CSCI 4220 Assignment 4

SimpleKad DHT Implementation

Due Date: Friday, December 6th, 11:59:59 PM

Your task for this team-based (max. 2) assignment is to implement a distributed hash table (DHT) that is similar to Kademlia. **Keep in mind that we are not making an exact implementation so some details in the paper/online resources may not apply**. For example, we will not be using replication (i.e. $\alpha = 1$), there is no PING command, and there is no replacement cache. Since this is a peer-to-peer application, there is no "server" versus "client" program. You will have to write a single program that can support both tasks.

It is recommended that you start early, both so that you can make sure your understanding of the protocol is clear and so that you have enough time to think about the data structures and code organization that you want to employ for this assignment. Refer to Lecture 12 notes for a gRPC primer, and Lecture 13 notes for more details on Kademila.

Your implementation will be done in Python using gRPC and will be autograded. The protobuf file is provided under Course Materials as csci4220_hw4.proto and we will provide the generated csci4220_hw4_pb2.py and csci4220_hw4_pb2_grpc.py during autograding (so you do not need to submit them). The autograder will probably ignore any .proto files you submit, so write your solution using the provided .proto file. The autograder expects one Python script which it will run, named *hw4.py*.

We will run your program by doing the following `python3 hw4.py <nodeID> <port> <k>`

Just like in Homework 3 we will be inputting hostnames instead of IP addresses. To save you a few includes, we have provided hw4_starter.py under Course Materials which shows how to get your local hostname and IP address and resolve a remote hostname into an IP address.

Upon starting up a node, it should do any initialization that is necessary and then start listening on the `port` that was provided as a command line argument. It will use the value `k` for the size of its k-buckets. The server handler will run in the background (as it does in all the gRPC examples we looked at), meanwhile the program should read from standard input where it can expect the commands listed below.

Printing k-buckets

When printing k-buckets use the following format for i=[0,N) where N is the number of bits in the IDs:
`<i> [<oldest entry> <next oldest entry> ... <newest entry>]<newline>`
where an entry is in the form
`<ID>:<port>`

BOOTSTRAP <remote hostname> <remote port>

This command lets a node connect to another node by exchanging information so that both nodes know each other's ID, address, and port. This is done by sending the remote node a FindNode RPC using the local node's ID as the argument. At the end of the command, the node should print to standard output `After BOOTSTRAP(<remoteID>), k_buckets now look like:` followed by printing the k-buckets (see below for formatting). `<remoteID>` should be replaced with the ID of the remote node. Both nodes should add each other to their k-buckets.

FIND_NODE <nodeID>

This command attempts to find a remote node, and has the side effect of updating the current node's k-buckets. If the node's ID is `<nodeID>`, then no search should be made and the node can skip directly to the post-search output, and should behave as though it found the node. The search works as follows:

```
While some of the k closest nodes to <nodeID> have not been asked:
  S = the k closest IDs to <nodeID>
  S' = nodes in S that have not been contacted yet
  For node in S':
    R = node.FindNode(<nodeID>)
    Update k-buckets with node
    Update k-buckets with all nodes in R
  If <nodeID> has been found, stop
```

Distance is found by taking the XOR of the two IDs, with a lower distance meaning the two nodes are closer. The node should ask one remote node, wait for a response, and then repeat the procedure; the queries are serial. Whenever a node receives a FindNode request, it should return the k closest nodes to the provided ID. The responder may need to look in several k-buckets to fulfill the request.

Remember that a k-bucket can only hold up to k entries. Also recall that a node will have distance 0 from itself. Finally, remember that during a FIND_NODE command, a node should not make a FindNode RPC to the same remote node more than once.

On the requesting side, when a FIND_NODE command is read from standard input, the program should print:

```
Before FIND_NODE command, k-buckets are:
<kbuckets>
```

After the FIND_NODE command completes (remember, the FIND_NODE command may invoke the FindNode RPC several times before finishing) the program should print:

```
After FIND_NODE command, k-buckets are:
<kbuckets>
```

Lastly if the target node ID was found, the program should then print:

```
Found destination id <nodeID>
```

Otherwise the program should print:

```
Could not find destination id <nodeID>
```

If a program receives a FindNode request, it should print:

```
Serving FindNode(<targetID>) request for <requesterID>
```

FIND_VALUE <key>

This behaves the same way that FIND_NODE does, but uses the key instead of a node ID to determine which node to query next, and the FindValue RPC instead of the FindNode RPC. If the remote node has not been told to store the key, it will reply with the k closest nodes to the key. If the remote node has been told to store the key before, then it does not return a list of nodes, and instead responds with the key and the associated value. The rules for updating k-buckets are the same as in FIND_NODE. The search stops as soon as the key/value is found, even if the node has not queries all k-closest peers in the current round.

On the requesting side, when a FIND_VALUE command is read from standard input, the program should print:

```
Before FIND_VALUE command, k-buckets are:
<kbuckets>
```

After the FIND_VALUE command completes (remember, the FIND_VALUE command may invoke the FindNode RPC several times before finishing) the program should print:

```
After FIND_VALUE command, k-buckets are:
<kbuckets>
```

Lastly if the target key was found, the program should then print:

```
Found value "<value>" for key <key>
```

Otherwise the program should print:

```
Could not find key <key>
```

If a program receives a FindValue request, it should print:

```
Serving FindKey(<key>) request for <requesterID>
```

STORE <key> <value>

The node should send a Store RPC to the **single** node that has ID closest to the key. Keep in mind that the current node may be the closest node and may need to store the key/value pair locally. For simplicty, values will never have spaces in them, but may be otherwise-arbitrary strings.

When calling the Store RPC, the requester should print:

```
Storing key <key> at node <remoteID>
```

If a node receives a call to Store, it should print:

```
Storing key <key> value "<value>"
```

QUIT

The node should send a Quit RPC to each node that is in its k-buckets. Before making a call to node `<remoteID>`, the node should print:

```
Letting <remoteID> know I'm quitting.
```

If a node receives a call to Quit from `<remoteID>`, and the remote node is in k-bucket , the entry should be removed from the k-bucket and the following printed:

```
Evicting quitting node <remoteID> from bucket <i>
```

Otherwise the node should print the following:

```
No record of quitting node <remoteID> in k-buckets.
```

Finally, the node should print the following message to `stdout`, where `<ID>` is its ID, and then terminate:

```
Shut down node <ID>
```