COP5615 – Operating System Principles – Spring 2010
Department of Computer and Information Science and Engineering
University of Florida

Project 4

Due dates:	March 26, 2010 (Friday) 11:59 pm ET (local)
	March 28, 2010 (Sunday) 11:59 pm ET (EDGE)

1. Introduction

Project Overview

This project is a continuation of project #3 with the addition of the join and leave functions. On top of project #3, we want to build the join and leave functions such that nodes can participate in the network or eliminate themselves any time.

IMPORTANT: You had freedom of choosing from two options in project #3. However, in this project, you are required to use the second option (approach) from the previous project. It is straightforward to convert from approach one to approach two.

Project Specification

Unless otherwise specified in this document, specifications for project 3(option #2) is also valid for this project. We use the term identifier and node interchangeably in this project specification but both refer to the term identifier in project #3 specification.

Helpful Resources:

[1] Chord: http://pdos.csail.mit.edu/chord/
[2] Chord: Sections 2.2.2 and 5.2.3 in textbook
[3] Java RMI: http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html
[4] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001, San Deigo, CA, August 2001, pp. 149-160. PDF. NOTE: Be aware of typos in one of the figures. Correction of type is written in this document at figure 1.

2. Join (The following description is a slightly modified quote from [4])

In a dynamic network, nodes can join (and leave) at any time. The main challenge in implementing these operations is preserving the ability to locate every key in the network. To achieve this goal, Chord needs to preserve two invariants:

1. Each node's successor is correctly maintained.
2. For every key *k*, node *successor(k)* is responsible for k.

In order for lookups to be fast, it is also desirable for the finger tables to be correct. To simplify the join and leave mechanisms, each node in Chord maintains a *predecessor pointer*. A node's predecessor pointer contains the Chord identifier and URL of the immediate predecessor of that node, and can be used to walk counterclockwise around the identifier circle.
To preserve the invariants stated above, Chord must perform three tasks when a node *n* joins the network:

1. Initialize the predecessor and fingers of node *n*.
2. Update the fingers and predecessors of existing nodes to reflect the addition of *n*.
3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node *n* is now responsible for. (this part is unnecessary for our project)

We assume that the new node learns the identity of an existing Chord node *n'* by contacting the initial server. Node *n* uses *n'* to initialize its state and add itself to the existing Chord network, as follows.

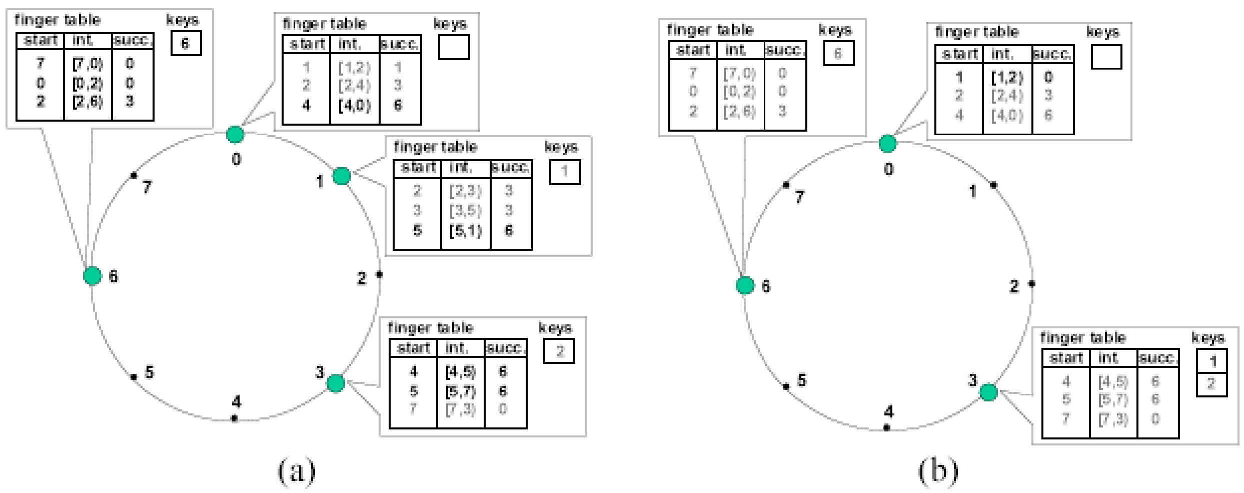

Figure 1: (a) Finger tables after node 6 joins. (b) Finger tables after node 1 leaves. Changed entries are shown in black, and unchanged in gray NOTE: The first entry in the finger table of node 0 has incorrect successor →it is 3 instead of 0. Please ignore the keys in the figure, as we do not simulate or implement the keys.

Initializing fingers and predecessor: Node *n* learns its predecessor and fingers by asking *n'* to look them up, using the *ini_finger_table pseudocode* in Figure 2. Naively performing *find_successor* for each of the m finger entries would give a run time of *O(mlogN)*. To reduce this, *n* checks whether the ith finger is also the correct $(i+1)^{th}$ finger, for each *i*, This happens when *finger[i].interval* does not contain any node, and thus $finger[i].node \geq finger[i+1].start$
As a practical optimization, a newly joined node *n* can ask an immediate neighbor for a copy of its complete finger table and its predecessor. *n* can use the contents of these tables as hints to help it find the correct values for its own tables, since *n*'s tables will be similar to its neighbors'.

Updating fingers of existing nodes: Node *n* will need to be encountered into the finger tables of some existing nodes. For example, in Figure 1(a), node 6 becomes the third finger of node 0 and 1, and the first and the second finger of node 3.
Figure 2 shows the pseudocode of the *update_finger_table* function that updates existing finger tables. Node n will become the i^{th} finger of node p if and only if (1) p precedes n by at least 2^{i-1}, and (2) the ith finger of node p succeeds n. The first node, p that can meet these two conditions is the immediate predecessor of $n - 2^{i-1}$. Thus, for a given n, the algorithm starts with the $(i)^{th}$finger of node n, and then continues to walk in the counter-clock-wise direction on the identifier circle until it encounters a node whose i^{th} finger precedes *n*.

IMPORTANT: There is a variation of the algorithm to prevent the problem caused by concurrent join (e.g. section 5 in [4] – periodic stabilization and update of finger table). In our project, we will assume sequential join and leave, therefore, you can ignore the second part (which we do not show in this document) from [4].

```
#define successor finger[1].node

// node $n$ joins the network;
// $n'$ is an arbitrary node in the network
$n$.join($n'$)
  if($n'$)
    init_finger_table($n'$);
    update_others();
    // move keys in $(predecessor, n]$ from successor
  else // $n$ is the only node in the network
    for $i = 1$ to $m$
      $finger[i].node = n$;
    $predecessor = n$;

// initialize finger table of local node;
// $n'$ is an arbitrary node already in the network
$n$.init_finger_table($n'$)
  $finger[1].node = n'.find\_successor(finger[1].start)$;
  $predecessor = successor.predecessor$;
  $successor.predecessor = n$;
  for $i = 1$ to $m - 1$
    if ($finger[i+1].start \in [n, finger[i].node)$)
      $finger[i+1].node = finger[i].node$;
    else
      $finger[i+1].node =$
          $n'.find\_successor(finger[i+1].start)$;

// update all nodes whose finger
// tables should refer to $n$
$n$.update_others()
  for $i = 1$ to $m$
    // find last node $p$ whose $i^{th}$ finger might be $n$
    $p = find\_predecessor(n - 2^{i-1})$;
    $p.update\_finger\_table(n, i)$;

// if $s$ is $i^{th}$ finger of $n$, update $n$'s finger table with $s$
$n$.update_finger_table($s, i$)
  if($s \in [n, finger[i].node)$)
    $finger[i].node = s$;
    $p = predecessor$; // get first node preceding $n$
    $p.update\_finger\_table(s, i)$;
```

Figure 2: Pseudocode for the node join operation

3. <u>Leave</u>

How to implement the leave function is up to you. We do not put any restriction; however, the implementation should be reasonable. Write logic of the algorithm in the report. Please refer to the figure 1(b) for the correctness of the result after the leave operation,

4. <u>Requirement</u>

The program is required to read the basic configuration first, then the command file.

System Configuration

You are required to make a system configuration file, `system.properties`. The `start` program will read system parameters necessary for server and clients (identifiers) from this file. We assume that the rmiregistry runs on the server. The format of `system.properties` is:

```
Rmiregistry.port=1099
Server=storm.cise.ufl.edu
numberOfNodes=8
```

Port number indicated at Rmiregistry.port should be used for all of the nodes. As we use approach #2 from project #3, server process may not be necessary depending on the implementation. Hence, for reading commands and initiating node threads, you can choose to ignore the URL of server process if it is easier for your implementation. However, your program should be able to read the very same format of `system.properties` given here.(e.g. parse the statement of server URL but ignore in execution) numberofNodes refers to the maximum number of nodes that can exist in the network). This information can be used for deciding the size of finger table. Note that we may use different `system.properties` file for grading (e.g. different number of nodes) but the format will be the same.

Reading commands

To test join and leave on your Chord system, your program is supposed to read command(s) from ***command*** file (no extension in the file name) and executes them in the system.

(1) *join.id=0*
This indicates that the identifier named 0 has joined the network. This is followed by the corresponding URL of the host represented by node 0. (e.g. host-name=lin114-01.cise.ufl.edu in the example below)

(2) *leave.id=1*
This indicates that the identifier named 1 has left the network. Thread for node 1 running on the remote computer should terminate after notifying appropriate node(s) to update their finger tables. Implementation detail for leave operation is up to each student as previously mentioned.

(3) *Exit;*
It means that all the requested functions in ***command*** have been completed. It terminates the Chord processes including the server and all nodes (identifiers). Note that identifiers have to output their finger table. (See more in sample output part)

The example format of the ***command*** file is following:

```
join.id=0
host-name=lin114-01.cise.ufl.edu
join.id=3
host-name=lin114-02.cise.ufl.edu
join.id=1
host-name=lin114-03.cise.ufl.edu
join.id=6
host-name=lin114-04.cise.ufl.edu
leave.id=1
leave.id=0
join.id=4
host-name=lin114-05.cise.ufl.edu
leave.id=6
leave.id=3
leave.id=4
Exit;
```

Your program should be able to take care of the situation where program terminates upon reading Exit; command when there are nodes that did not leave the network. (e.g. in above example, Exit; read before leave.id=4) We will use different command file for grading so test your program with other examples. You are encouraged to share different examples via discussion board (e.g. command file and finger table for each node) or we will upload the examples you send us on the course website.

start.java Requirements

Here is a list of actions that must be performed inside start.java

1. Read system configuration from `system.properties`.
2. Read command from command file.
3. Initiate node threads at corresponding remote machines and write a log after updating the finger tables of the nodes when it reads join command.
4. Write a log after updating the finger tables of the nodes and terminate the corresponding thread when it reads leave command.
5. Terminate the remaining node threads to finish the program and print "exit" message on the screen to notify the user of the end of program execution when it reads Exit;.

RMI registry

RMI registry should be started automatically by embedded code in each of remote machine where the node thread is initiated. Thus, we no long use manual typing of RMI registry command.

Compiling programs

You program should be able to be compiled by javac *.java in execution script. Creating a subdirectory and locating your source file under it will result in compile error. You will get zero points initially for compile error and can lose at least 20% of your grade even after fixing the error.

Running the System

You need to follow the steps below to run the system:

1. Start your system with `java start`
2. Read the System.properties file.
3. Read the command file and execute the command.
4. Print termination message after all the commands are executed and the threads are terminated.

Additional requirements

1. Graceful termination of execution is required to avoid deduction on grade. This includes but not limited to proper elimination of runaway processes. Terminating program before the entire execution is done may suffer deduction on grade.
2. Follow all the formats given in this specification. Your output finger log file should have the same format as the following. We may use a script to check the correctness of each log; therefore, it is important that you follow the exact same format. Deduction may apply for any difference.

Sample output format for finger log file of identifier 0 and 3 (based on system.properties and command files in this specification)

finger0.log	finger3.log
start: 1; succ: 0 start: 2; succ: 0 start: 4; succ: 0 start: 1; succ: 3 start: 2; succ: 3 start: 4; succ: 0 start: 1; succ: 1 start: 2; succ: 3 start: 4; succ: 0 start: 1; succ: 1 start: 2; succ: 3 start: 4; succ: 6 ……	start: 4; succ: 0 start: 5; succ: 0 start: 7; succ: 0 start: 4; succ: 0 start: 5; succ: 0 start: 7; succ: 0 start: 4; succ: 6 start: 5; succ: 6 start: 7; succ: 0 ……

5. Additional Instruction

Execution Script

```
#!/bin/bash
rm *.java;
mv *.tar proj4.tar
tar xvf proj4.tar;
rm *.class;
rm ID_?.???
javac *.java;
java start
cat report.txt | more
cat *.java | more
cat finger*.log | more
```

We will use the above script file to test your project in grading. Therefore, it is important for you to test with the given script before submission. Before you run this script to test your project, make sure you have backed up your source code at some other location. We will not be held responsible if you lose all your source code after you execute this script. This script is meant to be run with just the project tar file and this shell script in that directory.

Report

You are required to include a one-page project report with your submission. It should contain your experience and challenges you faced while doing project #4. You may include any additional comments and suggestions to the TA in this report file. Make sure your report file is named '**report.txt**'. Since we will be using UNIX environment to execute your project, we suggest you use a UNIX friendly text editor like pico/nano, vi etc. to write your report. **Make sure the lines in your report does not exceed**

80 characters per row.

Runaway Processes

Your threads should terminate gracefully. While testing your programs, run-away processes might exist. These run-away processes must be killed after end of each execution. If run-away processes remain on the system for some time and you do not clean them up properly, your access to the department machines could be restricted according to department regulation (refer to computing help in www.cise.ufl.edu for more details). You might want to embed the following script in your code for automated process killing with appropriate modification.

To check your processes running on a UNIX	`ps -u <user-name>`
To kill all Java processes easily in UNIX	`skill java`
To check runaway processes on remote hosts	`ssh <host-name> ps -u <user-name>`
To clean runaway processes on remote hosts	`ssh <host-name> skill java`

Grading Criteria

Correct Implementation/Graceful Termination	70
Proper Exception Handling	15
Nonexistence of Runaway processes upon termination	15
TOTAL	**100**

Submission Guidelines

- Tar all your source code, config.ini and report.txt into a single tar file.
- Name that tar file – proj4.tar
  - on UNIX use `tar cvf proj4.tar <file list to compress and add to archive>`
- Do not keep sub-directories in your tarred structure; keep all files in the same path.
- Test your tar file for successful execution with the provided shell script above.
- Log in to lss.at.ufl.edu portal using your gatorlink username and password
- Go to COP5615 course page and submit your project under assignments section.

Contact

Sungwook Moon(sungwook.ta@gmail.com) is the point of contact for project #4 description.
Piyush Harsh(cop5615@gmail.com) will be the point of contact for project #4 grading.

DOCUMENT MODIFIED ON 3/4/2010 4:56 PM
Author: Sungwook Moon / sungwook.ta@gmail.com
Version: 1.0