COMP 415/515 : Distributed Computing Systems

Assignment-1

Due: February 19, 2020, 11:59 pm (Late submissions are not accepted.)

This is an individual assignment. You are not allowed to share your codes with each other.

Development of Centralized/Decentralized KU-Store on AWS EC2

This assignment is about **centralized** and **decentralized system architectures** in distributed platforms. It involves application layer software development using the **client/server model, TCP socket programming, threads (Goroutines)** and **virtualization**. **Go Language** and **Amazon Web Service Elastic Compute Cloud (AWS EC2)** as the distributed platform would be used to deploy the systems, and the results would be reported.

You are asked to design and implement KU-Store (a simplified system resembling Dropbox) for file storage and retrieval. In part-1, a centralized (e.g., similar to PS 1) KU-store and in part-2, a decentralized KU-store (e.g., a torrent like system) would be developed and compared. In both systems, a client connects to the KU-Store system to store and retrieve its own files.

Part 1: Centralized KU-Store (20 points)

This part is very similar to the PS-1 client-server example. Your task is to implement a multi-threaded central server which can store and transfer files. You should have two files "task1-server.go" and "task1-client.go". Your client program should take 2 arguments: server IP and server port, while the server will only take the port argument.

Example:

./task1-server 9090 (The server will start running on port 9090)
./task1-client ip-of-server 9090

Your client program should display the following options on the terminal when it starts:

1) Enter the username:
2) Enter the filename to store:
3) Enter the filename to retrieve:
4) Exit:

>Please select an option:

Example:

>Please select an option: 1

>Enter the username: waris
>**Server Response**: Login Successful
>Please select an option: 3
>Enter the filename to retrieve: ds-1.pdf
>**Server Response:** File does not exist.
>Please select an option: 2
>Enter the filename to store: ds-1.pdf
>**Server Response**: ds-1.pdf stored successfully.
>Please select an option: 3
>Enter the filename to retrieve: ds-1.pdf
>**Server Response**: ds-1.pdf found.
(File will be downloaded in the current directory of the client program)

Important: A user cannot retrieve the files of another user. For example, if user 1 stores two files (e.g., ds-1.pdf and ds-2.pdf) on the server then user 2 cannot retrieve them. If the user2 tries to retrieve, your program should display "**>Server Response:** File does not exist" on the terminal window. The command-line arguments are a must.

Part 2: Decentralized KU-Store (70 points)

Motivation: Suppose that your central server contains the world's largest hard disk which has a capacity of 20 TB (Link). A user on a Dropbox can store up to 3 TB of data (Link). So, if we allow the same plan on KU-Store server, then the current system can only handle a maximum of 6 (6 x 3 = 18 TB) clients. Assume that you are given thousands of nodes to make the system scalable. How will you decide which node store which files? Even if you solve this issue, there is another problem. How will the user know on which computer she/he can store files? Thus, there is a clear problem of scalability and your Part-1 KU-Storage service is not scalable. To address this issue, in this part your task is to develop the following simple solution to make KU-Store scalable.

Assigning Identifiers: Let say you have 5 storage peers, nodes or computers which are p1, p2, p3, p4 and p5. In the first step, you will assign each computer an ID based on its IP and Port number. Now you will assign each peer an identifier (**Peer-ID**) based on its IP address and port number by taking the hash of the IP address and the port number.

Example:

peer 1: **hash("172.3.4.5:9090") = 0**
peer 2**: hash("172.3.4.6:9090") = 2**
Peer 3: **hash("172.3.4.7:7070") = 5**
Peer 4: **hash("172.3.4.8:9090") = 6**
Peer 5: **hash("172.3.4.9:9080") = 11**

Peer Connections: You will organize these peers (i.e., make connections between them) in **a logical ring,** such that the **successor** of a peer is the first node met going in clockwise direction and **predecessor** is the first node met in the counter-clockwise direction. You will make the connections between these peers as shown in Fig.1.

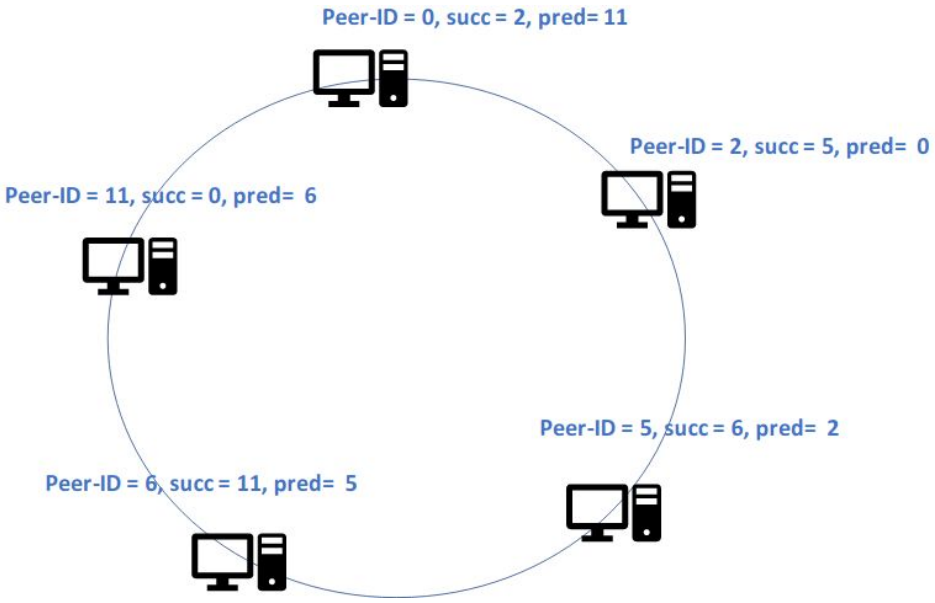

Figure 1: Formation of ring based on successor and predecessor id.

Storing Files: Use the same hash function and take the hash of the filename to associate a **file-ID** with it. Example:

hash(“ds-1.pdf”) = 12
hash(“ds-2.pdf”) = 1
hash(“go-in-one-lesson.go”) = 9
hash(“hello-world.txt”) = 14
hash(“ds-3.pdf”) = 4

Now store each file based on its **file-ID** on its successor. For instance, consider the above ring, the node with Peer-ID 2 will store **ds-2.pdf**. Similarly, the **node 0** will store **hello-world.txt** and **ds1-pdf**.

Lookup: To find the successor of a peer or a file based on their Peer-ID and File-ID use the following successor lookup algorithm (pseudo code):

```
func findSuccessor(id)
    if predecessor != nil and id ∈ (predecessor, n])
        return (my-id, my-ip, my-port)
    else if id ∈ (n, successor])
        return (successor-id, successor-ip, successor-port)
    else // forward the query to the successor of the node
        return successor => findSuccessor(id)

//(a,b] => the seqment of the circle include b but not a.
// e.g., (2, 7] = {3,4,5,6,7}
```

Joining the Ring: In order to join the ring, a peer x will first search for its successor (i.e., hash("ip:port")). Let say the successor of **x**, is the peer **y**. After finding the successor, the peer x will execute the following steps:

- Peer x sets its successor to peer **y.**
- Peer x asks peer y for its predecessor and set own predecessor to peer y predecessor.
- Peer x will notify peer y to set its predecessor to x.
- Peer x will notify peer y predecessor to set its successor to x.
- The files from peer y, whose successor is peer x, will move to peer x.

Example: The joining of a node in a ring is shown in Fig.2.

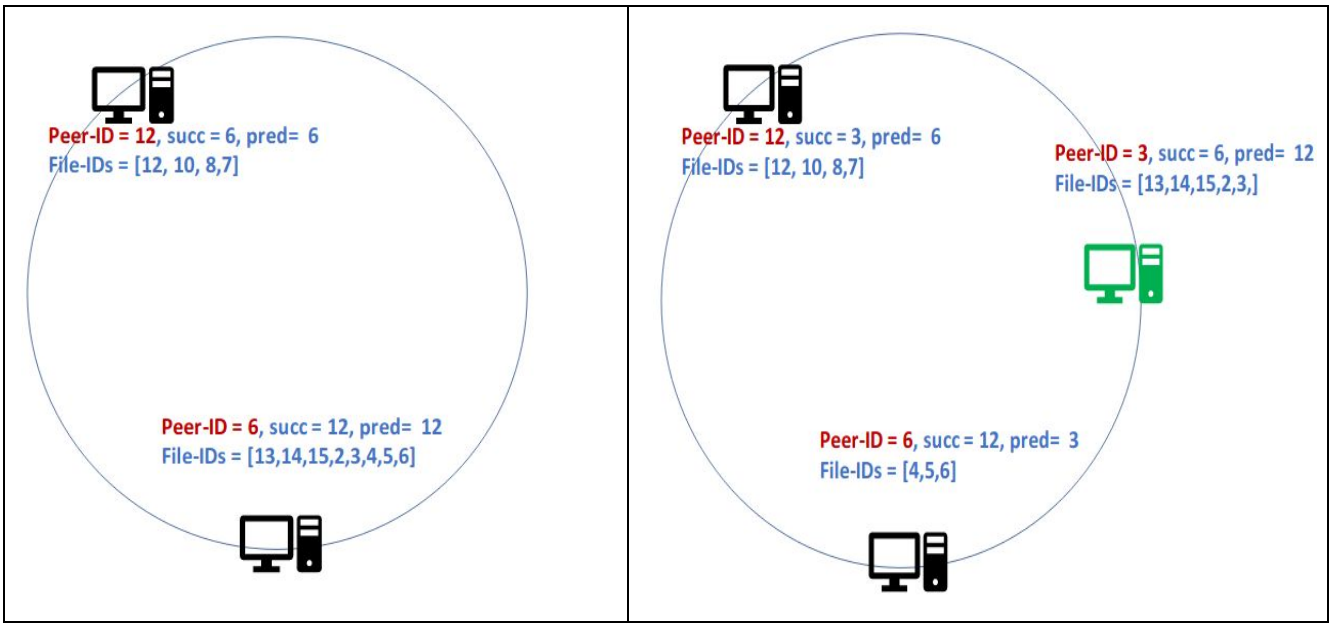

Figure 2: A new node with Peer-ID = 3 joined the ring.

Leaving the Ring: Similarly, when a peer wants to leave the system, it notifies its successor and predecessor. Then, it transfers all the files to its successor. An example is shown in Fig.3.

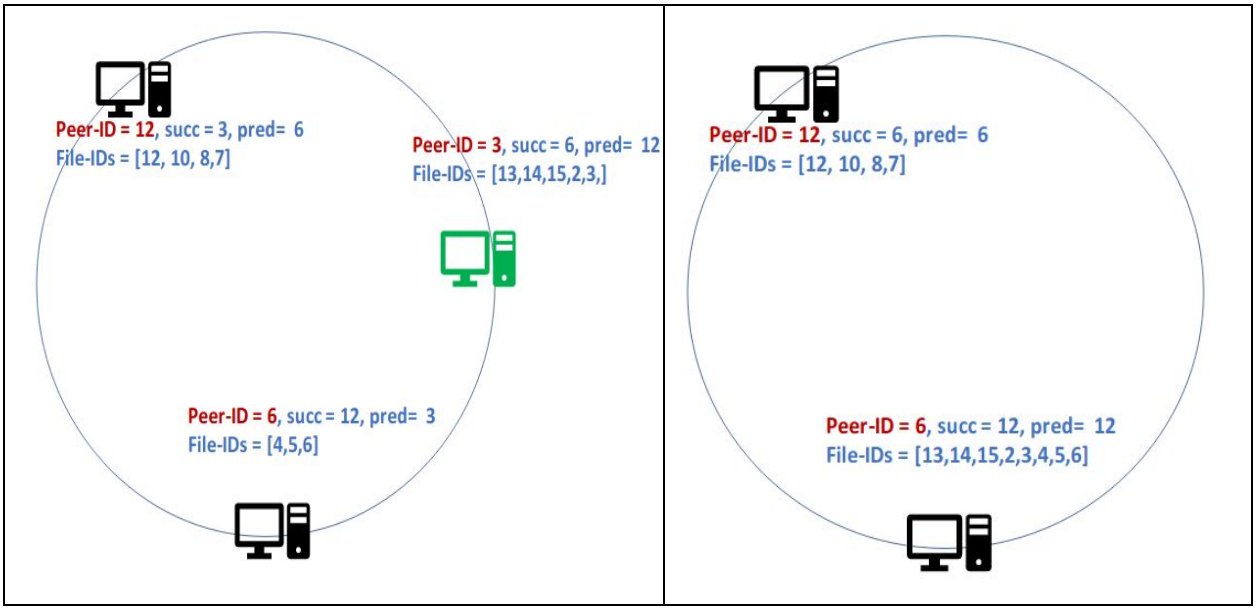

Figure 3: Node with Pee-ID 3 left the system.

Important: In the start, the ring will have only 1 node and you can set its predecessor or successor to nil or to the node itself. This choice is up to you, and you have to make findSuccessor(id) method consistent based on your choice otherwise you the message will start looping in the ring.

You should have two files "task2-peer.go" and "task2-client.go". Your client program should take two arguments: any storage peer's IP address and its port number, while the storage peer will only take the port argument to start.

Example:

./task2-peer 9090 (The server will start running on port 9090)
./task2-client ip-of-any-peer 9090

Your peer program should display the following options on the terminal when it starts:

1) Enter the peer address to connect:
2) Enter the key to find its successor:
3) Enter the filename to take its hash:
4) Display my-id, succ-id, and pred-id:
5) Display the stored filenames and their keys:
6) Exit.

>Please select an option:

Example:

>Enter the peer address to connect: 172.4.4.4:9090
>(Response): Connection Established
>Enter the key to find its successor: 77
>(Response): (succ-id, succ-ip, succ-port)
……

Your client program should display the following options when it runs:

1) Enter the filename to store:
2) Enter the filename to retrieve:
3) Exit:
>Please select an option:

Example:

>Please select an option: 2
>Enter the filename to retrieve: ds-1.pdf
>**Server Response:** File does not exist.
>Please select an option: 1
>Enter the filename to store: ds-1.pdf
>**Server Response**: ds-1.pdf stored successfully.
>Please select an option: 2
>Enter the filename to retrieve: ds-1.pdf
>**Server Response**: ds-1.pdf found.
(File will be downloaded in the current directory of the client program)

Report (10 points)

- Create virtual machines (VMs) in AWS. Deploy and run your processes into the VMs.
- Test the correct execution of your system both in part-1 and part-2.
- Examine your system with 8 processes (VMs) for part-2.
- Measure the average response time (time to store or retrieve files) of part-1 and part-2 systems and show your results in a bar graph (x-axis contains the name of the storage system, while y-axis represents the average response time). You should store and retrieve files of different sizes (e.g., 1KB, 2KB, 3KB and 4KB) and take the average of their storing or retrieving times.
- **Question:** How many nodes are traversed by findSuccessor(id) to find the successor if a system contains maximum of N nodes? Please answer this question in Big-O notation. Suggest different ways to reduce the number of traversed nodes.

Submission and Demonstration:

Please submit your assignment including report and Go source files on the Blackboard (lastname_KU_id.zip). You are required to demonstrate the execution of your KU-Store 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.

Important: 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.

Good Luck!