Network-Centric Computing - CS 382

Assignment 2: Chord DHT

Fareed Zaffar, Hassan Shahid Khan, Wajih Ul Hassan, Gohar Irfan Chaudhry

Due Date: 12th March 2015

Submission Instructions:

- Please use the following naming convention: <Roll# Person 1>_<Roll# Person 2>_Assignment2.zip
- You should show all the message passing between adjacent nodes by printing it on the console *in a readable format*!
- **We will NOT accept any submissions via e-mail.** There will an LMS tab for late-submissions as well. There are no exceptions to this rule.
- Be prepared to justify the code you have submitted in vivas. We take plagiarism concerns very seriously in this course.
- Please work in the same groups of two as before.
- Please direct all questions related to the assignment to the discussion forum at Piazza.
- Late submissions will be penalized 10% per day.

This assignment has been adapted from the Computer Networks course at the Swiss Federal Institute of Technology Zurich (ETH).

Introduction to Chord:

Chord is a distributed hash table (DHT) protocol currently under development at MIT. It was proposed in 2001 in a paper titled ”Chord: A Scalable Peer-to-peer Lookup Service for Internet applications. From an applications perspective, Chord simple provides a service that can store key-value pairs and find the value associated with a key reasonably quickly. Behind this simple interface, Chord distributes objects over a dynamic network of nodes, and implements a protocol for finding these objects once they have been placed in the network. Every node in this network is a server capable of looking up keys for client applications, but also participates as key store. Hence, Chord is a decentralized system in which no particular node is necessarily a performance bottleneck or a single point of failure (if the system is implemented to be fault tolerant). Check out the wiki page for Chord for a more detailed description.

1.1 Key space:

Every key inserted into the DHT must be hashed to fit into the keyspace supported by the particular implementation of Chord. The hashed value of the key will take the form of an **m** bit unsigned integer. Thus, the keyspace (the range of possible hashes) for the DHT resides between

0 and 2^m – 1, inclusive. Standard practice for most DHT implementations is to use a 128 or 160 bit hash, where the hash is produced by a message digest algorithm such as MD5 or SHA-1. Using these hashing algorithms ensures with high probability that the hashes generated from keys are distributed evenly throughout the keyspace. Note that this does not restrict the number of distinct keys that may be stored by the DHT, as the hash only provides a guide for locating the key in the network, rather than providing the identifier for the key. It is possible, though unlikely, for the hash values of distinct keys to collide.

1.2 The ring:

Just as each key inserted into the DHT has a hash value, each node in the system also has a hash value in the keyspace of the DHT. To get this hash value, we could simply give each node a distinct name (or use the combination of IP and port) and then take the hash of the name, using the same hashing algorithm we use to hash keys inserted into the DHT. Once each node has a hash value, we are able to give the nodes an ordering based on those hashes. Chord orders the node in a circular fashion, in which each node's successor is the node with the next highest hash. The node with the largest hash, however, has the node with the smallest hash as its successor. It is easy to imagine the nodes placed in a ring, where each node's successor is the node after it when following a clockwise rotation. To locate the node at which a particular key-value pair is stored, one only needs to find the successor to the hash value of the key. Keep in mind that a node stores keys ranging from its predecessor to itself.

1.3 The overlay:

The Chord paper states it would be possible to look up any particular key by sending an iterative request around the ring. Each node would determine whether its successor is the owner of the key, and if so perform the request at its successor. Otherwise, the node asks its successor to find the successor of the key and the same process is repeated. Unfortunately, this method of finding successors would be incredibly slow on a large network of nodes. But for the purpose of this assignment we will be implementing a simple DHT which handles requests in O(n).

1.4 Dynamic Addition/Removal of Nodes:

Chord would be far less useful if it were not designed to support the dynamic addition and removal of nodes from the network, requiring a static allocation of nodes instead. A production ready implementation of Chord would support the ability to add and remove nodes from the network at arbitrary times, as well as cope with the failure of some nodes, all without interrupting the ongoing client requests being served by the DHT. However, this functionality complicates the implementation considerably. To allow membership in the ring to change, protocols for creating a ring, adding a node to the ring, and leaving the ring must be defined.

Creating the ring is easy. The first node leaves its predecessor and successor as empty. Then, when node A joins the network, it asks an existing node in the ring to find the successor of the hash of A. The node returned from that request becomes the successor of A. The predecessor of A is undefined until some other node notifies A that it believes that A is its successor. In order to determine the successor and predecessor relationships between nodes as they are added to the network (and

voluntarily removed), each Chord node performs stabilize operating periodically. This function is described below.

n - this node
h - hash of n
m - the number of bits in a hash

stabilize()

```
x = successor.predecessor
if x in (n, successor]
        successor = x
        successor.notify(n)
```

notify(p)

```
if predecessor is None or p in [predecessor, n)
        predecessor = p
        // transfer appropriate keys to predecessor
```

The following method is called when a node leaves the network:

leave()

```
// transfer all keys to successor
successor.predecessor = None
predecessor.successor = successor
```

Assignment Details:

You will be implementing a simple version of Chord. Your implementation need not be fault tolerant, we are stressing correctness over performance for this assignment. To remove unnecessary complications, you should treat key-value pairs in the DHT as immutable. This means that once a key-value pair is inserted into the DHT, it cannot be deleted and the value associated with the key should not change (make sure to enforce this requirement). Your implementation of Chord should support dynamic insertion and removal of nodes, and continue to serve **get** and **put** requests simultaneously and correctly (if a value exists for a key, it must always be accessible). Keys should never reside at more than two nodes at any given time, and only one node when the ring is in a stable state. As specified above, you should implement the following methods:

- *create()*
- *join(node)*
- *leave()*
- *get(key)*
- *put(key, value)*
- *stabilize()*
- *notify(p)*

The protocol for communication between the nodes has been left to you. You should explain your decision in the report.

Testing:

For testing purposes, you should run your Chord implementation on a number of Rustam nodes. Since only a couple of the nodes are working, you can run multiple instances running on the same machine but listening on different ports. In addition since Rustam has a distributed file system, meaning that each node has access to files from all other nodes, you will require each node to create a separate folder where it will store its keys etc. on creation and delete this folder when leaving. Your solution needs to provide the possibility of serving get and put requests. When the lookup operation takes place, once you have found the correct file, for testing purposes, you can automatically open that file. Also, remember that nodes may dynamically join or leave the ring, in which case the finger tables need to be updated and the ring needs to be stabilized. A ring is stabilized when all nodes have the correct successors and predecessors and any key is stored at only one node at any given time. In order to test the correctness of your implementation, you need to dump the state of each node periodically (keys stored, successors and the finger table content), create a function that does that collects the state from each node

Hints:

- It would be helpful to look up the MessageDigest class javadocs to get started with how to take hashes in Java
- Don't confuse yourself when trying to compare hashes, you will not need to know the *quantitative* difference between two hashes for this assignment. Keep it simple, simply check whether Hash A is larger than Hash B (and not the underlying strings).