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

Project 3

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

1. Introduction

Project Overview

Through the previous projects, you have learnt distributed systems using multi-thread and client/server socket programming. In this project, we will learn a different process communication mechanism, **Java Remote Method Invocation (RMI),** which is an extension of RPC. We will exercise the concept of RMI through an implementation of Chord discussed in Chapters 2 and 5 (Sections 2.2.2 and 5.2.3). Chord is a protocol to support a scalable peer-to-peer (P2P) lookup service for Internet applications. It helps to improve the efficiency and reduce the complexity of P2P applications. Our implementation of Chord will consist of three stages in three projects. This first project is to get you familiar with nested RMI calls. Completing this project, you will learn how RMI works and how Chord can be implemented with the RMI, and you will be ready to go on to the next projects (Projects 4 and 5).

Project Specification

Unless otherwise specified in this document, specifications for project 1 and 2 are valid also for this project.

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

2. Chord

Many P2P systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node has same functionalities. The core operation in the P2P is an efficient lookup mechanism for locating data items. The Chord is developed to suffice such operation. The following description of Chord is an excerpt from the above references. Although we are not implement the whole Chord protocol in this project, the following is a fairly complete illustration of how Chord works, and it will be helpful for our subsequent projects.

Key Assignment

Chord assumes communication in the underlying network is both symmetric and transitive. In Chord, each node in the system has a unique identifier x falling into a name space of [0, N-1], i.e., $0 \le x \le 2^m - 1$, where N = 2^m. The primary operation of the Chord protocol is a lookup operation: given a key, it finds the node onto which the key is mapped. The key is assigned to a node with consistent hashing. Consistent hashing function assigns each node and key an m-bit

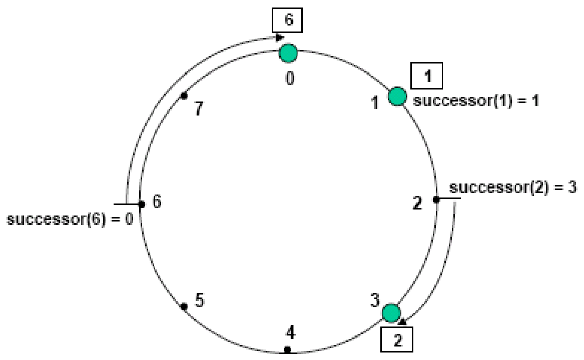


Fig 1. An identifier circle consisting of the three nodes.

identifier using SHA-1 base has function. Node's IP is hashed, and identifiers are ordered on an identifier circle modulo 2^m called a Chord ring. *successor*(k) is the first node whose identifier is equal to or greater than identifier of k in identifier space. ($0 \leq k \leq N-1$) Figure 1 shows an identifier circle with m=3. The circle has three nodes: 0, 1, and 3. The successor of identifier 1 is node 1, so key 1 would be located at node 1. Similarly, key 2 would be located at node 3, and key 6 at node 0.

Chord is designed to let nodes enter and leave the network with minimal disruption. To maintain the mapping when a node *n* joins the network, certain keys previously assigned to *n*'s successor now become assigned to *n*. When node *n* leaves the network, all of its assigned keys are reassigned to *n*'s successor. In the above example, if a node were to join with identifier 7, it would capture the keys with identifier 4, 5, 6, and 7. We will implement the join and leave operations in Project 4.

Routing Table

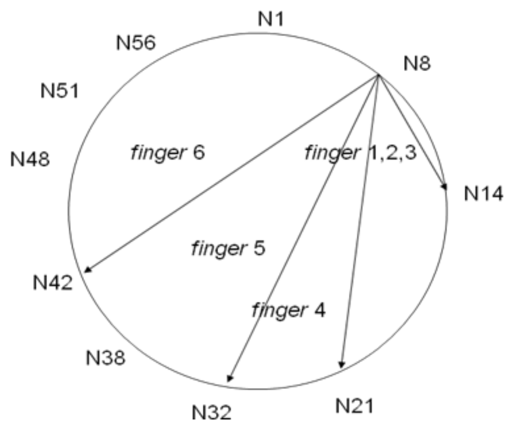
A very small amount of routing information suffices to implement the routing procedure in a distributed environment. Each node need only be aware of its successor node on the circle. Queries for a given identifier can be passed around the circle via these successor pointers until they first encounter a node that succeeds the identifier; this is the node the query maps to. A portion of the Chord protocol maintains these successor pointers, thus ensuring the all lookups are resolved correctly. However this resolution scheme is inefficient: it may require traversing all N nodes to find the appropriate mapping. To accelerate this process, Chord maintains additional routing information.

Notation	Definition
finger[*k*].*start*	$(n+2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
.*interval*	[*finger*[*k*].*start*, *finger*[*k*+1].*start*)
.*node*	first node $\geq$ *n*.*finger*[*k*].*start*
successor	the next node on the identifier circle; *finger*[1].*node*
predecessor	the previous node on the identifier circle

Table 1. Definition of variables for node n, using m-bit identifiers.

As before, let *m* be the number of bits in the key/node identifiers. Each node, *n*, maintains a routing table with *m* entries, called the finger table. The i^{th} entry in the table at node *n* contains the identity of the first node, *s*, that succeeds *n* by at least 2^{i-1} on the identifier circle, i.e., $s = successor(n+2^{i-1})$, where $1 \leq i \leq m$ (and all arithmetic is modulo 2^m). We call node *s* the i^{th} finger of node *n*, and denote it by *n*.*finger*[i].*node*. A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Note that the first finger of *n* is its immediate successor on the circle; for convenience we often refer to it as the successor rather than the first finger. Table.1 summaries the notation and definition.

Fig.2 shows how to figure out successor in finger table when m is 6. *finger*[k] (or *finger k* in the figure) has the first node that succeeds ($n + 2^{k-1}$) mode 2^m as its successor. Therefore *finger 1*, *2* and *3* point to the same N14.



Finger Table for N8

N8+1	N14
N8+2	N14
N8+4	N14
N8+8	N21
N8+16	N32
N8+32	N42

Fig 2. Scalable lookup scheme and finger table

Lookup

What happens when a node n does not know the successor of a key *k*? If *n* can find a node whose ID is closer than its own to k, that node will know more about the identifier circle in the region of *k* than *n* does. Thus *n* searches its finger table for the node whose ID most immediately precedes *k*, and asks *j* for the node it knows whose ID is closest to *k*. By repeating this process, *n* learns about nodes with IDs closer and closer to *k*. The pseudocode that implements the search process is shown below. The notation *n.foo*() stands for the function *foo*() being invoked (by request message through TCP/IP) at and executed on node n.

```
// ask node n to find the successor of id
n.find_successor(id)
    if (id belongs to (n, successor])
        return successor;
    else
        n0 = closest preceding node(id);
        return n0.find_successor(id);
// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
    for i = m downto 1
        if (finger[i] belongs to (n, id))
            return finger[i];
    return n;
```

find_successor works by finding the desired identifier among its finger range (n, successor]. It returns the successor when it finds, otherwise closest preceding node, because the preceding node's successor is what it wants. Thus the algorithm makes progress towards the predecessor of *id*.

3. RMI Implementation

RMI Registry

RMI registry is a Naming Service that acts like a broker. RMI servers register their objects with the RMI registry to make them publicly available. RMI clients look up the registry to locate an object they are interested in and then obtain a reference to that object in order to use its remote methods. Of course, servers register only their remote objects, which have remote methods. Details using the registry are given in the next section.

The Remote Interface

Remote methods that will be available to clients over the network should be declared. This is accomplished by defining a remote interface. This interface must extend *java.rmi.Remote* class and must be defined as public to make it accessible to clients. The body of this interface definition consists of only the declaration of remote methods. The declaration is nothing more than the signatures of methods namely method name, parameters and return type. Each method declaration must also declare *java.rmi.RemoteException* as the throws part.

Necessary Java packages: *java.rmi.Remote*

Implementation of Remote Interface

After defining the remote interface, you need to implement this interface. Implementing the interface means writing the actual code that will form the bodies of your remote methods. For this purpose, you define a class from which you create your remote objects that will serve the clients. This class must extend *RemoteObject* provided by the *java.rmi.server* package. There is also a subclass *UnicastRemoteObject* which is also provided by the same package that provide sufficient basic functionality for our purpose. When you call its constructor, necessary steps are taken for you so that you will not need to deal with them. An RMI server registers its remote objects by using *bind()* or *rebind()* method of *Naming* class. The latter is preferable even for the first time, since you can run your system as many times as you want without touching the registry.

Necessary Java packages: *java.rmi.**, *java.rmi.server.**

RMI Clients

RMI clients are mostly ordinary Java programs. The only difference is that they use the remote interface. As you now know remote interface declares the remote methods to be used. In addition, the clients need to obtain a reference to the remote object, which includes the methods declared in remote interface. The clients obtain this reference by using *lookup()* method of *Naming* class.

Necessary Java packages: *java.rmi.**

4. **Project Description**

This project implements the Lookup function for a static Chord with fixed number of identifiers (or nodes). In the implementation, you do not need the code for sockets (since RMI takes care of all necessary communication) nor the threads to handle Chord functions (since RMI also takes care of threads for us by creating necessary threads at the server). Although you might need to provide synchronization with other functions, but you do not have to worry about it since we only have one function at this time.

System Configuration

Like the previous projects, you are required to make a system configuration file, `system.properties`. `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 as below.

```
Rmiregistry.port=1099
Server=storm.cise.ufl.edu
numberOfNodes=32
numberOfIdentifier=8
client1.id=4
client1.host-name=lin114-01.cise.ufl.edu
client2.id=18
client2.host-name=lin114-02.cise.ufl.edu
client3.id=23
client3.host-name=lin114-03.cise.ufl.edu
client4.id=0
client4.host-name=lin114-04.cise.ufl.edu
client5.id=7
client5.host-name=lin114-05.cise.ufl.edu
client6.id=13
client6.host-name=lin114-06.cise.ufl.edu
client7.id=9
client7.host-name=lin114-07.cise.ufl.edu
client8.id=26
client8.host-name=lin114-08.cise.ufl.edu
```

Chord Server

In this project, a single centralized server manages the Chord. There are several identifiers in the Chord ring, and the server receives Lookup function from them. You should implement this part through RMI. Make sure the server will have other functions in the succeeding projects.

The server has another job for this project. It reads command(s) from ***command*** file. If there is a need to communicate with a certain node, the server uses RMI to communicate with it also. For example, to do the Lookup, the server communicates with a node indicated in the command to let the node invoke Lookup function. Note that you should wait for some time (10 seconds), for all identifiers to be initialized before starting reading commands. The commands will be described shortly.

Chord Identifier(s)

Each identifier is initialized with the id which is defined as a parameter. The identifier will be looking for a key. Please read Section 2 to see how to do that. Note that the procedure of Lookup in this project is recursive. If identifier 1 requested Lookup does not find in its table, it would ask another identifier, i.e., identifier 3, and finally returns the result received from the identifier 3.

When an identifier looks up a key, it needs a finger table as described earlier. In this project, we assume every identifier has its own finger table initially. To get the finger table, you need to know other identifiers, which are described in system.properties. Then you can establish the finger table with notation and description in Table 1.

Reading commands

To operate and simulate your Chord system, you need read command(s) from ***command*** file and executes them in the system. The commands consist of several functions, but currently we have two.

(1) Lookup: Node=3, Key=1;

It means that node 3 will be finding key 1. In other words, node 3 tries to find node 1 and the system should return the successor of node 1. The *Node* could be any of nodes in the Chord circle including identifier and non-identifiers. However for simplicity, assume that it is an identifier.

(2) Exit;

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

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

```
Lookup: Node=4, Key=11;
Lookup: Node=0, Key=30;
Lookup: Node=18, Key=9;
Lookup: Node=4, Key=10;
Lookup: Node=7, Key=30;
Lookup: Node=4, Key=28;
Lookup: Node=23, Key=2;
Lookup: Node=4, Key=31;
Exit;
```

start.java Requirements

Here is a list of actions that must be performed inside start.java
1. Read system configuration from system.properties.
2. Start the Chord server thread so that it can start accepting RMI requests.
3. Using Runtime.exec() start remote nodes (identifiers) defined in system.properties file.
4. Make sure you provide correct runtime arguments as they will act as command line parameters for the remote clients on remote machines, provide descriptive identifier Ids such as 4,13,23 etc.

RMI registry

RMI registry can be started on a Unix machine by typing:

```
/usr/local/java/bin/rmiregistry<port>&
```

<port> is the port number we want the registry to listen for connections from RMI servers and clients. This port number is optional and if not specified, registry is started on default port 1099. A host that provides continuous RMI support should use this port number for convenience and standardization. But **we will specify our own port number** by starting our own RMI registry. This port number should match the one in the **system.properties** file. Note that "&" is to run the registry as a background process. Important Note: The full path name above for the RMI commands are necessary in the CISE environment. Otherwise you may end up using some other version of RMI which is not compatible with the JDK.

Compiling programs

Compile your server and client code as usual.
Note: If the server needs to support clients running on pre-5.0 VMs, then a stub class for the remote object implementation class needs to be regenerated using the **rmic** compiler, and that stub class needs to be made available for clients to download.

Running the System

You need to follow the steps below to run the system:
1 Start `rmiregistry` manually as a background process as explained above. Specify an arbitrary port number that is large enough to avoid collisions with the well-known port numbers. This port number should match the one in the `system.properties` file. Use *ps* to make sure the rmiregistry is running.
2 Start your system with `java start`
Check the outputs and processes running and repeat this step as many times as you need to test your system. Even after your server thread is completed, the server process, which makes the remote object available, will keep running. To terminate the process, you should use `System.exit()` method (in the server thread) immediately after the server thread finishes counting the specified number of accesses. Note that, the server thread is very simple and its job is to keep track of the number of accesses made.
3 Terminate the rmiregistry using *kill*, manually.

Note on Registering and Looking up Remote Objects

As we now know, the server needs to register its remote object to the RMI registry to make its own remote methods available to the clients. In turn, clients need to look up the RMI registry to obtain a reference to that object and then to call its remote methods. You must use your CISE login name as the name of the service for registering the remote object to the registry. Similarly the clients must look up the object by giving your CISE login name. The reason for this restriction is to prevent possible name collisions by making the names unique, since we will use the same RMI registry to grade all the projects.

Sample Execution Scenario

Consider the Chord ring in Fig 3. Suppose node 3 wants to find the successor of identifier 1. Since 1 belongs to the circular interval [7,3), it belongs to 3. *finger*[3].*interval*; node 3 therefore checks the third entry in its finger table, which is 0. Because 0 precedes 1, node 3 will ask node 0 to find the successor of 1. In turn, node 0 will infer from its finger table that 1's successor is the node 1itself, and return node 1 to node 3.

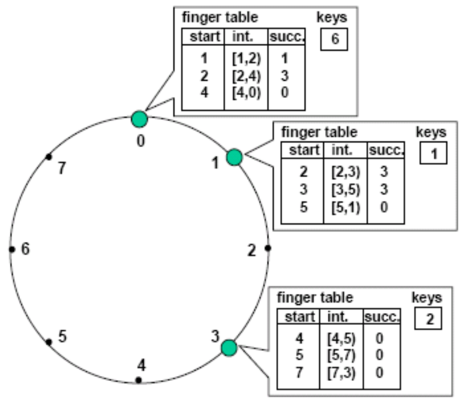


Fig 3. Finger tables and key locations for a net with nodes 9, 1, and 3, and keys 1, 2, and 6.

Sample Output Format for identifier 4(based on system.properties and command files)

```
              query4.log                                  finger4.log
  4 Lookup 11: routing path 4->9->13          start: 5; interval: [5,6); succ:7
  4 Lookup 10: routing path 4->9->13          start: 6; interval: [6,8); succ:7
  4 Lookup 28: routing path 4->23->26->0      start: 8; interval: [8,12); succ:9
  4 Lookup 31: routing path 4->23->26->0      start: 12; interval: [12,20); succ:13
  Complete                                    start: 20; interval: [20,4); succ:23
```

5. <u>Additional Instruction</u>

Execution Script

```
#!/bin/bash
rm *.java;
mv *.tar proj3.tar
tar xvf proj3.tar; rm *.class
javac *.java;
java start
cat report.??? | more
cat ID_#.??? | more
```

Before you use this script file 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 our script. This script is meant to be run with just the project tar file and this shell script located 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 project3. 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. However these should be killed frequently. Since the department has a policy on this matter, your access to the department machines might be restricted if you do not clean these processes.

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 – proj3.tar
  - on UNIX use `tar cvf proj3.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.
- Non existence of sub-directories is crucial for successful execution of your code with our script
- 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.

DOCUMENT MODIFIED ON 2/8/2010
Author: Jae Woong Lee / jwlee@cise.ufl.edu
Version: 1.0