ECE428/CS425/CSE424: Distributed Systems Fall 2011
Programming assignment 2 Due: by 5:00p.m. on Dec 2, 2011

Distributed Hash Table

In this programming assignment, you will be implementing a variant of the Chord system [1]. To simplify your job, you are provided with a hash function code that generates node identifiers and keys for file lookups. Your main task in this assignment will be to implement (i) node joins, (ii) adding a file, (iii) deleting a file, and (iv) searching for a file.

Constraints and Assumptions

1) There are no node failures; 2) The keys and the node ids are *m* bits in size (5 <= *m* <=10); 3) Use sockets for communication between nodes; 4) Each node in the system should be simulated by a separate process.

Specification

You are to use the UIUC College of Engineering EWS workstations, named `linux.ews.illinois.edu`. Information about the workstations is available on the homepage

```
http://www.ews.uiuc.edu
```

Make a directory called MP2 somewhere inside your main directory and store all your files in this directory. Use only C or C++ for developing the system.

Hash Function

In this programming assignment you will be using a SHA-1 hash function for generating the keys corresponding to a filename. We have provided a library that implements the hash function. You may untar the library stored in the ece428 directory:

> tar -xf /class/ece428/ece428_sha_library.tar

You will see a directory called ***mp2_sha1-c***, which will contain three files, namely *sha1.c, sha1.h, key_gen_test.c*. The hash function is implemented in *sha1.h* and *sha1.c*. To understand how to use the library files, please refer to *key_gen_test.c.* In particular, refer to the statement `SHA1Input(&sha, argv[1], strlen(argv[1]));` to understand how to call the hash function in your code and the statement `key_id = sha.Message_Digest[4]% (1<<m);` to understand how a key is generated from a file name. The keys generated by this library will be a 160 bit SHA-1 key modulo 2^m, where *m* (in bits) denotes the size of the node IDs and the keys. Your code should read *m* from the command line, as described later. We will test your code with values of *m* between 5 and 10 (5<= *m* <= 10). (Please make sure that when you compile your code with this library, you need include <math.h> and when you link your codes with this library, you need an option like '-lm' to use math library)

Design Details

The Chord system that you will design should be capable of supporting node joins, adding a file, deleting a file, and looking up for a file. In this assignment, you will be not be storing actual files, but you will instead store an attribute of the file in the form of an IP address. The IP address is intended to denote the actual device where the file is located. The IP address will be passed on as a parameter while adding a file to the Chord system and should be in a dotted decimal format of the form 192.168.xx.xx. Your code will not need to access the hosts with the IP address you supply.

Multiple nodes may join the Chord system concurrently. Your code, therefore, should be capable of handling concurrent node joins. Each time a node joins, your code should create a new process. Furthermore, you should also maintain a finger table at each node that joins the system. Each node that joins the system will be assigned a node ID (a positive integer), passed on as a parameter while adding the node. We will test your code with up to 10 nodes in total.

It is possible that more than one file may be mapped to the same key by the hash function. This may result in multiple files mapped to the same key. However, your code should be capable of handling this situation and must return the correct file information when queried. Please explain how you handled this situation in your report.
You must have a file named *chord_sys.c* or *chord_sys.cc*. This file should be executed form the command line as follows:

>./chord_sys <m>

where *m* is the parameter (between 5 and 10) to be passed to the hash function, as described earlier. Upon execution from the command line, your code should create two processes, say **Listener** and **Introducer.** The functionalities of these two processes are as follows:

Listener: The Listener process should always be listening for user inputs from the terminal. The user input may be one of many commands that are described later. The Listener should be capable of passing the commands issued by the user to the Introducer process (using a socket connection) for further action.

Introducer: The Introducer is a default node of the Chord system. The Introducer process should be capable of receiving the commands from the listener and processing them accordingly. The Introducer is also the node that will be contacted first during node joins or file lookups. The node ID for the Introducer should be 0.

The commands that are to be supported by your system are detailed in the following table:

Command name	Usage	Expected output (on screen)	Function
ADD_NODE	ADD_NODE <node ID(s)> E.g., ADD_NODE 23 13 // Add two nodes with IDs 23 and 13.	1) The finger table of the node(s) added.	The ADD_NODE command should accept a list of node IDs from the command line and i) initiate a node-join procedure to add that many number of nodes to the Chord system, ii) create the finger tables for the new nodes while updating the finger tables of the existing nodes, iii) maintain the consistency of the finger tables that are created/updated when more than one node is added concurrently.
ADD_FILE	ADD_FILE <filename> <IP address> E.g., ADD_FILE f1 192.168.1.1 //Add a file named f1 with the IP address 192.168.1.1 as its attribute.	1) The key obtained as a result of hashing the filename, 2) The node ID that stores the key.	The ADD_FILE should take in a <filename> from the command line, and i) find the corresponding key using the hash function provided, ii) store the file information (e.g., IP address) at an appropriate node determined by the Chord system.
DEL_FILE	DEL_FILE <filename> E.g., DEL_FILE f1 //Delete file f1.	1) The key obtained as a result of hashing the filename, 2) A message saying the file is deleted, (or) 3) An error message if the file to be deleted cannot be found.	The DEL_FILE should read in a <filename> and i) find the corresponding key using the hash function provided, ii) delete the corresponding file information from the appropriate node.
FIND_FILE	FIND_FILE <filename> E.g., FIND_FILE f1 //Get the information of file f1.	1) The key obtained as a result of hashing the filename, 2) The node id that stores the key, 3) The IP address stored as part of the file information, (or) 4) An error message if the file cannot be found.	The FIND_FILE should accept a filename and i) find the corresponding key using the hash function provided, ii) find the node ID that stores the key.

In addition to the above Chord system commands, you should also be implementing the following queries:

Command name	Usage	Expected output (on screen)	Function
GET_TABLE	GET_TABLE <node id> E.g., GET_TABLE 23 //Prints the finger table of node 23 along with the keys stored at 23	1) The finger table of the node, 2) The keys stored at the node. (or) 3) An error message saying no such node exists.	GET_TABLE should print the current finger table of the corresponding node along with the list of keys it stores.
QUIT	QUIT	1) Exit the code and return to the UNIX prompt.	QUIT should kill all the processes created by your code, close all the socket connections, and gracefully exit the code.

Experiments

The above commands may be given in any order and your code should be capable of returning the correct output for each command. You may use an input files that supplies the commands one after another to your code. You may either choose to issue a `sleep` command after every command or supply a batch of commands at once. When more than one command is issued in a batch, your code should treat them to be concurrent and manage them accordingly. An example input file is shown below (the comments are for explanation only and need not be present in your file) :

```
ADD_NODE 1                  //Add a node with ID 1
SLEEP 10                    //Sleep for 10 seconds
ADD_NODE 32  10  11         //Add 3 nodes with IDs 32, 10, and 11
ADD_FILE f1  192.168.10.1   //Add a file named f1 with IP address attribute 192.168.10.1
GET_TABLE 32                //Print the finger table of node with ID 32, if it exists
SLEEP 20                    //Sleep for 20 seconds
ADD_FILE f2  192.168.10.2   //Add a file named f2 with IP address attribute 192.168.10.2
DEL_FILE f1                 //Delete a file named f1, if it exists
SLEEP 5                     //Sleep for 5 seconds
FIND_FILE f1                //Find the file named f1, if it exists and print its info
QUIT                        //Quit the program
```

In the above input file, the commands between SLEEP(s) will be treated concurrently.

Report

Submit a typed report of up to 5 pages (12pt. font). Your report should contain the following components:

- The names of all the team members.
- Explain how you managed to retrieve the correct file info when more than one file is mapped to the same key.
- The measurements and plots on number of messages for the given input file, along with your comments on whether the measurements make sense.
- How did you split up the work between the team members?

Submission

Your submission should contain the following:

1) The source code of all the components. Please do not turn in any executable or binary files (.o files).
2) A Makefile for compiling all your code.
3) Your report.

You will be asked to present or demo your code to the course staff. More information about the demo will be provided later.

Hand In

Please follow these instructions for handing in your programming assignments.

1) Log in to your EWS account.
2) Create a folder called MP2. Put all your assignment files in to this folder (*.c, *.h files, Makefile, and your report only; no *.o files, scripts, or text files shall be turned in).
3) Go to MP2 directory (*>cd MP2*) and type the following at the prompt:

>~ece428/Handin/handin 2

You may be able to submit multiple times until the deadline. We will only grade the latest version. Please let us know if you have any trouble getting the handin script working.

Grading policy

- (25 points) Correct execution of (ADD_NODE, ADD_FILE, DEL_FILE, FIND_FILE, GET_TABLE) = 5 points each x 5
- (5 points) Correct execution of QUIT
- (10 points) Handling multiple files mapping on to the same key
- (10 points) Handling concurrent node joins
- (10 points) Concurrent execution of commands in a batch (between SLEEP(s))
- (5 points) Correct Listener process
- (5 points) Correct Introducer process
- (20 points) Report
- (10 points) Clean code with comments and Makefile

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, ``Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications'', ACM Sigcomm 2001.