that value is broadcasted to all nodes and returned and the coordinator sends a write request to all nodes for that majority value. Otherwise, the request is deemed unsuccessful and the coordinator requests nodes to delete the value from their local store.
- Reply, err ← CoordinatorWrite(key, value)
    - The Coordinator will send out a write request for this key-value pairing to all network nodes. It keeps track of the successes of each request. If quorum writes succeeded, success is returned, otherwise unsuccessful.
- Reply, err ← CoordinatorDelete(key)
    - The Coordinator sends a delete request for the given key to all network nodes. Keeping track of the successes of each request, success is returned if quorum deleted the key.
- Err ←NodeFailureReport(failedNode)
    - If a node fails to receive heartbeats from another node in a specific time frame (See *Node-Node API*), it will send a failure report to the Coordinator. If this is the first failure report for failedNode, the Coordinator begins the voting period to receive failure reports from other nodes. If quorum number of failure reports are

        received, NodeFailureAlert is sent out to all nodes, thus removing failedNode from the network.  Otherwise, failing reports are ignored.

Node-Node API

- Err ← SendHeartbeats()
    - To detect node failures, we periodically send heartbeats between all existing nodes.

Coordinator Election

If a quorum number of failure reports for the Coordinator are received, an election for a new Coordinator will occur. Each CoordinatorFailureReport comes with a vote for a potential new Coordinator.  The server keeps a tally of how many votes each candidate receives, electing the candidate that reaches a quorum number of votes. If a tie ever ensues, then the server chooses from one of the ties randomly. The server lets the network know who is the new Coordinator. The new Coordinator will reach out to the Server and maintain a connection.

Handling Failures

Case: Node A dies before Coordinator

When the remaining nodes attempt to report the failure of node A to the Coordinator, they find that the Coordinator is no longer connected.  In this case, an election process will begin for a new Coordinator.  The failure of node A will be handled once a new Coordinator has been appointed.

Case: Node A dies during Coordinator Election

When broadcasting election results, the server will first attempt to broadcast to the newly appointed Coordinator.  If this fails for any reason, the server will move on to elect the Candidate with the next highest number of votes.

Case: Network can connect to Coordinator but Coordinator cannot connect to Server

If the connection between the Coordinator and Server fails, the Coordinator will attempt to reconnect a given number of times.  If it fails to reconnect, it will inform the network that it is resigning from its role as Coordinator.  The network will then vote on a new Coordinator.

Case: After voting, only Node A knows it is the Coordinator.  Node A then dies before other Nodes have saved Node A as the coordinator.  Other nodes have no coordinator to send a failure report of Node A to.

When a network node receives broadcast of a new coordinator, it will first check that this node is valid before saving it as the new coordinator.  If this node is no longer valid, it will send an error back to the server.  The node will detect the failure of this newly elected coordinator through the lack of heartbeats.  Because the node have an appointed a Coordinator to send this failure to, it will send a coordinator failure to the server.

Case: Server cannot connect to network node to broadcast new coordinator
The server will continue to broadcast to rest of network nodes it it fails to broadcast to one network node.

## Verification and Regression Testing

We manually tested behaviour of each new feature and thoroughly performed regression testing after integrating each new feature with the existing functionality. The following detail the testing procedures we followed for each of the specified cases:

### Testing Read, Write, Deletes

To test failure-free read, write, and delete requests is simple. We have the client make the request and verify that the correct value is read, written, and deleted by examining the contents of each node and the result returned to the client. In attempts to read or delete a key that does not exist, we verify that we receive an indication that the value does not exist or an error occurred, respectively.

### Testing Read in Cases With Diverged Values

To test the case where nodes have diverged on their representation of certain keys, we populated each node's datastore with different values for the same key. We then ensured that the correct behaviour occurs in the cases where a majority value exists (the value is returned) and a majority value does not exist (key does not exist and is deleted from all nodes).

### Testing Node and Coordinator Failures

As defined above, there are many Node and Coordinator Failures that we needed to handle in our system. For each failure, we had to ensure that our system was able to recover and maintain a stable state. Testing this involved killing the aforementioned node and verifying whether the system was able to recover. Depending on the situation, this meant that after the system stabilized, the following conditions were met:

- There exists an elected coordinator
- The node that failed is removed from the system
- Nodes are still able to join the system
- Clients are still able to make successful read, write, delete requests
- All nodes are aware of all other nodes
- All nodes are aware of the identity of the coordinator node

### GoVector and Shiviz

Both GoVector and Shiviz were used as tools to confirm that expected events were happening in our system in the correct order. Although testing with print statements allows us to peek into the state of our system, they are sometimes hard to follow and prone to bugs. Having a diagram which clearly identifies interactions between nodes in the system and the order in which they occurred helped us confirm we were handling expected functionality and errors in the correct manner. Please refer to *GoVector and Shiviz* below for our diagrams of key work-flows of our system and their explanations.

## Azure

We use Azure to deploy our server and network nodes. The clients of our system may be deployed from anywhere that is able to reach our Azure machines. This allows us to test our system on a large scale.

## Evaluation/ Discussion

Our final product incorporates everything outlined in our proposal and accomplishes nearly all of the functionality that we wanted. Our system works just as we envisioned and is able to support up to 10 network nodes. As defined in *Functionality*, we are able to send requests to the network, elect new coordinators, handle node failures at any time, update the server periodically, and have an eventually consistent datastore. We were also able to create a client that can dynamically issue read, write, delete, and sleep operations through the command line. Through our testing, we were able to identify more failure cases which we previously had not thought of.

Aside from learning of new unexpected failure scenarios, one other major problem dealt with concurrency. Because most calls are sent through RPC, the time it took for callers to receive replies varied. In addition, the time to query votes and determine quorum during coordinator elections and database operations also varied with the size of the network and the network latency. Therefore, defining satisfactory periods (timeout, voting, election, sleep times, etc) that reduced false failures but also allowed for a usable system was somewhat of a challenge.

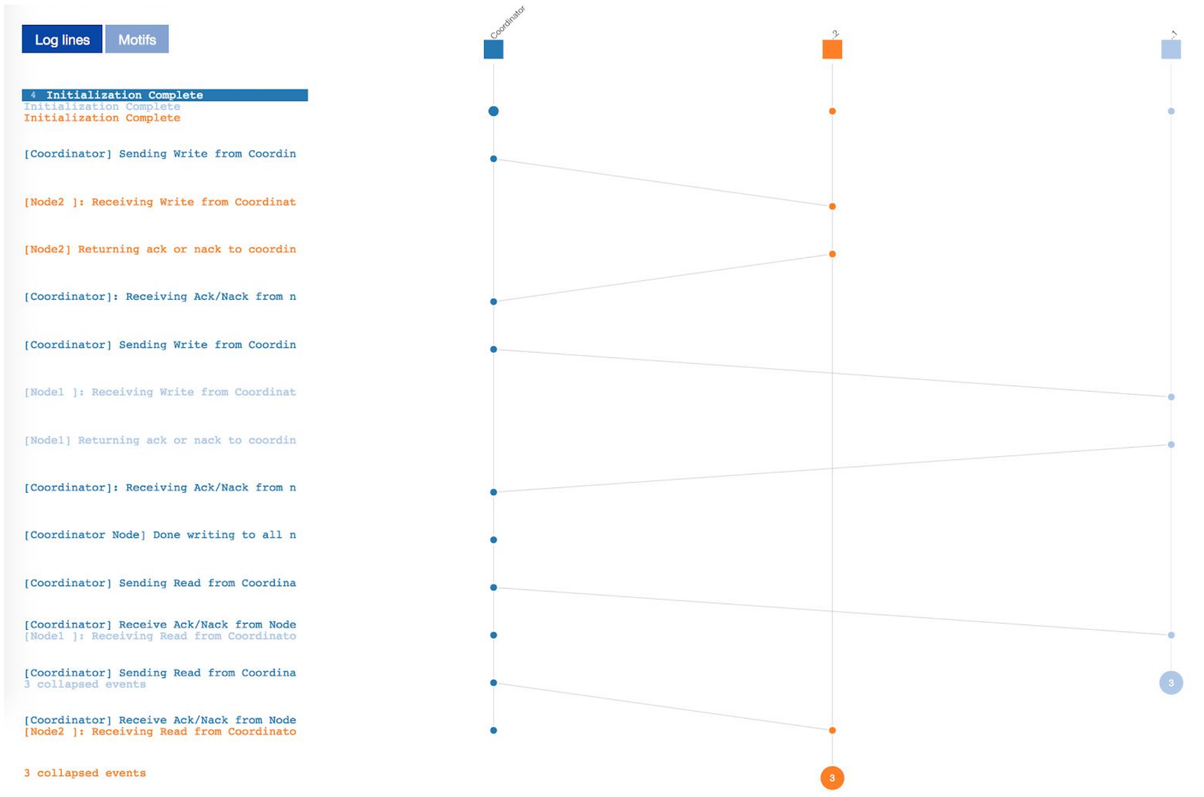
If we were to implement the distributed datastore differently, we may consider changing how the nodes communicate. Currently, we have a fully connected network of network and coordinator nodes. As a result, the nodes must maintain multiple connections to all other nodes at a time, which are subject to random failures and congestion, and is not very scalable.

## Conclusion

With the completion of this project, our team believes that we have created a distributed key-value store that is highly available and eventually consistent. Although there were some difficulties in relation to concurrency, the development of this project was finished within our schedule. As such, we believe that we have created a well structured, fault-tolerant system.

# GoVector and Shiviz

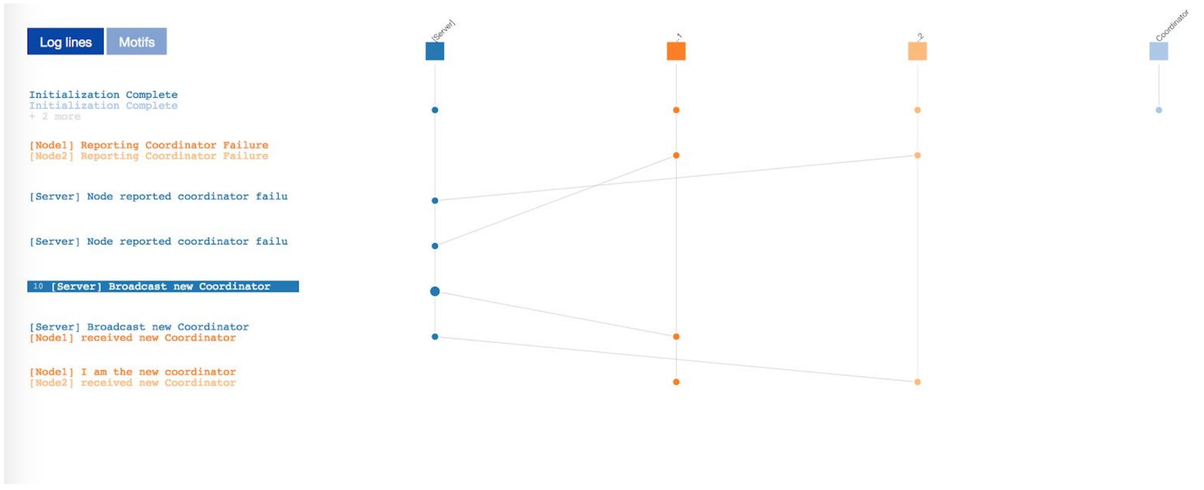
## Reads and Writes



```
4  Initialization Complete
Initialization Complete
Initialization Complete

[Coordinator] Sending Write from Coordin

[Node2 ]: Receiving Write from Coordinat

[Node2] Returning ack or nack to coordin

[Coordinator]: Receiving Ack/Nack from n

[Coordinator] Sending Write from Coordin

[Node1 ]: Receiving Write from Coordinat

[Node1] Returning ack or nack to coordin

[Coordinator]: Receiving Ack/Nack from n

[Coordinator Node] Done writing to all n

[Coordinator] Sending Read from Coordina

[Coordinator] Receive Ack/Nack from Node
[Node1 ]: Receiving Read from Coordinato

[Coordinator] Sending Read from Coordina
3 collapsed events

[Coordinator] Receive Ack/Nack from Node
[Node2 ]: Receiving Read from Coordinato

3 collapsed events
```

There are 3 nodes in this Shiviz diagram: the coordinator node and 2 network nodes. The Coordinator makes a write request to the first node and subsequently receives an acknowledgement. It then makes a call to write the same value to the second node and receives an acknowledgement as well. This process is repeated for read calls to the coordinator as well.

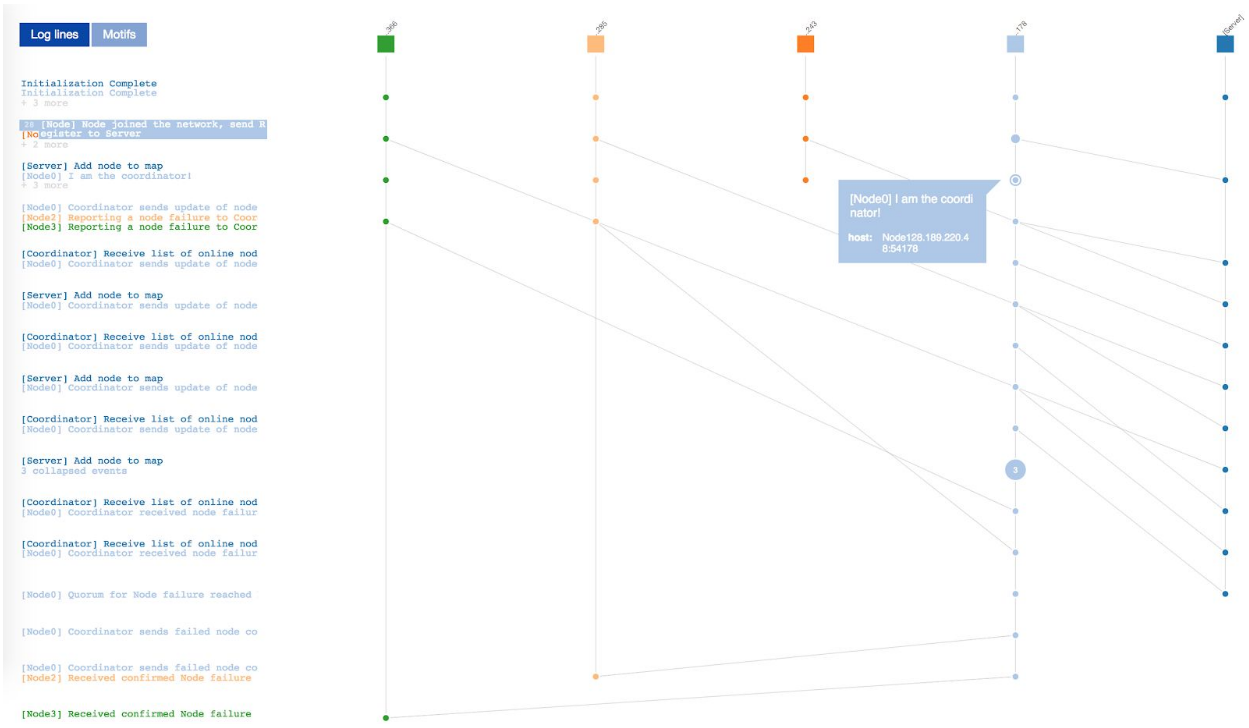
## Coordinator Failure



```
Initialization Complete
Initialization Complete
+ 2 more

[Node1] Reporting Coordinator Failure
[Node2] Reporting Coordinator Failure

[Server] Node reported coordinator failu

[Server] Node reported coordinator failu

10  [Server] Broadcast new Coordinator

[Server] Broadcast new Coordinator
[Node1] received new Coordinator

[Node1] I am the new coordinator
[Node2] received new Coordinator
```

The above diagram demonstrates the handling of a coordinator failure in our system. The coordinator (node on far right) fails and Node 1 and Node 2 notice the failure due to missing heartbeats and notify the server. The server has received quorum number of votes (2) and chooses the new coordinator based on these votes. The new coordinator is broadcast from the server to both of the remaining nodes in the system.

Node Failure + Server Updates



```
Initialization Complete
Initialization Complete
+ 3 more

28 [Node] Node joined the network, send R
[Noegister to Server
+ 2 more

[Server] Add node to map
[Node0] I am the coordinator!
+ 3 more

[Node0] Coordinator sends update of node
[Node2] Reporting a node failure to Coor
[Node3] Reporting a node failure to Coor

[Coordinator] Receive list of online nod
[Node0] Coordinator sends update of node

[Server] Add node to map
[Node0] Coordinator sends update of node

[Coordinator] Receive list of online nod
[Node0] Coordinator sends update of node

[Server] Add node to map
[Node0] Coordinator sends update of node

[Coordinator] Receive list of online nod
[Node0] Coordinator sends update of node

[Server] Add node to map
3 collapsed events

[Coordinator] Receive list of online nod
[Node0] Coordinator received node failur

[Coordinator] Receive list of online nod
[Node0] Coordinator received node failur

[Node0] Quorum for Node failure reached

[Node0] Coordinator sends failed node co

[Node0] Coordinator sends failed node co
[Node2] Received confirmed Node failure

[Node3] Received confirmed Node failure
```

```
[Node0] I am the coordinator!
host:  Node128.189.220.48:54178
```

Node Failure Handling is very similar to Coordinator failure handling except that the process is directed by the coordinator instead of the server. In the above diagram we have 4 nodes and one server. The 4th node is the initial coordinator. The above diagram shows that the 4th node connects and because it is the coordinator, it will periodically update the server with the list of nodes that are alive in the system. After initialization, node 3 (bright orange node) fails. Nodes 1 and 2 report the node failure to the coordinator. The quorum for the node failure is reached, the node is removed from the system, and the coordinator sends back confirmation to Node 1 and 2 that the node was removed so that they can also stop monitoring and sending heartbeats to it.