COMP 415/515 – Distributed Computing Systems

Assignment-3

Due: April 3, 2019, 11:59 pm (Late submissions are not accepted.)

Submit your assignment deliverables to the folder: **F:\COURSES\GRADS\COMP515\HOMEWORK\A3**

Note: This is a group-oriented assignment, you are encouraged to form groups of 2, and we suggest a fair task distribution at the end of this assignment description. Working individually is allowed but not recommended.

Distributed NoteBook Using Chord DHT

This assignment is about the **Distributed Hash Tables (DHTs)** as a specific type of structured peer-to-peer (P2P) system architectures in distributed platforms. It involves distributed software development using the concepts of **DHTs, structured P2P systems**, **Remote Method Invocation (RMI)**, **threads, and virtualization**. As the distributed platform, **Amazon Web Service Elastic Compute Cloud (AWS EC2)** would be used to deploy the Distributed NoteBook to be developed, measurements to be performed and the results tol be reported.

You are asked to design and implement a Distributed NoteBook Service Using Chord DHT. Essentials:

- Distributed NoteBook service is composed of peers each acting as a node in a Chord DHT.
- Peers are responsible for the maintenance of the Distributed NoteBook.
- Each page of the Distributed NoteBook is represented by a single (virtual) Chord node.
- You need to use RMI for communication among the peers (i.e., processes). Each process should support multiple **concurrent RMI** for other processes.
- As clarified later, your implementation should support peer dynamics in arrivals and departures.
- Each peer would be able to create a note and the result shall be accessible by other peers in a consistent manner.

Overview:

Figure 1 illustrates an overview of Distributed NoteBook with 8 processes. ID of each process is computed as the hash of its IP address and Port number. Each note to be stored on DHT is represented as a (key, value) pair. The value of a note is the concatenation of subject and body of the note. The key of a note is the hash of note's subject. As an example, detailed content of a

note is shown in Figure 1. Process ID_3 is assumed to be the successor of this note on the Chord ring and hence is responsible to hold it.

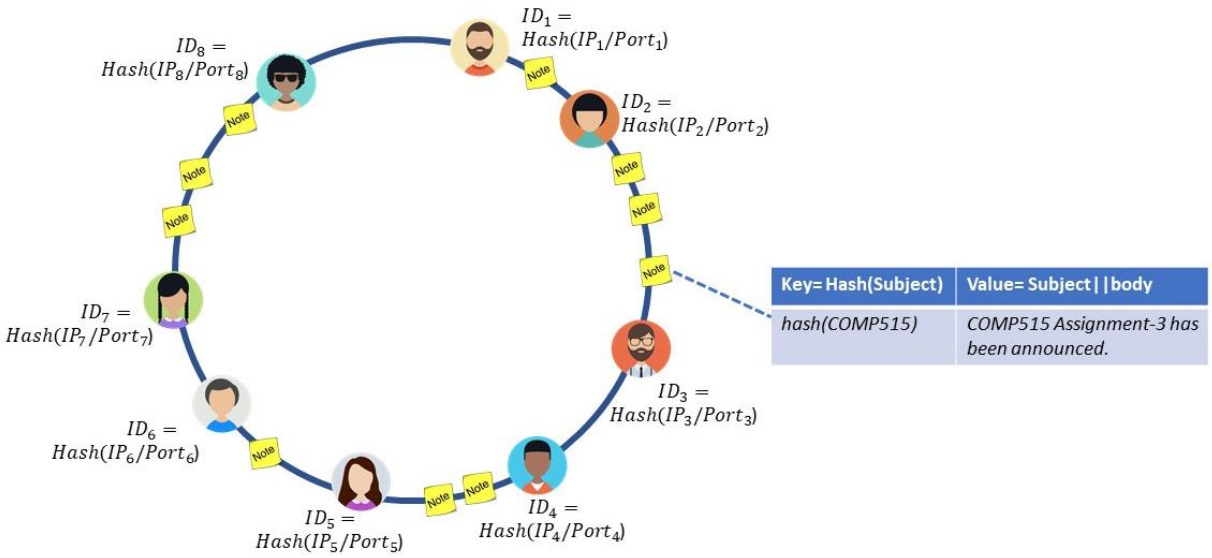

Figure 1- The overview of the distributed notebook. A sample note is shown inside the table (this note resides in the process with ID_3 as its successor).

Part1- Distributed NoteBook Process

Unique Identifier:

A Distributed NoteBook Process represents a participating peer in the system. Each process in the system is identified by the hash value of the concatenation of its IP address and port number. For example, a Distributed NoteBook Process that is executed on a machine with IP address of 1.2.3.4 and the port number of 1111 has the identifier of hash(1.2.3.4/1111). You are free to choose a standard hash function the (e.g., SHA-3 or MD5) to compute the hash values.

Joining the Chord:

The processes should form a Chord DHT. Following this, each new process should know one of the existing processes that has already joint the Chord overlay, which is called the new process's **introducer.** The IP and port number of the introducer of a process should be given at the startup by the user. The introducer then performs a lookup for the new process, finds its successor on the overlay, and returns its address to the new process. Having the address of its successor, the new process should join the Chord as described in the textbook. Also, the new process should make an RMI call to the **Join** function of its successor. As the result of this call, the successor will transfer the notes for which the new process is responsible based on the notes keys.

Lookup:

Lookup on the Chord should be implemented by recursive RMI calls to the **Lookup** function. Each call is executed on a process, and based on the lookup target either is handled internally, or is forwarded to another process of the system. Forwarding a lookup to another peer is done by invoking the Lookup function on that peer using an RMI call. Once the lookup is terminated, the IP address and port number of the lookup result should be circled back to the lookup's initiator.

Posting a note:

A process receives a note from the user and posts it to the system. A note should have two parts a subject and a body. Following is the example of a note:

Subject: COMP515, Body: Assignment-3 has been announced.

You should implement a Note class to represent a note object. To post a note to the system, the (poster) process computes the key of the note as the hash value of its subject, performs a lookup for its key, and finds the process that is responsible for keeping the key, i.e., the successor process of the key on the Chord. The poster then contacts the responsible process to post the note. The post operation is done by invoking the **Post** function of the responsible process by an RMI call. The responsible process then stores the note as a (key, value) pair in its own memory, where the value corresponds to the subject||body of the note (|| indicates concatenation). The previous example of a note is stored as:

(hash(COMP515), COMP515 Assignment-3 has been announced.)

In case of repeated keys, the old (key, value) pair should be updated as

(key, subject||old body||new body)

Considering the previous example, if the responsible process **already holds a note as**

(hash(COMP515), COMP515 Assignment-2 deadline is over.)

upon posting the new message with the same key as hash(COMP515), it should update its note as

(hash(COMP515), COMP515 Assignment-2 deadline is over. Assignment-3 has been announced.)

As an alternative of reading the notes from the console, you may store the notes associated with each process in an input.txt file on its directory. Each note as is stored as a (subject, body) pair on a separate line in the input file. Once all the processes joined, each user posts its batch of notes by writing *post file* to the console, which makes the process to read the notes one-by-one from the input file, and post them in the same order into the DHT. As we discuss it later, this alternative approach especially helps you for experimenting over AWS.

Retrieving a note:

The responsible process for a note should be identified via a lookup for the note's key. Once the responsible process's address is determined, the body of the note is retrievable via an RMI call to the **Get** function on the responsible node. On invoking the Get function on a node using a key, the node should check for the existence of a (key, value) pair corresponding to the input key of interest. If such a pair exists, the node should return it to the caller process, otherwise, it should return NULL. For example, assuming that the responsible process for the key hash(COMP515) receives an RMI call on Get(hash(COMP515)), it should return the value corresponding to the key that it holds to the caller, i.e., *COMP515 Assignment-2 deadline is over. Assignment-3 has been announced.*

Leaving the Chord:

A process may leave the Chord overlay anytime by informing its predecessor and successor as well as transferring all the notes (i.e., (key, value) pairs) it maintains to its successor. This should be done via an RMI call to the **Leave** function on both predecessor and successor. The Leave invocation on successor should also take the list of all the notes (to be transferred to the successor) as input, while the same input for the predecessor should be left empty (i.e., NULL).

Interactions with user:

- On startup, the process should ask for the IP and port of the introducer from the user and join the Chord.
- After joining the Chord, the process should continuously show a menu for "Getting a note", "Posting a note", and "Departing the system" operations to the user, ask user for her choice.
    - Upon choosing the "Getting a note", the process should ask for the subject of the note, retrieve it from the Chord, and show the body of the retrieved message to the user. In case such a subject does not exist, the process should show a proper message to the user.
    - Upon choosing the "Posting a note", the process should ask for the subject and body of the note, and perform the posting operation, and show the result.
    - Upon choosing the "Departing the system", the process should perform leaving the chord operation as specified, and show a proper message once the departure is done successfully.

Showing the state of a process:

A process should print the followings upon the specified events:

- Printing the finger table upon its join as well as each time its finger table gets updated (by other processes' arrival or departure)

- Printing the lookup target key as well as the processes that have been visited for this lookup, each time the Lookup function of the process is getting (RMI) called.
- Printing the poster of a note as well as its note content, each time the Post function of the process is getting (RMI) called.
- Printing the requester of a note as well as the result that is returned back to the requester, each time the Get function of the process is getting called.

Part2- Deployment on the AWS EC2

- Create virtual machines (VMs) in AWS where each VM should reside on a distinct AWS data center e.g., one in Virginia, the other one in Beijing, etc.
- Deploy and run your processes into the VMs.
- Test the correct execution of your system for all the listed operations in part-1. Take a screenshot of each scenario and insert it into your report (as specified later in this description).
- Examine your system with 8 and 16 processes (VMs). For each system size, perform 10 tests on lookup operation at each node and record the time each lookup concludes. In your report, plot a graph with the X-axis as the system size, and Y-axis the average lookup operation time in ms.
- Following the previous experiment, also measure the number of processes a lookup visits on the average. Plot a graph with the X-axis as the system size, and Y-axis the average number of processes a lookup visits. Considering that lookups are visiting c * log n many steps, approximate c based on your results (n is the number of processes in your system).

Bonus: P2P-based Sockets

As you also experienced in Assignment-2, the traditional way of implementing P2P processes is to dedicate separate Server Sockets and ordinary Sockets for each process. Additionally, each process should also handle concurrent communications via threading. As the bonus, you are asked to investigate other ways of implementing P2P-based sockets, which do not require separate sockets for client and server part of your peer, and handle concurrent communications by themselves. For the bonus to be considered accomplished, your code should be implemented with the proper alternative in a way that only one socket object can handle the both sides' communications for a process without the need for any multi-threading. Note that you still need to enable RMI functionality on your solution. In other words, the valid solution should not replace the RMI with message passing.

Report:

The assignment report should contain the step-by-step description of the following in such a way that anybody who reads the report could do the exact configuration without using any external references. Hence, you are strongly recommended to use screenshots and explain your answers by referring to the screenshots. In the report, you should provide captions, figure numbers, and referral to the figures as we did in this assignment description.

- How did you implement the Note data structure?
- How did you implement the finger tables of the processes?
- In case you are doing the bonus, you should also provide a clear step-by-step description of your approach inline with the report criterion.

Deliverables:

You should submit your source code and assignment report (in a single .rar or .zip file). The name of the file should be <last name of group members>.

- Source Code: A .zip or (not .rar) file that contains your implementation in a single Eclipse, IntelliJ IDEA, or PyCharm. If you aim to implement your assignment in any IDE rather than the mentioned one, you should first consult with TA and get confirmation.

- The **report** is an **important part of your assignment**, which should be submitted as both a .pdf and Word file. The **report acts as proof of work to assert your contributions to this assignment.** Anybody who reads your report should be able to reproduce the parts we asked to document. If you need to put the code in your report, segment it as small as possible (i.e., just the parts you need to explain) and clarify each segment before heading to the next segment. For codes, you should be taking a **screenshot** instead of copy/pasting the direct code. Strictly avoid **replicating the code** as whole in the report, or leaving code **unexplained**.

Demonstration:

You are required to demonstrate the execution of your Distributed Notebook on AWS with the defined requirements. The demo sessions would be announced by the TA. Attending the demo session is required for your assignment to be graded. **All group members** must attend the demo session. **The on-time attendance of all group members is considered as a grading criterion.**

Important Notes:

- **Please read this assignment document carefully BEFORE starting your design and implementation. Take the report part seriously and write your report as accurate and complete as possible.**
- Your entire assignment should be implemented in Java, C++ or Python. Implementation in network-based languages (e.g., Ruby) is not allowed. For implementation in other languages, you should first consult with TA and get confirmation.
- In case you use some code parts from the Internet, you must provide the references in your report (such as web links) and explicitly define the parts that you used.
- You should **not** share your code or ideas with other assignment groups. Please be aware of the KU Statement on Academic Honesty.
- For the demonstration, you are supposed to bring your own laptop and run your program on your own laptop. Bring more laptops if you need more than one.
- Your entire code should be well-commented. If you are programming in Java, you are required to provide JavaDoc as well. If you are implementing in any other language, you are required to add the general description of each function, its inputs, and outputs.
  You may use the following reference to get ideas about writing a clear and neat JavaDoc.: https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html
  Also, you may use the following reference to get ideas about commenting your codes properly: https://improvingsoftware.com/2011/06/27/5-best-practices-for-commenting-your-code/

Suggested task distribution:

We recommend you to work in a group of 2, and suggest the following task distribution.

Student 1: Finger table management, join, leave of processes, and AWS deployment.

Student 2: Notes posts, management, and retrievals, AWS measurements and report.

Please be informed that despite the task distribution, we expect all the group members to learn the project concepts entirely.

References:

- Structured Peer-to-Peer Systems (In Section 2.3 and Note 2.5 of the textbook)
- Distributed Hash Tables, Chord (In Section 5.2 of the textbook)
- Threads and Virtualization (Section 3.1 and 3.2 of the textbook)
- Multithreading in Java [Link] , AWS EC2 [Link]

Good Luck!