Assignment 3:
Peer-Peer File Sharing System

Deadline: _____May 8_____ at 11:55 PM

The key purpose of this assignment is to familiarize you with the concept of **consistent hashing**, that is widely used in distributed systems. ***You will be building a peer-peer file sharing system that uses Distributed Hash Table, leverages consistent hashing and is resilient to node failures.***

This assignment is to be done individually. You need to pass all the test cases to obtain full credits.

Note: You should post any assignment related query ONLY on CampusWire. Do not directly email the course staff.

Note: Course policy about plagiarism is as follows:

- This assignment should be done individually.
- Students must not share actual program code with other students.
- Students must be prepared to explain any program code they submit.
- Students must indicate with their submission any assistance received.
- All submissions are subject to plagiarism detection. We will run MOSS on your assignment and compare with both online solutions as well as solutions from previous years.
- Students cannot copy the code from the Internet. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.
- **Late day policy:** You have a total pool of 5 days for late submissions across all the assignments, without deductions.

Introduction

In this assignment, you will build a peer-peer file sharing system. Your system will be a Distributed Hash Table (DHT) based key-value storage system that has only two basic operations; **put()** and **get()**. put() operation takes as input a filename, evaluates its key using some hash function, and places the file on one of the nodes in the distributed system. Whereas, get() function takes as input a filename, finds the appropriate node that can potentially have that file and returns the file if it exists on that node. We will make this system failure tolerant i.e. it should not lose any data in case of failure of a node.

You will be building the DHT using consistent hashing. Consistent hashing is a scheme for distributing key-value pairs, such that distribution does not depend upon number of nodes in the DHT. This allows to scale the size of DHT easily i.e. addition or removal of new nodes in the system is not a costly operation.

Background

Begin by reviewing Lecture 18 & Lecture 19 slides on LMS. You should also watch the video recording of these lectures on YouTube. For extra resources, you are welcome to look at the following articles:

- Consistent Hashing - System Design Blog

  https://medium.com/system-design-blog/consistent-hashing-b9134c8a9062

- A Guide to Consistent Hashing

  https://www.toptal.com/big-data/consistent-hashing

These articles go in a little extra detail of having virtual nodes for better load balancing. Since, you would not be implementing that in this assignment, you can ignore it.

Provided code

Download & Setup

- Download the assignment files from the LMS, you will have two files: DHT.py and check.py.
- DHT.py is the file you will be writing all of your implementation. We have already provided some starter code to help you get started. You should not change any of the starter code.
- DHT.py is for testing, you are welcome to look at it, but you should not change anything in this file.

DHT.py Explanation:

We will test your assignment using the following API.

1. **Node()** : Initialization of Node, you may maintain any state you need here. Following variables which are already declared should not be renamed and should be set appropriately since testing will take place through them. Some of them have already been set to appropriate values.

    a. host: Save hostname of Node e.g. localhost

    b. port: Save port of Node e.g. 8000

c. M: hash value's number of bits

d. N: Size of the ring

e. key: it is the hashed value of Node

f. successor: Next node's address (host, port)

g. predecessor: Previous node's address (host, port)

h. files: Files mapped to this node

i. backUpFiles: Files saved as back up on this node

2. **join()** :

This function handles the logic for a new node joining the DHT: this function should update node's successor and predecessor. It should also trigger the update of affected nodes' (i.e. successor node and predecessor node) successor and predecessor too. Finally, the node should also get its share of files.

3. **leave()** :

Calling leave should have the following effects: removal of node from the ring and transfer of files to the appropriate node.

4. **put()** :

This function should handle placement of file on appropriate node; save the files in the directory for the node.

5. **get()** :

This function is responsible for looking up and getting a file from the network.

Some functions have already been provided to you as utility functions. You can also create more helper functions.

1. **hasher()** :

Hashes a string to an M bits long number. You should use it as follows:

i. For a node: hasher(node.host+str(node.port))

ii. For a file: hasher(filename)

2. **listener()** :

   This function listens for new incoming connections. For every new connection, it spins a new thread handleConnection to handle that connection. You may write any necessary logic for any connection there.

3. **sendFile()** :

   You can use this function to send a file over the socket to another node.

4. **receiveFile()** :

   This function can be used to receive files over the socket from other nodes. Both these functions have the following arguments:

   a. soc: A TCP socket object.

   b. fileName: File's name along with complete path e.g. "CS382/PA3/file.py"

Implementation instructions

Although assignment can be done in any order, we suggest you follow the following pattern as this order of checking is followed in testing as well.

Note: You are required to use ***sockets for any communication between the nodes*** i.e. you should *not* access another node's variables or functions directly.

1- Initialization: *1 mark*

When a new node is created, its successor and predecessor do not point to any other node as it does not know any other node yet. In this case both these pointers should point to the node itself. *successor* and *predecessor* both store a tuple of host and port e.g. ("localhost", 20007).

2 - Join: *9 marks*

Before you implement join, put or get; it is a good idea to implement a look-up function. This look-up function should be able to find the node responsible for a given key. Benefit of doing this is that you can reuse this function in join, put and get.

Let's come to *join* now. Join function takes as input the address (host, port) of a node (joiningAddr) already present in the DHT. You will connect with this node and ask it to do a lookup for the new

node's successor and update the new node's successor based on the response. You should also write the logic of updating the predecessor. This usually happens through pinging; a node keeps pinging (after around **0.5 second**) its successor to know whether it is still its successor's predecessor. If the successor node has updated its predecessor, this node should update its successor to the current successor's predecessor. The joining process is explained through the figures below.

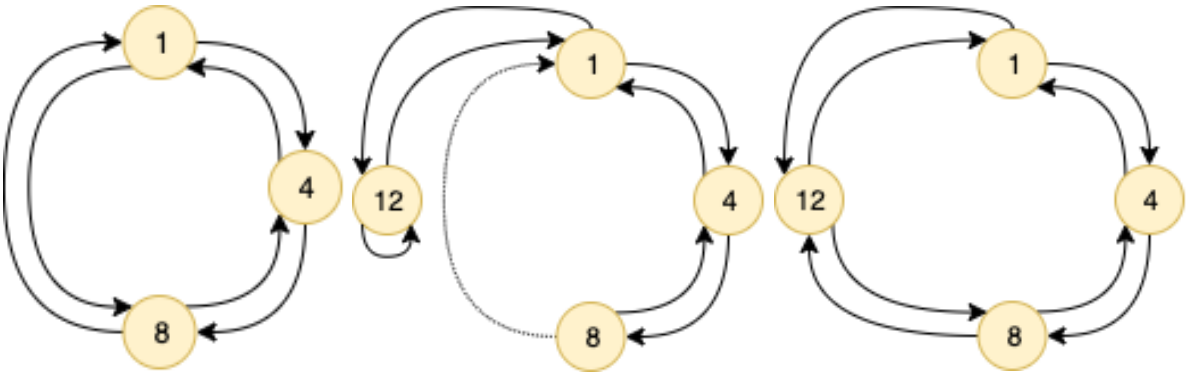

Figure 1: Left to right: ***i)*** A DHT with three nodes, clockwise arrows are for successors and anti-clockwise are for predecessors. ***ii)*** A new node 12 is added, it finds its position and updates its successor to point at 1, 1 points its predecessor at 12. ***iii)*** 8 asks 1 for its predecessor and gets 12, it updates its successor to 12, 12 updates its predecessor to 8

You should pay special attention to corner cases such as when there is only one node in the system; in this case, instead of the joining address, an empty string will be passed. Similarly, when there are only two nodes, they will be each other's successor and predecessor.

3 - Put and Get: *10 marks*

When *put* is called on a node with a file, it finds the node responsible for that file and sends the file to that node. You can use sendFile and receiveFile functions here. You should store the file in the directory assigned to that node given by node's host and port as host_port e.g. localhost_20007. This directly is already made in the provided starter code.

Similarly, the *get* method should again first find the node responsible for the file and then retrieve the file from that node. Again, you can use sendFile and receiveFile functions. This time you should

get the file from the node and save it in the current directory. If the file exists you should return the filename, if it does not you should return None.

Every node is responsible for the space between itself and its predecessor, See the following figure for a better understanding.

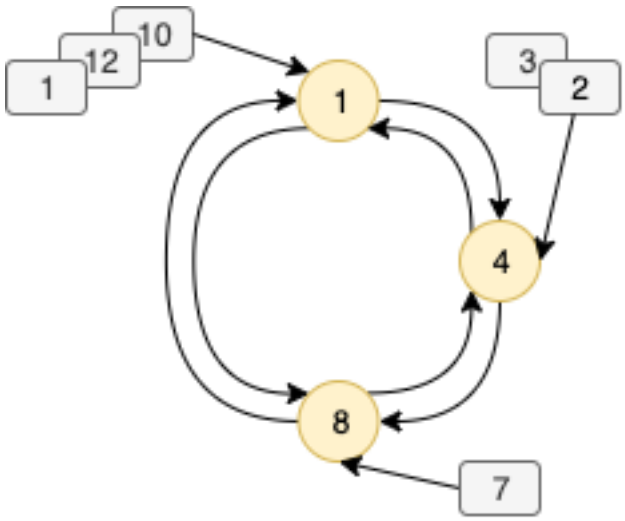

Figure 2: Placement of files on a DHT, every node is responsible for space between itself and its predecessor

4 - File Transfer on Join: *5 marks*

When a new node joins the network, it should get its share of files i.e. the files that hash to its space. Think about what file will come in share of a new node. After transferring the file, the new node will be responsible for all these files and the previous responsible node should delete these files from its list.

5 - Leave: *5 marks*

When a node leaves the DHT, it should do so gracefully. This means it should tell its predecessor that it is leaving, communicate with the predecessor, the address of its successor node so the predecessor can update its successor. Moreover, leaving node should also send all its files to the new responsible node. Think about which node will be responsible for these files now.

6 - Failure Tolerance: *10 marks*

As we talked earlier, we need to make our system tolerant to node failures. Failures are commonplace in real life systems, in fact, most designs are made keeping failures in mind as a requirement. We also want our DHT to be resilient to failures. Firstly, you should be able to detect that a node is down, typically this is done by pinging a node. Since we are already pinging the successor, we can check if the successor does not respond for 3 consecutive pings, this means the successor node is down. You will need to keep some state in advance to account for failure. For example, to maintain the ring structure of DHT, every node should keep a list of successors instead of immediate successor. In case the immediate successor is detected to have failed, we can update our immediate successor to the next successor in our list. You may just keep the state of the second successor instead of keeping a list of successors for this assignment. Next you want to replicate all the files so that even if a node fails, no data is lost. Think about where you should replicate files? At the predecessor of the node? At the successor? At the start of the DHT.py file mention where you replicated files and argue why.

Running and Testing

Test cases are run in the order described here. You may not be able to test one part before completing its previous parts as most of the parts have previous parts as their pre-requisite. You will need Python3 to run this file, you can run the tests on any OS but Linux preferable. Use the following command to run the tests:

```
python3 check.py <port>
```

You should pass a port between 1000 and 65500. If you start getting an error like:

```
error: [Errno 48] Address already in use
```

Just choose a different port number with a significant gap. Or alternatively, restart the terminal.

Submission

You only need to submit the DHT.py file. Rename it with your student ID as: [Student ID]_DHT.py e.g. 22100212_ DHT.py.

Grading

We will run the check.py file with your solution to mark your assignment.

You can earn up to 40 points from this assignment.

The following table describes each test case along with its maximum points. You can get partial points based on the following criteria:

Test Description	Maximum Points
Test 1: Initialization	1
Test 2: Join a) Corner case of 1 node b) Corner case of 2 nodes c) General case	 1 3 5
Test 3: Put and Get a) Put Test b) Get Test	 7 3
Test 4: File Transfer on Join	5
Test 5: Leave a) Updating successor and predecessors of affected nodes b) Files transferred correctly	 3 2
Test 5: Failure Tolerance a) Updating successor and predecessors of affected nodes b) Files recovered on correct nodes	 3 7

As always, start early and feel free to ask questions on CampusWire and in office hours.