Arvind Kouta (kouta2)
Parker Drake (ptdrake2)

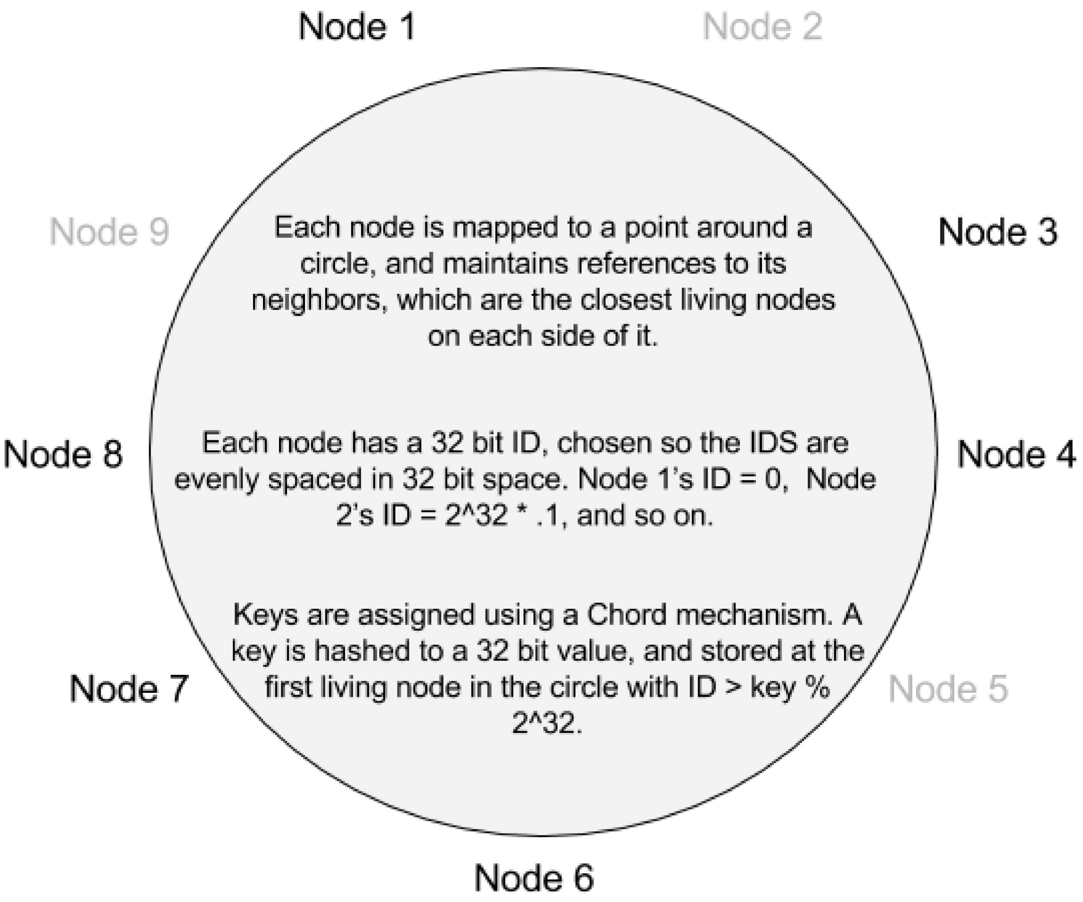
CS 425 MP2 Design Document

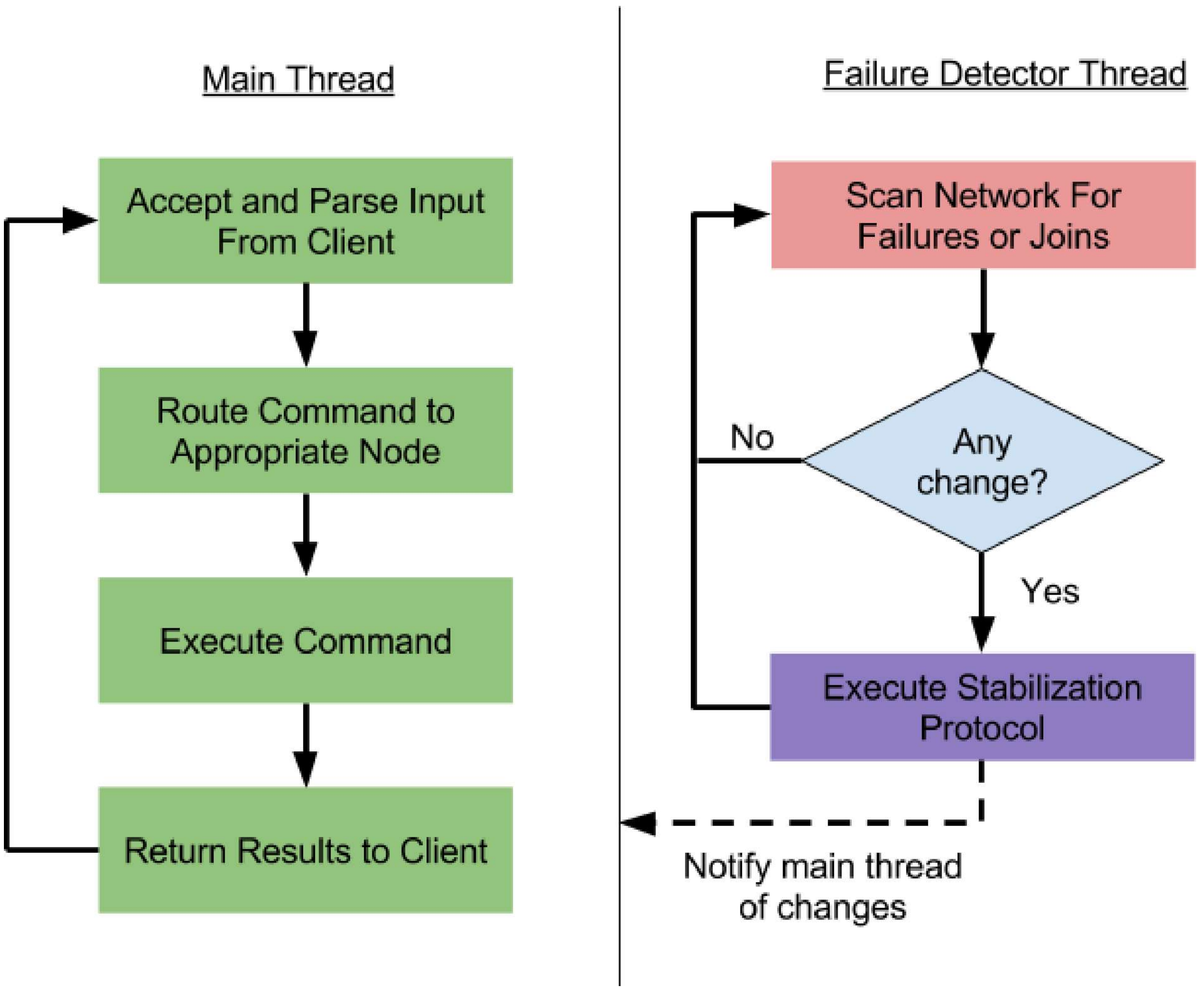
Introduction

For this MP, we implemented an in-memory, scalable, fault-tolerant, distributed key value store. The goal of this key/value store is to support several hash table operations, including *get* and *set*, on multiple computers in a distributed environment. This implementation makes use of remote function invocation, synchronization, failure detection, and replication to achieve these goals.

System Model and Control Flow

Below is an image demonstrating the system model we based our implementation from.


Node 1
Node 2
Node 9
Node 3
Each node is mapped to a point around a circle, and maintains references to its neighbors, which are the closest living nodes on each side of it.
Node 8
Node 4
Each node has a 32 bit ID, chosen so the IDS are evenly spaced in 32 bit space. Node 1's ID = 0, Node 2's ID = 2^32 * .1, and so on.
Node 7
Node 5
Keys are assigned using a Chord mechanism. A key is hashed to a 32 bit value, and stored at the first living node in the circle with ID > key % 2^32.
Node 6


Main Thread
Accept and Parse Input From Client
Route Command to Appropriate Node
Execute Command
Return Results to Client
Failure Detector Thread
Scan Network For Failures or Joins
Any change?
No
Yes
Execute Stabilization Protocol
Notify main thread of changes

Design

To make the code more modular and improve debugging, the code was split into several different classes.

The *main* class holds all the code to execute changes to the key value store on the local machine, handles all input to the process, and holds all the data structures necessary for the process to run. This includes a membership list of living nodes, the replicas of the local process, and the local key/value store itself.

The *Executor* class holds the functions necessary to hash and route a key to the process that owns that key.

The *Stabilizer* class holds the helper functions necessary to maintain the membership list, update a node's neighbors, and assist with the redistribution of keys when a node fails or joins the cluster.

The *Failure Detector* class holds all the code to do failure detection.

In several cases, synchronization techniques were used to prevent race conditions on critical data structures. These data structures included the key/value store at each node, the membership list, and the process identification numbers of that node's replicas. Synchronization was achieved through the use of mutex locks.

The data structure used to store the keys and values is a Java HashMap. The data structure supports constant time lookup and write operations, and so was a good choice. The membership list is maintained as a Java TreeMap. This structure is similar to a HashMap, with the additional property that the keys are maintained in sorted order. As such, it supports log time operations, but is a good choice for a membership list because it allows easy lookup of processes neighboring a process of interest.

For all communication between processes we use the Java RMI library. The abstraction makes remote function calls simple and errors are easy to handle.

Fault Tolerance, Failure Detection, and Stabilization

Since the specification allows only 2 failures before stabilization can take place, storing 3 replicas of each key is sufficient to prevent data loss. Each node stores the keys routed to itself, as well the keys that route to its left neighbor in the circle, and its right neighbor in the circle. Only 2 of these 3 replicas can fail simultaneously, so one copy of the data will survive.

Failure detection is done using an all to all ping ack protocol, combined with a notification system. When a node detects that another node has either failed or joined the network, it notifies all other running nodes using a unicast, and the cluster enters a stabilization state.

Stabilization takes place in stages. First, each node must update its membership list. When a node is notified of a failure or a join, it scans its membership list for the relevant process, and either adds the node or removes it as appropriate.Second, once the membership list is repaired, each node updates its neighbors to the appropriate living process.

Finally, each node initiates a scan of the keys stored locally. For each key stored, the node routes it to determine the direct owner of the key. Then it does one of the following actions: 1) if the node itself owns the key, it sends a copy to both of its neighbors. 2) if the node's right neighbor owns the key, it sends a copy to its right neighbor. 3) if the node's left neighbor owns the key, it sends a copy to its left neighbor. 4) If the key belongs to neither the node, or either of its neighbors, it eliminates the key from its local store. Stabilizing in this manner ensures that no node ever stores keys unnecessarily, since any extra key stores are wasteful.

Stabilization is therefore an O(nlogn) time operation on each node.

Scalability

This node would scale beyond 10 nodes with some adjustments. Each node is required to store a sorted membership list, which stores O(n) integers and requires O(logn) time to update. Routing with an up to date membership list always takes O(logn) time. Replica storing would be more expensive, since 2 replicas would most likely not be sufficient in a larger system. The number of replicas would have to factor in a realistic cluster size, combined with a worst case scenario of how many nodes could fail simultaneously.

Executing Commands

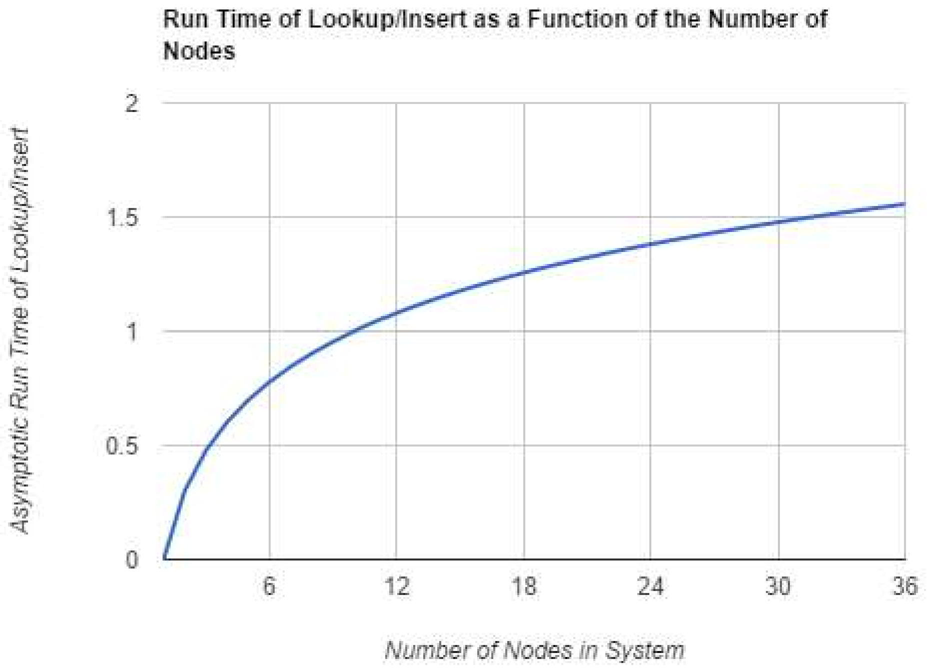
In order to execute a get or set command passed by a client, a node first *routes* the relevant key to determine who owns that key. This is done via a binary search on the membership list stored locally. The command is then sent via an RPC function call to the appropriate process. Thus, any command only requires O(logn) time to route, and O(1) message bandwidth. This comes at a cost of storing O(n) state at each node.

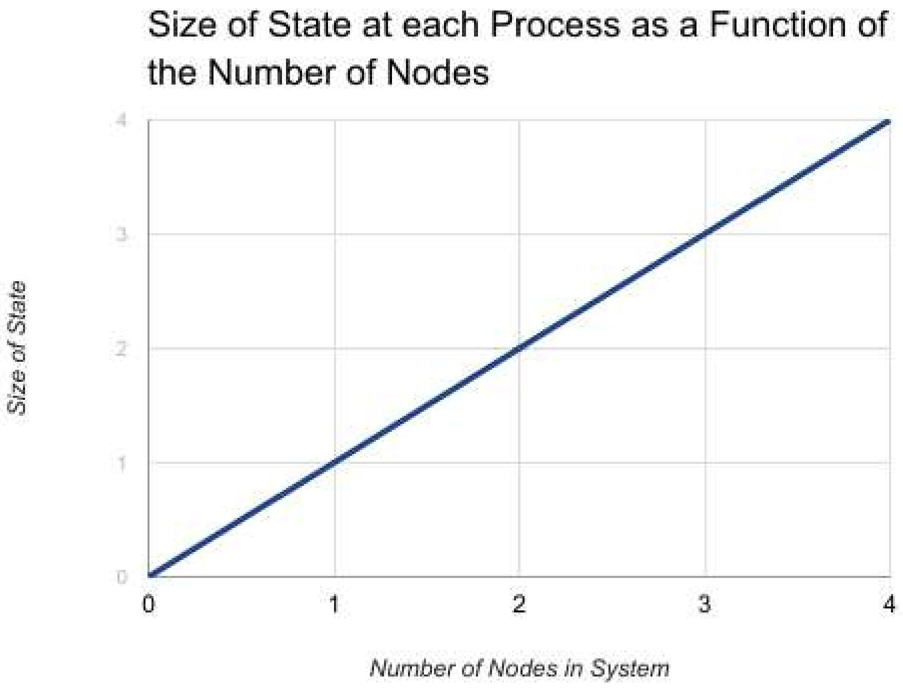
List local and owners commands are all executed entirely on the machine that accepted the command from a client. List local requires simply a O(n) table scan on local keys, and owners requires O(logn) time to determine the owners of each key.

The decision to use a binary search on a sorted membership list, rather than the finger tables in a standard Chord implementation greatly simplifies the routing code. No finger tables need to be updated, and binary search for the proper owner is very straightforward. It also reduces the amount of messages that need to be forwarded via RPC. Rather than O(logn) jumps, it's O(logn) search time and 1 jump.

How We Tested:

We wrote some BATCH scripts to ensure GET, SET, OWNERS, and LIST_LOCAL all worked. We also simulated (simultaneous and/or a series of) failures and verified whether data shifted around properly. Additionally, we tried to simulate simultaneous client calls to test synchronization. We conducted these test on different sized distributed hash tables and with failures in different locations.


Run Time of Lookup/Insert as a Function of the Number of Nodes


Size of State at each Process as a Function of the Number of Nodes