CSCI 4220 Assignment 4

DHT Implementation

Due Date: Friday, December 7th, 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**. Since this is a peer-to-peer application, there is no “server” versus “client” program, your program will have to 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. We will use lab time on Friday November 16th for you to work on the assignment and ask questions as a class. Refer to Lecture 21 notes for more details.

Submitty will not be autograding this time around, so you should submit a `Makefile` that produces a `dht.out` along with your code.

To start a node from the command line, assuming the compiled program is ./dht.out use
`./dht.out <nodeName> <port> <nodeIDseed> <k>`
In this example, nodeName can be an IP address (local testing/TA testing) or a string (if we were autograding).

The application will use UDP. Upon startup the application should immediately start listening on the provided port using UDP. Input should also be accepted from the command line. The current command will always be processed to completion before the next command can start. The following commands are to be supported:

`CONNECT <node name> <port>`
This command lets a node connect to another node by exchanging information so that both sides have a complete 3-tuple to insert into the appropriate k-bucket. The node should send a message to the remote node on the specified port of the following format:
`HELLO <sender name> <sender node ID>`
Anytime a node receives a `HELLO` message, it should respond with a `MYID` message:
`MYID <receiver node ID>`
and then the receiver of the MYID message can store the (name, port, ID) in the appropriate k-bucket.

`FIND_NODE <node ID>`
The node should send a message to the closest known node to the provided node ID. Distance is found by taking the XOR of the two IDs, with a lower distance meaning the two nodes are closer. The format is identical to the standard input command above, FIND_NODE . The search terminates after either finding the target node or asking k closest nodes. The node should ask one remote node, wait for a response, and then repeat the procedure, in other words the queries are serial. Whenever a node receives a FIND_NODE request, it should return the k closest nodes to the provided ID. If there are not enough nodes in the bucket, then nodes from other k-buckets should also be returned. Each returned entry should be in the form:
`NODE <remote node name> <remote port> <remote ID>`
If a node recieves a NODE command, it should add the node to the appropriate k-bucket. Remember that a k-bucket can only hold up to k entries. Also recall that a node will have distance 0 from itself.

`FIND_DATA <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. The syntax when communicating messages to remote nodes is the same as the FIND_DATA command provided to standard input. If the remote node has not been told to store the key, it will reply with the k closest nodes to the key, `NODE <remote node name> <remote port> <remote receiver>` . 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:
`VALUE <node id> <key> <value>`
If a node receives a VALUE command, it should print out `Received [value] from [remote ID]` where [value] is the value that was associated with the key, and [remote ID] is the ID of the node that sent the VALUE command.

`STORE <key> <value>`
The node should send a `STORE <key> <value>` message to the node with the lowest distance between the remote node's ID and the key. If a node receives a STORE message (through the network, not a command through stdin), then it should store the key-value pair locally. For simplicity, values will always be integers.

`QUIT`
The node should message all nodes in its k-buckets with:
`QUIT <node ID>`
to notify them that it is quitting. It should then close any open sockets/descriptors and then exit. A node that receives a QUIT message should remove the entry associated with that node ID from its buckets, if it exists.

We will not be using replication (e.g. $\alpha = 1$), there is no PING command, there are no RPC IDs, and there is no replacement cache. Any hexadecimal input will use lower case letters (0-9a-f).

We will use a node ID length and key length of 8-bits, a default value of k=3, and for simplicity will use SHA-1. An implementation exists in C in the OpenSSL development libraries (i.e. `sudo apt install libopenssl-dev`, and then add `-lcrypto` to the end of your compile line). To generate a node's ID, start with the passed in "node ID seed", S. Assume that S only contains digits (but it doesn't matter, you should leave it as a char*). The first byte of the SHA-1 digest of S is the node's ID.

If a node received a message from a remote node, the remote node should be added as the most recent entry in the appropriate k-bucket. This may result in a node being evicted from the k-bucket.

Since we are using UDP, we will use a timeout of 3 seconds. If a message should receive a response but doesn't receive one within 3 seconds, assume the remote node has quit (remove it from your k-bucket), abort the command, and print an error.

Output should be written to `stdout` and formatted as follows:

1. Whenever we know the remote ID and we receive a message, use this format. (In this case the message comes from 2a and is `FIND_NODE 5f`:
   `<2a FIND_NODE 5f`
2. Whenever we don't know the remote ID and we receive a message, we should print it, and then send a `HELLO` message to the remote node to resolve IDs:
   `<? FIND_NODE 5f`
3. When sending a message, use this format. (In this case we are writing to node 2a with a message of `NODE 127.0.0.1 138 5f`:
   `>2a NODE 127.0.0.1 138 5f`
4. If a timeout happens, use this format. (In this case there was a timeout sending to node 3c with a message of `QUIT 2a`):
   !3c QUIT 2a
5. Do not write any other output to stdout.