Programming Assignment 3 (MP3)
CS 425/ECE 428 Distributed Systems
Spring 2016
Assigned: April 2, 2016
Due: 7:00 p.m. April 26, 2016

This MP must be performed by groups of two students each. Any exceptions must be approved by the instructor.

The purpose of this assignment is to implement a peer-to-peer lookup service based on the paper "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications" from the SIGCOMM 2001 conference.

In particular, you will implement the **key insertion, key lookup (or key location)**, **node join, and handle node failures.** They are described in Section 4 and 5 of the above paper.

Assumptions

Assume that only one operation will occur at any time (i.e., while a key look up is in progress a new node will not join, and while a new node is in the process of joining, a new key lookup will not be initiated).

Assume that the node identifiers and keys both consist of 8 bits (i.e., in the range 0 to 255). We will not be hashing the keys or node identifiers (or, in other words, they are assumed to be already specified as hash values). If there is more than 1 node in the system, an extra replica of any key should be stored at the successor node.

Implementations

Simulate each node in the system using a collection of threads. You will need multiple threads to support the functions of each node. Node-to-node communications have to be made through message passing with delay similar to previous MPs. Keep in mind that we may test the MP with up to 32 nodes added to the system.

Configuration file specifies the min and max delay between all the node-to-node communications and the starting port (port for node 0).

> min_delay(ms) max_delay(ms)
> port number

Initialize your system to consist of a single node, with identifier 0, with keys 0 through 255 initially stored at node 0.

There should be a client that reads commands from the keyboard input and print out the response. The client should connect to all the nodes and propagate the user's input to the appropriate node and then wait for a response before executing the next input command.

Commands

Your code should support following commands, which are read from the keyboard input by the client. Client should contact the appropriate node directly.

- join p -- the client should create a collection of threads that will simulate the behavior of node p. This node should then take steps to add itself to the network. Use node 0 in place of node n' in Section 4.4 of the above paper. When a new node joins, some of the keys will be transferred to the new node. During the execution of "join p", make sure your client is only communicating with p.

- find p k -- client should inform node p to locate key k, upon finding, print out the node identifier q that contains k. (if node p does not exist or crashed, then print "p does not exist"). During the execution of "find p k", make sure your client is only communicating with p.

- crash p – the client should inform node p to **clean** crash. Node p will stay crashed unless join p is executed later. When a node crashes, no additional message is sent (the node does not inform the others about the crash in any form). You should implement failures detectors for the other nodes to learn that p has crashed (You may assume the maximum delay of any message is 5 seconds). You should ensure that after a "crash p" command, no key will be lost and the system will continue to work correctly. To ensure fault-tolerance, you need to maintain two copies of each key. You should determine when the crash command is finished and ready for the next command. (You may assume we will never execute "crash 0")

- show p – print out p, finger table at p, and the keys stored locally at p (if node p does not exist or crashed, then print "p does not exist"). During the execution of "show p", make sure your client is only communicating with p.

- show all – for all the node p in the system (in increasing order of the node identifiers), print out p, finger table at p, and the keys stored locally at p.

When each of the above operations is completed, the node p (specified in each command above) should inform the client (it can be other nodes for the "crash p" command) – the client can then process the next command. This will ensure that two commands will not be processed simultaneously. (However, note that a practical system needs to be able to handle overlapping operations too.)

Format

We will use the output of the "show" commands above for the purpose of grading. **Failure to implement the show commands will result in a zero grade.** The format for the output of the "show" commands, corresponding to any one node, should be as follows. f1 is the first entry in the finger table and f_last is the last entry in the finger table; key1 is the smallest key stored at the specified node, and key_last is the largest. The keys should be printed as integers separated by a single space. "show-all" should print such output for all the nodes, in an increasing order of the node identifiers.

node-identifier
FingerTable: f1, f2, f3, f_last
Keys: key1 key2 … key_last

Performance evaluation:

Repeat the following experiment N times without any node crashes, and report the average message overhead.

A single experiment is to be performed as follows:

- Phase 1: Add P nodes to the initial system, with their identifiers chosen randomly (taking care that a node that is already in the system is not added again)
- Count the total number of messages in Phase 1, to be able to derive the average number of messages required for each node addition.
- Phase 2: Perform the find operation F times, with p and k chosen randomly, taking care that node identifier p is randomly chosen from only the nodes that were added above. Any given key k can be repeated during the F find operations.
- Count the total number of messages in Phase 2, to be able to derive the average number of messages required for each find operation.

Run the experiment for N >= 5 (larger is better), for P = 4, 8, 10, 20, 30, 35, 50 and F = 128. Ensure that the different runs use a different seed for random number generation (otherwise, the different runs will produce identical data).

All trails F = 128	Phase1 avg. # of messages	Phase2 avg. # of messages
P = 4, N=		
P = 8, N =		
P = 10, N =		
P = 20, N =		
P = 30, N =		
~~P = 35, N =~~		
~~P = 50, N =~~		

Fill in the table that records average number of messages in phase 1 and 2 for each value of P (averaged over all N runs for each value of P).

Report
Include 1) the instructions for compiling and running your code, 2) the above table containing your performance data, 3) and one or two sentences about your observations from the experiments.

Rubric
To receive a score for the operation items below, they must be 1) correct with any number of nodes, 2) in the presence of any other combination of operations, 3) and the system must continue to operate correctly after the command.

- 5% data replication
- 20% join p
- 20% find p, k
- 20% fail p
- 20% report
- 5% basic code cleanliness (good comments, reasonable organization and design).
- 10% questions during demo.