Compsci 512: Distributed Systems
Assignment 2
Implementing Chord

Instructor: Bruce Maggs

March 1, 2015

OVERVIEW

In this assignment, you will implement the Chord Distributed Hash Table (DHT) scheme. You should reuse portions of the code that you wrote for the first assignment, in which you implemented a Web proxy server. We would like you to write your program in C and you provide a Makefile so that it can be easily compiled and tested. **Any other programming language must be approved in advance by the teaching assistant.** Your program should be designed so that it will operate if different Chord nodes are run on different machines communicating over network sockets. But for testing you may simply run multiple instances of the same program on the same machine, with each instance listening on a different port.
Your program should be invoked in one of two ways. Both ways create a new Chord node. In the first example, a new Chord ring is created, consisting of a single node, which is listening for Chord messages on port 8001.

```
chord 8001
```

After the program is started, the new node must maintain its position in the Chord ring, assist other new nodes in joining the ring, and implement searches.
Alternatively, if the new Chord node is to join an existing Chord ring, then the program could be invoked as follws:

```
chord 8001 128.2.205.42 8010
```

```
Joining the Chord ring.
You are listening on port 8001.
Your position is 4320fbb2.
Your predecessor is node 128.2.205.42, port 8010, position 033df12a.
Your successor is node 128.2.205.42 at port 8010, position 033df12a.
```

(You may want to provide even more information, for debugging purposes.)
In this example, a new Chord node is created. This node listens for Chord messages on port 8001, and it joins an existing Chord ring by contacting node 128.2.205.42 on port 8010. Note that the port numbers 8001 and 8010 are just examples, and you can use other port numbers. Also note that on certain platforms (e.g., MacOS X) you may need to use sudo to run the program with root privileges in order to have access to network ports.
You should also write a separate program called `query` that allows you to issue queries to a Chord node. This program doesn't actually do any searching on its own. It might be invoked as follows:

```
query 128.2.205.42 8010
```

In this example, no new Chord nodes are created, but a connection is made to Chord node 128.2.205.42 on port 8010 and then search queries are passed to the node. Output from this program might have the following form.

```
query 128.2.205.42 8010
Connection to node 128.2.205.42, port 8010, position 033df12a:
Please enter your search key (or type "quit" to leave):
Gettysburg Address
Hash value is 02b21c2d
Response from node 128.2.205.42, port 8010, position 033df12a:
Not found.
Please enter your search key (or type "quit" to leave):
quit
```

You will have to design the format of the messages that the Chord nodes exchange. It is highly recommended that these messages be encoded in plain text, so that you can debug your program easily.

MAINTAINING A SIMPLE RING

Chord uses consistent hashing to map nodes to positions on a circle. You should use the SHA-1 hash algorithm applied to a node's IP address and listening port number to determine where the node appears on the circle. You do not need to program the SHA-1 algorithm from scratch. You may use a library or use another existing implementation. The hash algorithm produces a 20-byte hash value. If you find 20-byte values inconvenient to work with, you could convert the 20-byte value to a 4-byte (32-bit) by, for example, breaking the 20-byte value into 5 4-byte chunks and XORing them together.

Each node should maintain pointers to the next two nodes that appear on the ring and the previous two nodes that appear on the ring. You will have to be careful when the ring consists of fewer than five nodes!

When a new node attempts to join the ring, the node receiving the request should do a search to determine where the new node should appear in the ring. For this assignment, rather than passing the search query around the ring, the node receiving the initial request should perform repeated queries directly to other nodes on the ring until it finds the correct position. Once it finds the position where the new node should appear, it should provide the new node with the identity of the new node's successor node. The new node should then contact the successor node (and possibly others), and insert itself into the ring.

For this assignment, you may assume that only one node joins the ring at a time, although it would be interesting for you to think about how you might handle the case in which multiple nodes join simultaneously.

Nodes that appear in successive positions on the ring should send periodic "keep-alive" messages to each other (e.g., every 5 seconds). If a node fails to respond to these messages, then its predecessor and successor must splice it out of the list.

Your implementation should allow a Chord node to leave the ring in an orderly fashion. For example, the user might provide an input to the program indicating that it should leave the ring. Your implementation should also work if the a node leaves the ring abruptly, e.g., if you kill the chord process using the `kill` command. For this assignment you may assume that only one node leaves the ring at a time.

SEARCH

The Chord nodes should support search. In particular, a user should be able to present a search key to a Chord node in the form of a text string (e.g., "Gettysburg Address"). (I.e., your program should allow the user to type in search strings, and then perform the searches.) The node should compute a SHA-1 hash of this string to determine the position in the ring at which any document with the key "Gettysburg Address" is stored. The search algorithm should be the same as the algorithm that is used to find the new position of a node in the ring. I.e., the node receiving the search request should perform repeated queries until it finds the node to follow the position of the key.

It is not necessary for your implementation to actually store any (key,value) pairs at the Chord nodes. (Although it might be fun.) Consequently, you do not have to support operations such as inserting or deleting (key,value) pairs. Because no (key,value) pairs are to be stored, when the search request reaches the correct node, that node should respond with a message saying that it acknowledges the request, but does not have an object with the requested key.

SHORTCUTS

One disadvantage of the design, as explained so far, is that joining a ring of n nodes, or performing a search on a ring of n nodes may require $\Omega(n)$ messages to be sent between the nodes, resulting in both long delays and a lot of network traffic.

The Chord paper explains how the number of messages can be reduced to a logarithmic number by having each node maintain a logarithmic number of pointers. In particular, suppose that we are using 32-bit (4-byte) values to represent positions on the ring. (E.g., we have taken the 20-byte SHA-1 hash value and reduced it to 4-bytes.) Then a node at position p would store pointers to the nodes that appear first after positions $p+1$, $p+2$, $p+4$, $p+8$, …, $p+2^{31}$. (Where all values are taken mod 2^{32}.)

You should add the functionality to your program to maintain these pointers. Once these pointers have been added, your join and search algorithms should require only a logarithmic number of messages to be sent (logarithmic in the range of positions), rather than a number linear in the number of nodes.

Note that when a node joins or leaves the ring, it may be necessary for many nodes to update their pointers. This is probably the most difficult part of the assignment. One approach is to make all pointers bidirectional, so that when a node leaves, it can directly contact all of the nodes that point to it.