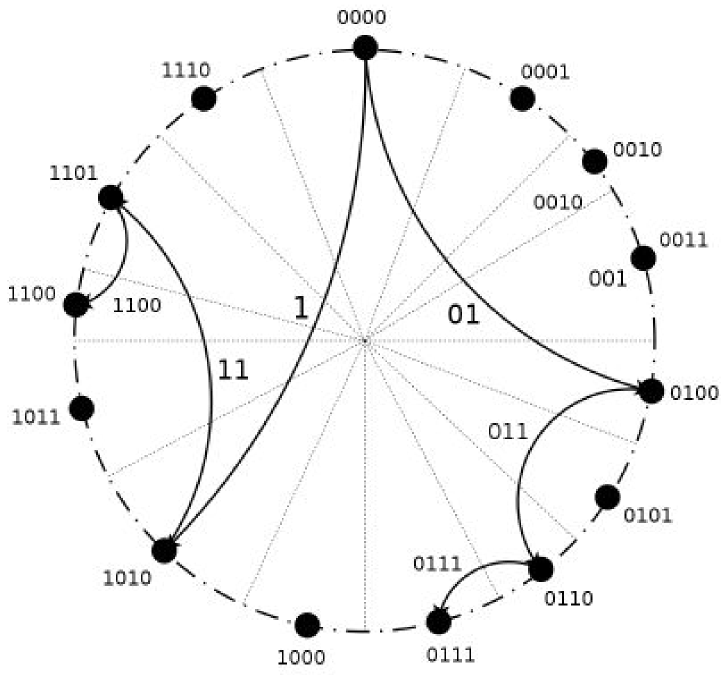
Operating System - CSE 403

Distributed Hash Table/ Replicate Hash Table

A report of challenges, implementations and the design

*Second project description is added in section 4,5 and 6 (from page 7)

Amin Hosseiny Marani



1 Introduction

In this report I will describe the design I had in mind, the reason I chose the design and theory behind the model. In implementation part I will go through ideas that I had first, those which ended up in failure, the reasons I thought of that failures and also the alternatives. Unfortunately, my implementation has not reach to the final successful point, and I stock at communication part. I described the steps I took and failed me, and the reason I took them. I used Python for implementation, because of its consistency and portability over different operating systems and also because I have been working with it lately.

2 DHT design model

2-1 Transfer Protocol

For a Distributed Hash Table which is running over multiple nodes, the first crucial thing is handling the communication and connection between nodes. No matter, we want to see which nodes are available at the start, searching for a key, or sending a key to other nodes, in all such operations we need to connect two nodes and transfer messages. In the early design I decided to choose *UDP* for transfer protocol. UDP does not check for packet lost and it is up to senders to check if their packets are receiving by listeners or not.

Missing control over sending packets, surely is a disadvantage, especially when we are sending a key to another server and waiting for another specific node to return its value or a message that the desired node stored the (key,value) pairs. However, it is not the whole problem. Sending and receiving thousands of packets, and checking all of them to make sure if they are received or handled well, will surely decrease network speed, while we could check if the corresponding message for previous send one is received or not and decide to resend it. UDP protocol surely will increase the speed, since it does not check every packets and message to be received or not, but we should make sure we get back the message for the desired packet we sent earlier.

2-2 Broadcasting nodes to find each other

A distributed Hash Table network must be capable of adding nodes whenever possible. And we also should consider early in the beginning no nodes know about others. In such case every node which is new can broadcast a message and show it is available to contact other nodes. However, such an assumption is ideal and we were told we could consider network with fixed number of nodes and without any crash. So I decided to use broadcast to show other nodes that i-th node is available by sending a message. While a node receives a message from other nodes, it pushes the node’s address and its key related value to a list for later connection and also searching which we will discuss in the next subsection. In this case each node start broadcasting to other nodes until they get a reply message of each node that they receive the broadcast message. This feature will help us to start a DHT without knowing other nodes public IP. Later in the next session, I will describe why this plan fails and why I decided to use a simple UDP connection to tell every other node that the sender is ready.

2-3 finding other nodes

The main feature of DHT is being distributed, so every node must be able to receive a key or key,value pair and check if it could respond to it or should send it to other nodes. When a request such get(key) receives to a node, the node performs a search over all node's key value to see which node is closer to the requested key. Bear in mind, we store each node key at first stage when they are communicating to see if they are ready or not.

A simple to find closest node's key to the requested one, will show us which node is a probable target of this request. We must send a message and ask the target node to search, so the target node gets the message and check if the index of hash value is existed or not. It might be out of target node, too. In this case the node will resend the message to the nearest node for further search. For put(key,value) a similar thing will happen, too. We search for the hash table of current key on the closest node to this key and continues it till we find the spot.

2-4 Manage lost packets

It is common to lose some packets Using UDP connections. We cannot afford to lose a request, since the availability of the system is important in such distributed models. As we get a request from a user (such as get or put) we will find the target node and if it is different than current node, we will send a message consisting of request, key and probably the value for put function. We then don't want to remove the request in send/receive queue but to monitor the status. After we send a message for that request, we don't change the status of message in sending list, till we get a message related to that.

Only when we get the message and returning the result we could delete the request out of sending list.

2-5 other notes

We should assume the size of network can grow, so we start with a small number like one hundred or one thousand and when we reach the half of the size, we double it. It may not be the most efficient method, but there is a guarantee we never run out of space, since we will check the size everytime we store a new key,value pairs. It is not efficient since when we get to a really big size like 1 GB of data, doubling the storage capacity could be only space wasting and other smarter method that considers previous history of requests can solve the issue, but for now we can skip that. Another important key note here is, when we used Python list we can always append the list and add data at the end of it without considering the current size or future size. So basically this issue won't affect our model, but it could affect our searching method, while we are looking for the closest node to a specific key.

Needless to say, we should devise a way to lock the bucket for both get/put operations if there is an update. So we make sure the get(key) is exactly returning the current value. Luckily there is no update for put(key,value) and we either store it or return false as the project description says, so we could skip this part, too.

3 Implementation and Challenges

In this section, I will talk about each part in the coding phase and see the challenges and possible failures. I also talk about alternative methods I chose and the reason behind.

3-1 Using Amazon EC2 instances

In order to run the whole system distributed, I chose Linux distribution (micro with pre-installed Python) over Amazon EC2 which is free. I launched three instances and copied all instances public IP in one file. We later see, I failed to get each node's IP using broadcasting and then decided to put all public IP in one file and share it with all instances manually, so they can easily send their messages to other instances.

3-2 Broadcasting failed, UDP connection replaced

Usually we can broadcast packets over network by using below codes. But when I used it over Amazon EC2 instances, none of the nodes were able to catch the packets. I also set the permission for inbound/outbound security group to all traffic and all ranges of IPs, but could not figure it out why nodes are not able to receive any packets. The other alternative I considered was using Google API to write a file in Google Drive, so each node can write their address and hash keys to that file. The problem I could not find a way to write directly to a text file, rather to download, write and upload a text file. By uploading and downloading, if two nodes want to change the file at the same time, we might lose one of them since they both download one file but upload different file with different timing and we will lose some nodes.

Another idea was to make a Google Drive directory and ask each node to open a file with address name and hash key inside it. Since it needs special google APIs to be installed and the whole process was time consuming, I shut down the process for later usage, however it could be and idea to expand. Needless to say like any other ideas, this one is not faultless, too. For every separate execution we should clear directory files and we might have some nodes attributes that are dead or not related to this run etc.

After struggling more than one day, I decided to change the first phase (knowing other nodes are available and ready) to a simple UDP communication, over knowing IPs. In the new approach, since each node has a list of all other nodes' address, they just connect and repeatedly send their hash key, till they receive other node's hash key. They won't stop in that case, since one node may get the other node's address but that node has not received its address yet. So, the node continues sending the hash key to all nodes and when it receives a 'ready' message it decides the nodes might get its message, too. For any possible fault tolerant I let the program to receive up to 3 'ready' message.

3-3 simple UDP connection failed, too

In this case, I consider each node to have at least 2 ready messages of each other node to finish sending 'ready' or showing the node is available. Since in UDP protocol we have to check if packets received or not, I devised this method, but the problem is that even with bigger number than 2, there are always one node that can get 'ready' message of all others and don't send the exact amount to those nodes. In this case, selecting an upperline to finish the availability messaging became a simple/unresolved issue for me. I believe replacing UDP connection with TCP can solve my problem, but I haven't spent much time on TCP since I considered speed and I was willing to find a solution for UDP connection, which unfortunately I did not.

As another alternative, I define a function for listening and one for sending messages and then run it using Threading.Thread to run simultaneously. I got an error of using same address can be only used once, and I am struggling to find a way out of this. By now, I can resolve the problem of sending and receiving, I thought maybe I can lock each function and see if I can receive any data and send. But when I try to test that, I lost packages and most of my packages failed to receive by other nodes. I do it repeatedly for a large number of times, it seems after a big number of sending and receiving I am able to receive other node's packets. However, not only it is not trivial, but also I have a problem to cut-off this repeatedly sending packets.

3-4 Conclusion

Unfortunately, the code I attached with the report does not working and I could not manage to solve node's communication. By solving this issue I will be able to contact other nodes and then searching inside other nodes become possible. As a conclusion I can say, considering TCP for connection may reduce such problems and we don't want to fight over, problems in communication. The other thing is, I had better design a function for sending messages and one for receiving (or one for both), if I receive a message that needs a response I could do the operations and send the results and wait for the 'OK' message and then remove the whole request out of the list. And for sending, I should listen on the UDP connection I send message.

4 DHT implementation

Unfortunately, I was not able to finish the DHT project on time, but using XML-RPC I have finished the simple put/get for the first project. As you can see in figure1, a client can remotely connect to a server each time and call a function named put/get and finish the task. By using XML-RPC, server does not care about the type of message, all it has to do is to let a client connect to it and call the remote functions. Each time a client wants to get a key, first it asks for a connection and then call get function using that connection. Server then process the function and return the result. During the connection if a crash/problem appears, the client gets an exception and knows that the get procedure is not completed, so it puts it back to the list for later request. This is pretty much the same for put function, a client calls the function and server takes care of inserting a value into Bucket list or return false/update if there exists that key.

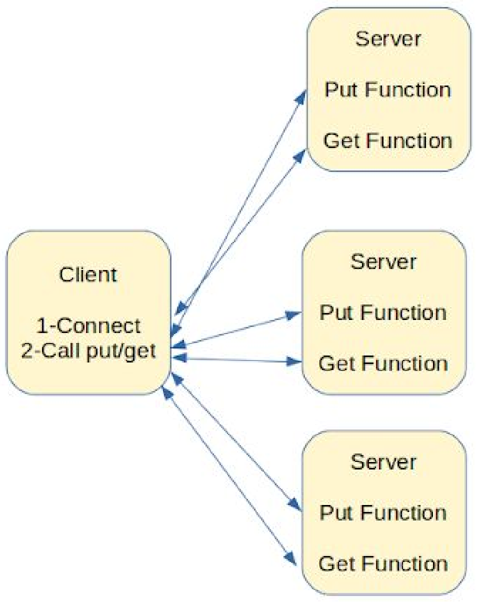


Fig1 - Distributed Network model connects through XML-RPC

The DHT also can works under multiple clients, since there is a list of server nodes and different clients can connect different servers. Clients are responsible for finding closest nodes and storing/retrieving data on servers. Each client has the key of servers and can figure out which node is closest. At the start, clients do not know about server keys, so they connect to servers calling a function “conn” and retrieve their keys. Later on, they used these key to call put/get functions.

5 RHT Model

In the new project we have two new features we have to take care about:

1-Each key,value pair must be stored on more than one node (preferably two)

2-There is a new request called multi-put on which a client call to put more than one put and must be processed if and only if all servers targets are ready to store the data.

I devised a coordinatore that works separately. Coordinator each time can receive bunch of messages from clients and it must redirect requests to servers. The main reason I devised a coordinator was because a node can handle the errors, requests and manage the packets to servers. Now, coordinators are responsible for calling servers and clients call coordinators functions remotely using XML-RPC. Each time a client call a function inside coordinator, the coordinator handle the on-going message and deliver messages to the servers. To make sure we won’t miss a package, face an unwanted multi access and similar problems as we discuss in the class three main features are considered.

1-locks: Each time a get has been called via a client, coordinator calls ‘get’ function inside server remotely and if the bucket is locked at the position return failed, otherwise locket the position using thread.locks in python and get the value. Before it returns the value from server, it will release the lock to make sure no error will occur. If an error shows up, the function also catch the error first and then release the lock for others usage. For put and multi-put the same thing happens and we lock the position and after the job is done or an error happens we release the lock.

2-2Phase Commit: In order to make sure we won’t store one value for a key on one server and another value on others, I used 2Phase commit. I did not use 2phase commit for get, since we consider that if a node returns a value we can rely on that. For both put and multi-put first coordinator calls server targets and asks for a lock on desired positions.

Later coordinator asks for commit if all servers return “READY’ or abort to release locks if at least one server returns “NOTREADY”. Since in commit part one node may fail to put a value, and there is no procedure to roll-back the data, we return ‘SEMISUCCESSFUL’ if at least one server fails and at least one server succeeds. We can easily roll-back the data if we get the value at the beginning and put the data back if one server fails.

3-logs: Logs are used to recover crashes. Since clients do not want to receive messages related to prior requests to their crashes and servers are accessed directly and remotely by coordinators, so coordinators are the best choice to store logs and later they can continue sending messages to servers or getting back the results to clients. Each log consists of message, sender and a sign that shows it is succeeds or not. If a server crashes before it can say the job was successful or not, then there won’t be a sign ahead of the message and we will know the request must be repeated just like when it was unsuccessful. Fig2, shows a schematic of the model using RPC protocol.

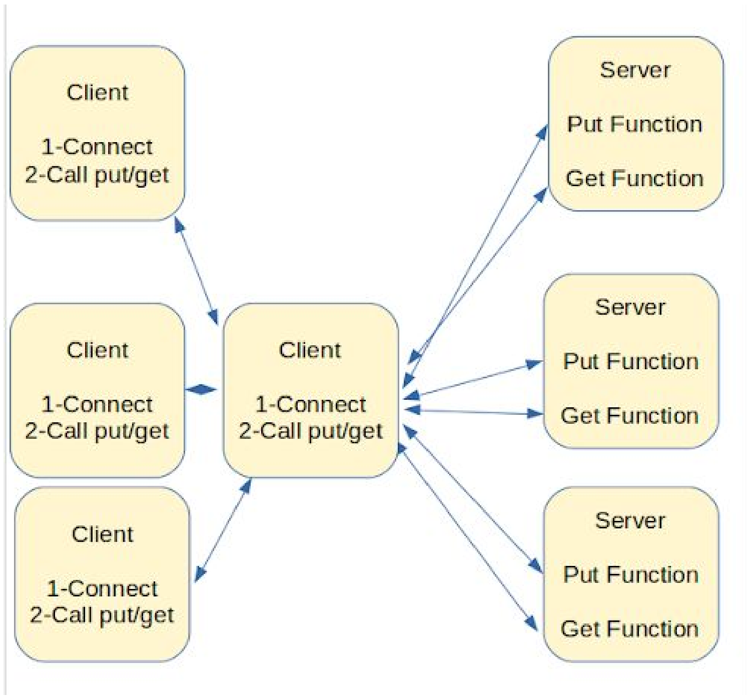
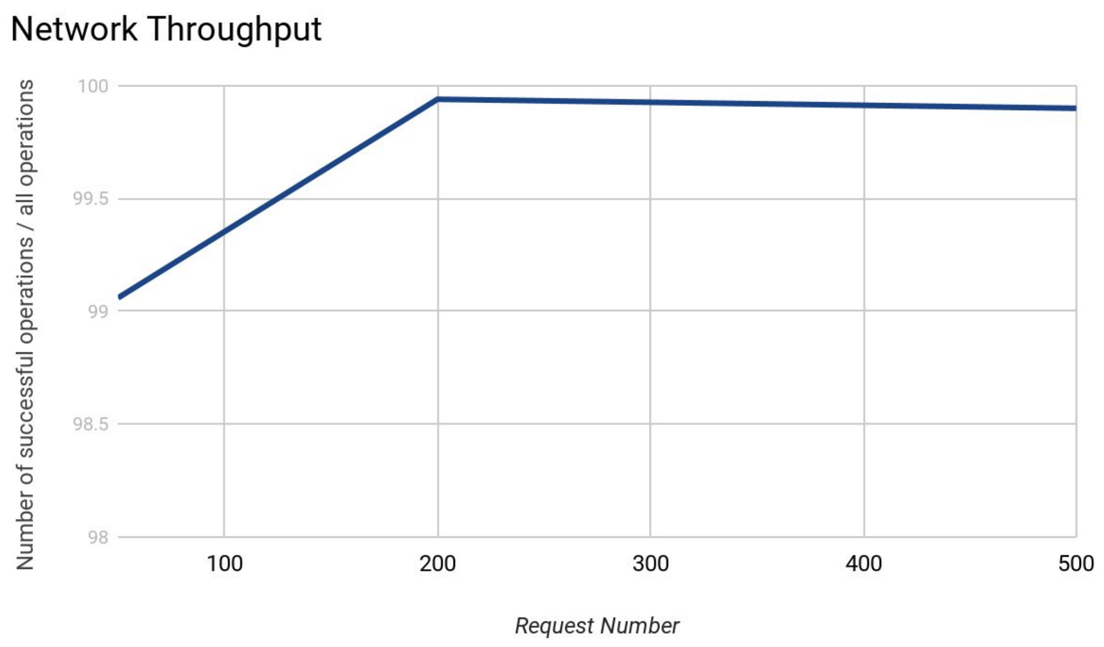


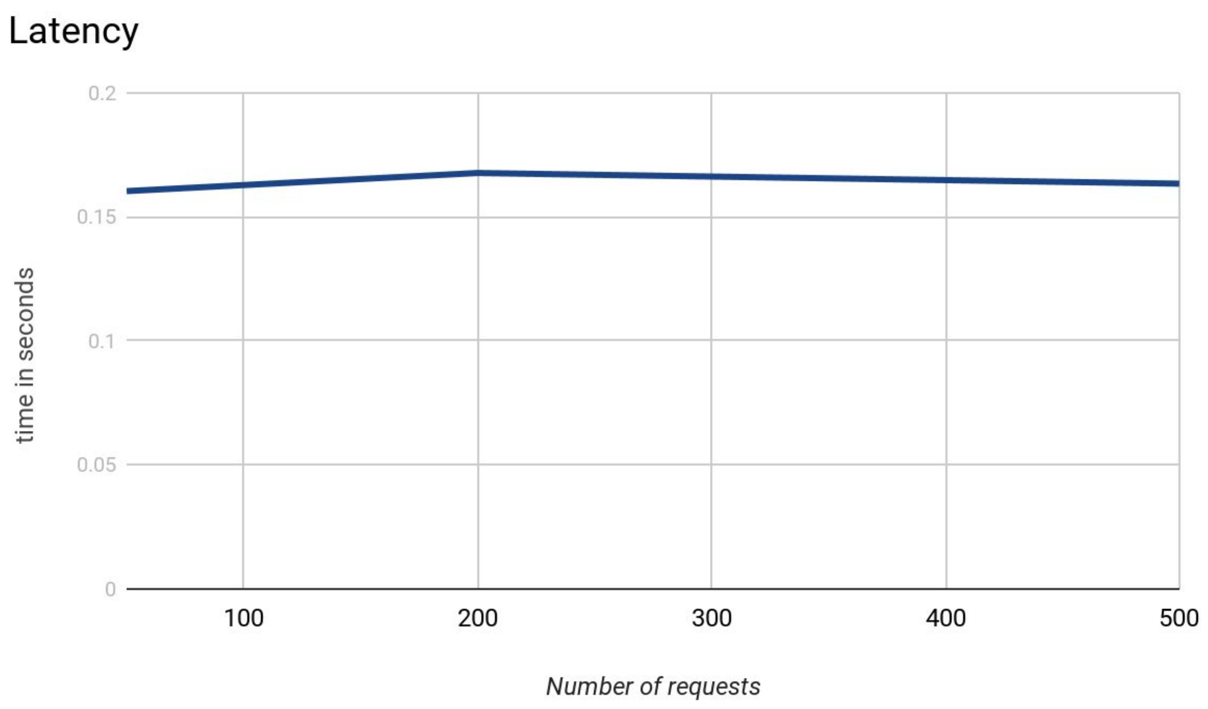
Fig2 - RHT model

5 Experiment Results

Here are the throughput and latency for different number of requests, 5 nodes, 8 threads of clients. As you can see the best performance for throughput is at 200 request which is 99.94% while the worst is for 50 requests, since the number of conflicts increases and there is less data stored in the bucket.



Latency like throughput starts with the lowest value and continues to the highest at 200 requests and got a little less in 500 requests. While it is expected for 50 requests that spent time is the less since there are fewer connections and communications, and by increasing the requests we locks more bucket positions and this make blocks for some access which ends in some delay. However, it is a little bit strange that the as the requests increased from 200 to 500 the time decreased.


Latency
0.2
0.15
0.1
0.05
0
time in seconds
100
200
300
400
500
Number of requests