The Distributed Key-Value Storage protocol:

For our distributed key-value store, we are using a simple DHT protocol based on Chord which achieves eventual consistency. In our system, nodes are arranged in a ring-based network similar to Chord. Each node's location in the ring is predetermined based on its hostname/IP address. Each node is aware of the rest of the cluster, and keeps a list of all nodes currently alive in the network.

Routing Algorithm

The routing algorithm is very similar to Chord's, but we sacrifice on memory to decrease the lookup time to *O(1)*. We use a simple MD5 hashing algorithm to uniformly distribute the keys among the 10 possible nodes. Whenever a new key-value pair being stored or requested at any peer, it calculates the a hash of the key mod N, where N is the total number of nodes possible in the cluster. Specifically:

$$h = int(md5(key)) \ mod \ N$$

The key-value pair is then sent directly to that node:

> Say nodes 1, 5 and 7 are active in a ring where N = 10. Node 1 stores all keys hashed from 8-1, node 5 stores all keys hashed from 2-5 and node 7 hashes all keys between 6-7. If node 1 receives a SET request, and the key's hash has value 3, it will know that this key will need to stored in node 5 because that is the smallest alive node with ID > 3. It then sends the key-value pair to node 5 to be stored.

If new node joins, a stabilization protocol is run by the new node's successor and predecessor (based on the ring topology).

> If node 9 joins the network, 9 now stores all keys hashed from 8-9 and 1 now stores all keys hashed to 1. The key stores of all other nodes are unaffected.

We can see how this system achieves both constant lookup time (the owner of any key-value pair can be directly located and contacted by any node) as well as a relatively simple (and hence rapid) stabilization protocol for the joining of new nodes.

Failures

The design document lists that our key-value store should tolerate up to 2 simultaneous failures. In order to deal with failures, replicas of each node's data are stored at its

predecessor and successor node. Hence, if a node is found to have failed during a lookup, its data can still be retrieved from either one if its replicas. This satisfies allowing a maximum of two simultaneous failures to be tolerated without data loss - having three successive nodes simultaneously fail will cause the middle node's data and both its replicas to be lost, effectively removing all copies of the data from the system. Node failures are detected via TCP's ACK protocol, as the socket is maintained between any pair of nodes. All nodes, once they detect a failure, update their alive list to no longer contain that node. In addition, if the failed node happens to be the successor or predecessor of the detecting node, the detecting node runs the stabilization protocol including merging the failed node's replica with its own data and sending the appropriate data to its new successor and predecessor (i.e. push stabilization). Hence, we satisfy eventual consistency in the case of failures, and we will eventually return to having three copies of all data on the system.

Advantages over other algorithms

In implementing key-value storage, we considered using modified versions of other Consensus systems like Raft or DHT's like Cassandra, as well as arranging nodes using a tree-based structure instead of Chord's ring-based structure. Most alternatives ended up requiring us to implement one of leader election or totally-ordered multicasting, both of which would greatly delay stabilization: a failed leader would require a leader election to occur before stabilization could occur; and totally-ordered multicasting requires $\mathbb{O}(n^2)$ messages to be sent, which would be too slow for efficient stabilization.

Future work

Although our protocol is simple and relatively fast, it only guarantees eventual consistency, which is quite weak. Implementing a Raft-like consensus algorithm would give us even stronger consistency measures. Additionally, a more efficient stabilization protocol would reduce the load on the network when there is a lot of data on each node.

Metrics

In order to evaluate the performance of our Key-Value storage system, we collected the following data.
We ran a BATCH operation of 10 SET and GET operations each, the results of which have been displayed in the table and graph below. We can see that as the number of nodes in the cluster increases, the time to complete the BATCH operations does not change, showing that **routing time in our protocol is constant with respect to the number of messages sent**.

No. of Nodes	*BATCH SET (seconds)*	*BATCH GET (seconds)*
3	2.23	2.24
4	2.24	2.24
5	2.26	2.24
6	2.25	2.26