Homework 2
Yuyi Qu
Session: CS6675
Problem 1. Hand-on Experience with a Peer to Peer System
Option 1

Introduction

In this homework, I experimented with a distributed hash table, called OpenDHT (https://github.com/savoirfairelinux/opendht). OpenDHT is a Kademlia implementation of a distributed hash table. OpenDHT offers a map from a distributed shared key to a value store. The key is of 160 bits and the value can be arbitrary binary up to 64KiB. It can differentiate different values with the same key with the help of a key-unique ID. I experimented with OpenDHT through running nodes in python, in c++, and also in the command line. I tested different peer discovery approaches supported by OpenDHT. I experimented with both synchronous and asynchronous implementation of a node for some basic operations like put and get. I collected the time statistics needed for the get operation when varying the number of nodes in the network.

Peer Discovery

A DHT node can join an existing network by pinging other nodes already in the network, which can either be a known bootstrap node like bootstrap.jami.net or be some known address like localhost:39695. Figure 1. shows joining a network through bootstrap.jami.net, and Figure 2 shows joining a network through a known address. As shown in Figure 2, at first the DHT node running on port 39695 is not in any network, therefore the get operation fails. And then another DHT node running on port 46464 pings this node and joins its network, and puts some key-value pair into the network. And then the DHT node running on 39695 can successfully perform the get operation and retrieve the value from the service on port 46464.

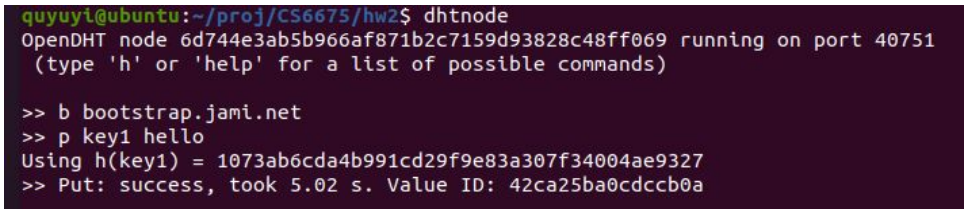

Figure 1. Join an Existing Network through Bootstrap.

```
quyuyi@ubuntu:~/proj/CS6675/hw2$ dhtnode
OpenDHT node d26c6f3a26a6339553fc4e020047a1c5bc2394d3 running on port 39695
 (type 'h' or 'help' for a list of possible commands)

>> g key1
Using h(key1) = 1073ab6cda4b991cd29f9e83a307f34004ae9327
>> Get: failure, took 387 us

>> g key1
Using h(key1) = 1073ab6cda4b991cd29f9e83a307f34004ae9327
>> Get: found 1 value(s) after 212 us
        Value[id:561e9c55eed5fcf8 data(text/plain):"hellofrom46464"]
Get: completed, took 771 us
```

```
quyuyi@ubuntu:~/proj/CS6675/hw2$ dhtnode
OpenDHT node 32e787cfa762553b4a978c6b69ef5641b2783898 running on port 46464
 (type 'h' or 'help' for a list of possible commands)

>> b localhost:39695
>> p key1 hellofrom46464
Using h(key1) = 1073ab6cda4b991cd29f9e83a307f34004ae9327
>> Put: success, took 890 us. Value ID: 561e9c55eed5fcf8
```

Figure 2. Join an Existing Network through a Known Address.

If no known node can be pinged to join a network, OpenDHT offers a peer discovery service to both advertise a node's own service and discover other nodes' services. Figure 3 shows that DHT node running on 4222 advertises its service and DHT node running on 4223 also advertises its service. And then it starts discovery. Figure 4 shows that result of discovery that it discovered both service on 4222 and 4223.

```
// peer discovery
// A single instance & port can be used for both advertising and discovering of multiple different services
dht::PeerDiscovery discovery(MULTICAST_PORT);
ExampleServiceData my_data {4222, 1, "1"};
discovery.startPublish(MY_SERVICE_NAME, my_data);

dht::PeerDiscovery discovery2(MULTICAST_PORT);
ExampleServiceData my_data2 {4223, 2, "1"};
discovery2.startPublish(MY_SERVICE_NAME, my_data2);

discovery.startDiscovery<ExampleServiceData>(MY_SERVICE_NAME,
[](ExampleServiceData&& d, dht::SockAddr&& addr) {
    addr.setPort(d.service_port);
    std::cout << "Discovered some peer for our service at " << addr.toString() << std::endl;
});
```

Figure 3. Peer Discovery Implementation

```
quyuyi@ubuntu:~/proj/CS6675/hw2$ ./sync
Discovered some peer for our service at 192.168.217.4:4222
Discovered some peer for our service at [fe80::6c2d:2cd7:cfbd:4ba%ens33]:4223
Discovered some peer for our service at 192.168.217.4:4223
Discovered some peer for our service at [fe80::6c2d:2cd7:cfbd:4ba%ens33]:4222
```

Figure 4. Join a Network through Peer Discovery

Synchronous and Asynchronous

DHT requests are asynchronous. I can specify two callback functions. One is triggered when a portion of the results are returned and I can decide whether I need to continue to get more values. And the other is triggered when the get operation is complete and I can check the status of the operation. When I specify an operation for the DHT node, the two callback functions are

put into the queue of the DHT node thread. I have to explicitly block the process in order to check the result of some operation. I can achieve this by using mutex. Let the conditional variable wait where blocking is needed and notify the conditional variable in the callback function which is triggered when the operation is finished. Figure 1. shows an implementation for a synchronous DHT node in C++.

```cpp
/*** Sync heler function ***/
auto wait = [=] {
    *ready = true;
    std::unique_lock<std::mutex> lk(*mtx);
    cv->wait(lk);
    *ready = false;
};
auto done_cb = [=](bool success) {
    if (success) {
        std::cout << "success!" << std::endl;
    } else {
        std::cout << "failed..." << std::endl;
    }
    std::unique_lock<std::mutex> lk(*mtx);
    cv->wait(lk, [=]{ return *ready; });
    cv->notify_one();
};
/*** End sync heler function ***/
```

```cpp
// get data from the dht
node.get("unique_key",
[](const std::vector<std::shared_ptr<dht::Value>>& values) {
    // Callback called when values are found
    for (const auto& value : values)
        std::cout << "Found value: " << *value << std::endl;
    return true; // return false to stop the search
},
[=] (bool success) {
    std::cout << "Get: ";
    done_cb(success);
});
wait();
```

Figure 1. Synchronous Implementation of a DHT Node in C++

Performance Measurement for “get”

When I perform the operation with the same key one the same node, the time used for the first operation is long while the time becomes small for the following repeated operations. This is probably because the most time used when performing a put or get operation is the time for hashing the key, which is mathematically heavy.

```
quyuyi@ubuntu:~/proj/CS6675/hw2$ python3 run.py
node1 put with key0, time elapsed: 4.8046488761901855
node1 put with key1, time elapsed: 6.371002435684204
node1 put with key2, time elapsed: 5.822587013244629
node1 put with key3, time elapsed: 4.899983882904053
node1 put with key4, time elapsed: 5.126538038253784
node1 put with key5, time elapsed: 2.309572219848633
node1 put with key6, time elapsed: 3.7739641666412354
node1 put with key7, time elapsed: 5.0000293254852295
node1 put with key8, time elapsed: 4.188786029815674
node1 put with key9, time elapsed: 4.700290679931641
node2 get with key0, time elapsed: 3.6732540130615234
node2 get with key1, time elapsed: 7.51068377494812
node2 get with key2, time elapsed: 5.546885251998901
node2 get with key3, time elapsed: 4.883469343185425
node2 get with key4, time elapsed: 7.662014961242676
node2 get with key5, time elapsed: 2.408334493637085
node2 get with key6, time elapsed: 6.736317873001099
node2 get with key7, time elapsed: 5.357922792434692
node2 get with key8, time elapsed: 5.192080736160278
node2 get with key9, time elapsed: 5.033796310424805
```

Figure 2. Performance of Get and Put when Different Key is Used

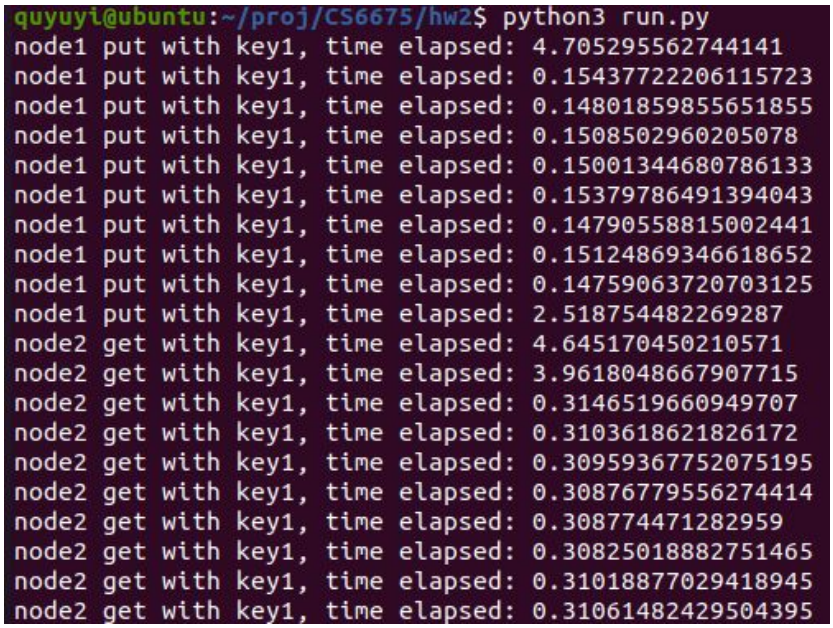

Figure 3. Performance of Get and Put when the Same Key is Used

In Figure 2, different keys are used, as we can see, the time used for each operation is about 4 seconds on average. In Figure 3, the same key is used, as we can see, the node takes a long time when it executes the operation with the specific key at the first time, and then the time drops significantly in the following operations with that same key. I think it is because OpenDHT will cache the key hash for some amount of time in case the same key is used shortly after. Another thing we can notice from Figure 3 is that the average time for the get operation is larger than the put operation when using the cached key for the operation. This is because there are a lot of values under the same key, and the get operation with that key will retrieve all the values under that key, which make it longer. This implies that if there are a lot of hash conflicts, the performance will be affected. However, it is negligible when the hashing process dominates the time usage.