MP2 Design Document
CS 425
Aadil Bhatti (abhatti2)
Krishna Chaitanya Kura (kura2)
April 10, 2017

System Design

In this assignment, we chose to implement the Kademlia Distributed Hash Table (DHT) algorithm. We chose this algorithm due to the fact that it has a simpler implementation than Chord or Cassandra while maintaining the fast lookup times held by those DHTs. Kademlia nodes maintain a subset of the network (much like Chord) held in a binary search tree-like structure. The nodes contain a routing table which holds this structure. The "tree" is organized by distance, i.e. the higher the difference is between two node IDs, the further away they are in the tree. This distance is computed with the traditional XOR function.

More specifically, the "tree" is a list of lists. Suppose our system has 2^k nodes (i.e. we are representing all IDs with k bits). Then we structure the tree as a list of length $log(2^k) = k$, with each sublist of length at most k. The individual sublists are referred to as k-buckets since they can store up to k node references. Then we are storing $O(log\ N)$ node references in each of the k lists, where $N =$ the number of nodes in the system. Most of the time (in a larger system) we will only store a subset of the node data, much like Chord. This results in total $O(log\ N)$ storage per node. The buckets are organized by distance: the buckets towards the ends of the list contain nodes further in distance from the source node than the buckets towards the middle of the list. This is important for the reason that all of our lookup operations depend on querying a subset of nodes in *one* of our k lists. Since the buckets are organized by distance, we pick the bucket which contains the k closest nodes in distance to the hashed value of the key. In a lookup operation, we are sure that these nodes will take us one step closer to the key since they are closest nodes that we know about to the hashed value of the key. Since the distance shortens each time and we continue to only query the k-closest nodes in each round of iteration, we ultimately (see the paper for a more rigorous proof) end up with $O(log\ N)$ network bandwidth and search hops.

Replication is done due to the Kademlia requirement for lookups to return the k-closest nodes to a key. Because of this requirement, we store the key in each of these k nodes. These allow for fault-tolerance: if a node fails, we still have the key present in the k-closest nodes to the key. In addition, we restore keys every T time units to make up for any failed nodes, which will maintain a similar amount of keys at each node.

For new nodes joining, we *generally* have no added latency per lookup (in theory). Since we have $O(log\ N)$ search steps, this does not grow substantially with more nodes. As for capacity per node, since we republish all keys periodically, the capacity per node should

decrease as more nodes increase. We do get a slight increase in network traffic/overhead per node join as there will be more RPCs being sent over. Failure detection bandwidth will drop as well since we have more network requests being made periodically for failure detection. Finally, failure detection time will decrease since we have more nodes to detect other nodes' failures. For our implementation we chose k-size to be 3 since the requirement was to tolerate at most 2 failures. For alpha which is the no of parallel requests in our node lookup we chose alpha to be 1. This would make our implementation much simpler and in terms of message cost it would be similar to Chord's lookup algorithm. The latency can be shortened by augmenting the individual routes with RTT times which we did not implement.