Simple DHT

Introduction

Although the design is based on Chord, it is a simplified version of Chord; node failures are not handled. Therefore, there are three things this project implements:

1) ID space partitioning/re-partitioning, 2
2)) Ring-based routing, and
3) 3) Node joins.

Content provider implements all DHT functionality (including communication using sockets) and support insert and query operations. When running multiple instances of the app, all content provider instances should form a Chord ring and serve insert/query requests in a distributed fashion according to the Chord protocol.

References

Before we discuss the requirements of this assignment, here are two references for the Chord design:

1. Lecture slides on Chord
2. Chord paper

The lecture slides give an overview, but do not discuss Chord in detail, so it should be a good reference to get an overall idea. The paper presents pseudo code for implementing Chord, so it should be a good reference for actual implementation.

We will use SHA-1 as our hash function to generate keys. The following code snippet takes a string and generates a SHA-1 hash as a hexadecimal string. Given two keys, we can use the standard lexicographical string comparison to determine which one is greater.

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Formatter;

private String genHash(String input) throws NoSuchAlgorithmException {
        MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
        byte[] sha1Hash = sha1.digest(input.getBytes());
        Formatter formatter = new Formatter();
        for (byte b : sha1Hash) {
                formatter.format("%02x", b);
        }
        return formatter.toString();
}
```

Step 1: Writing the Content Provider

This content provider implements all DHT functionalities. It creates server and client threads, opens sockets, and respond to incoming requests; it should also implement a simplified version

of the Chord routing protocol; lastly, it also handles node joins. The following are the requirements for the content provider:

1. The app may be tested with any number of instances up to a total of 5.
2. The content provider implements all DHT functionality. This includes all communication using sockets as well as mechanisms to handle insert/query requests and node joins.
3. Each content provider instance has a node id derived from its emulator port. *This node id is obtained by applying the above hash function (i.e., genHash()) to the emulator port.* For example, the node id of the content provider instance running on emulator-5554 should be, *node_id = genHash("5554")*. This is necessary to find the correct position of each node in the Chord ring.
4. The content provider implements insert(), query(), and delete(). The basic interface definition is the same as the previous assignment, which allows a client app to insert arbitrary <"key", "value"> pairs where both the key and the value are strings.
    a. For delete(URI uri, String selection, String[] selectionArgs), you only need to use use the first two parameters, uri & selection. This is similar to what you need to do with query().
    b. However, please keep in mind that this "key" should be hashed by the above genHash() before getting inserted to your DHT in order to find the correct position in the Chord ring.
5. For your query() and delete(), you need to recognize two special strings for the *selection* parameter.
    a. If * (not including quotes, i.e., "*" should be the string in your code) is given as the *selection* parameter to query(), then you need to return all <key, value> pairs stored in your entire DHT.
    b. Similarly, if * is given as the *selection* parameter to delete(), then you need to delete all <key, value> pairs stored in your entire DHT.
    c. If @ (not including quotes, i.e., "@" should be the string in your code) is given as the *selection* parameter to query() on an AVD, then you need to return all <key, value> pairs *stored in your local partition of the node*, i.e., all <key, value> pairs stored locally in the AVD on which you run query().
    d. Similarly, if @ is given as the *selection* parameter to delete() on an AVD, then you need to delete all <key, value> pairs *stored in your local partition of the node*, i.e., all <key, value> pairs stored locally in the AVD on which you run delete().
6. An app that uses your content provider can give arbitrary <key, value> pairs, e.g., <"I want to", "store this">; then your content provider should hash the key via genHash(), e.g., genHash("I want to"), get the correct position in the Chord ring based on the hash value, and store <"I want to", "store this"> in the appropriate node.
7. The content provider implements ring-based routing. Following the design of Chord, your content provider maintains predecessor and successor pointers and forward each request to its successor until the request arrives at the correct node. Once the correct node receives the request, it processes it and returns the result to the original content provider instance that first received the request.
    a. As with the previous assignment, we will fix all the port numbers (see below). This means that you can use the port numbers (11108, 11112, 11116, 11120, & 11124) as your successor and predecessor pointers.
8. The content provider handles new Chord joins.

9. We have fixed the ports & sockets.
    a. The app opens one server socket that listens on 10000.
    b. Use run_avd.py and set_redir.py to set up the testing environment.
    c. The grading will use 5 AVDs. The redirection ports are 11108, 11112, 11116, 11120, and 11124.
    d. The above 5 ports are hardcoded to set up connections

Testing

We have testing programs to help you see how your code does with our grading criteria. If you find any rough edge with the testing programs, please report it on Piazza so the teaching staff can fix it. The instructions are the following:

1. Download a testing program for your platform. If your platform does not run it, please report it on Piazza.
    a. Windows: We've tested it on 32- and 64-bit Windows 8.
    b. Linux: We've tested it on 32- and 64-bit Ubuntu 12.04.
    c. OS X: We've tested it on 32- and 64-bit OS X 10.9 Mavericks.
2. Before you run the program, please make sure that you are running five AVDs. `python run_avd.py 5` will do it.
3. Run the testing program from the command line.
4. On your terminal, it will give you your partial and final score, and in some cases, problems that the testing program finds.