INF3200: Distributed Systems Fundamentals
Assignment 1 Report

Michael Lau Ka Hin, Sufyan Saleem

September 29, 2020

Contents

1 Introduction

Almost every peer to peer systems face the issue of how to locate a particular node that stores certain data. To solve this issue, the Chord protocol was introduced as a distributed lookup protocol. When provided with a key, the Chord protocol maps it on the node and stores the key/value pair on the node to which the key maps. First, we decide a common key space for the nodes and data, then we position them in a hash ring or circle like structure and define a strategy for assigning data items to the nodes. The key we used for nodes is computes-$< X >$-$< Y >$: $< ZZZZZ >$ where compute-$< X >$-$< Y >$ is the host name and $< ZZZZZ >$ is port number. We use the key of the data and the key of host to hash them separately with the same MD5 algorithm into hash table, which produces a 256 bits identifier id of each node and data. All data and nodes are positioned on the logic ring structure, then we sort the hash table in alphanumeric order. As a result, every node can maintain a neighbours table and a table of data that the node is responsible for. These two tables will be sent to the node during the process startup phrase into order to serve requests.

2 Background

Distributed systems, like *peer-to-peer* systems (p2p), runs on different computers that share the same functionalities and a decentralized control. The main idea of key value store is the lookup of data items stored on different computers using the various protocol, the *Chord protocol* is one of them. Given a key, the Chord protocol maps it on a node(computer) and stores the values associated with the key. It improves performance, scalability, fault tolerance at the cost of complexity and the developing effort compared to running a single computer.

A hash table is used for managing the data in a p2p system, allowing process to communicate with one another through message passing. The data is distributed between different nodes. The node responsible for storing the key value pair (data) is decided by the partitioning scheme used.

Consistent hashing is used by chord to assign the keys to the nodes(computers). To know Consistent Hashing one should know Hashing, it is about mapping an object of any random size to an integer results in a hash code. Cryptographic hash functions like SHA1 and MD5 are used to hash the keys before assigning to different nodes. A Distributed Hashing schema assigns the nodes (servers or computers) and objects (data), position on an abstract hash ring or circle without affecting the overall system. This allows the system to scale easily and is independent of the number of nodes. Chord protocol is the backbone of databases Dynamo, Apache, Cassandra and other like Redis and Memcached.

3 Related Work

Chord stores key/value pairs at the node to which the key maps. A value can be any data like an address or a document etc. The Chord protocol can utilize hostname and IP address without relying on any other sever while DNS relies on different servers. The hostname is the key while the IP Address is the value. It imposes no naming structure and can be used to find data not related to a certain server(machine). A strict naming structure is followed by DNS, which is used to find named hosts.

Other decentralized system like the chord is Freetnet (p2p). It does not assign data to nodes(servers) but look for cached copies of data and does not provide assurity of retrieving the data like chord. Just like chord consistent hashing is used by the Ohaha system but follows routing style like the Freenet. OceanStore uses a variation of distributed data protocol which is similar to the Chord protocol. It provides a stronger guarantee than chord and makes sure that the request does not go further in the network than the node where the data is stored.

While chord can support concurrent failures and node joins well and is less complicated. The peer to peer (p2p) systems design is simplified by using chord. It acts like a natural load balancer distributing the keys evenly over all the nodes. It is used in loosely coupled p2p systems and is fully distributed. It has no strict naming structure and is highly available even if the system is changing by nodes leaving and joining. Chord is also scalable as the number of nodes grow so does the cost of lookup, but no manual tuning is required for the system to scale.

4 System Mode

4.1 Architecture and Design

The Chord system is based on the chord protocol, which has one primarily goal: to decide which node is responsible for a given key. The Chord system enables applications to perform lookup, insert values actions with a key as the medium. A logical ring with consistence hashing is implemented in order to make the Chord system works in a more effective manner.

The design of model starts with a common identifier space for both nodes and the key-value pairs. To achieve this, the hostname and a randomly assigned port number of the node is used for hashing, and the key of the key-value pair is used for the same hashing. So, each node and each data has its own identifier id. After the hashing table is created, all the data including the nodes and keys are sorted in alphanumeric order.

Once the data are being sorted, the nodes are able to identify their neighbours next to them, and each key is able find its corresponding successor, which is the first node it encounters when it keeps going up in the ring. One special case here is that the previous node of the first node is the last node of the ring, and the successor of keys that are greater than the last node is the first node of the ring. Once all the nodes and keys are connected as a logical ring, each node is aware of its neighbours and the keys that it is responsible for.

As a result, to lookup or insert a data, the node uses the same hashing function to hash the key of the incoming data first, then it will know instantly whether it contains the data or not. If the hashed key is not found, two possible cases would happen: In the first case, if the hashed key is inside it range, which can be represented by $(< previous\_node\_id >, < own\_id >]$, then it will handle the case and consider the data is not found. The second case is that it will just distribute the task to its previous node or its next node depending on the id of the incoming data, then wait for the reply. In the worst case, the lookup speed would be $O(n/2)$ with n nodes.

4.2 Implementation

To start the Chord, there are some prerequisites. Since we are using NodeJS as the server, you need to download "node" first from , and then run the command *npminstall* to install the required packages that is defined in the file $package.json$. As for the data, the key-value pairs can be pre-defined inside a file named $keysfile$. The format of the data is $(key, value)$, and one set of data on each row. This file will be read during the startup. The startup script is stored in a file named $chord\_start.sh$, and two js files are required for the hashing function and server startup, named $app.js$ and $hash_keys.js$respectively.

After the set up, we can start the Chord. The startup command is $./chord\_start.sh\ X$, where X is the number of server that you want to use. The first thing inside $./chord\_start.sh$ script will do is to generate server and stored the hostname of the server into a file named $hostfile$.

Next, it will read the hostname line by line, randomly generated a port number between 49152 to 65535, and stored the address in the format compute-$< X >$-<

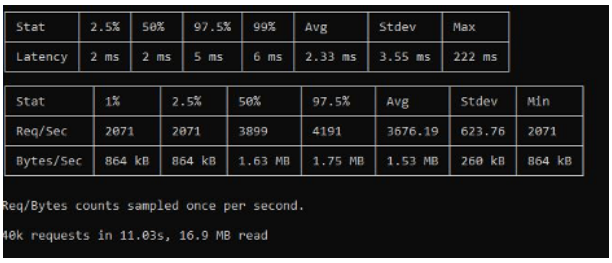
$Y >:< ZZZZZ >$, where compute-$< X >$-$< Y >$represents the hostname and $< ZZZZZ >$ represents the port number. We will then use the *hash_keys.js* file to hash this address into the hash table by MD5 and get back a 256-bits unique Chord node identifier. The keys of the key-value pairs are also being hashed into the using the same hash algorithm so consistent hashing is achieved.

Moreover, we will sort node identifiers in alphanumeric order for both data and server, such that each server will have its own neighbours table, along with a table of data identifier that it is responsible for.

Finally, we will start running the Chord by using ssh to get to that server and then use *app.js* file to start the process for serving 3 APIs calls: PUT and GET request for key-value pairs, and a GET request for neighbours. A few environmental variables are passed into the process, including the port of the process, the neighbours table, the table of data identifier, its own index and its own identifier. These variables are used in the *app.js* to implement the logic of that 3 APIs calls.

5 Evaluation and Results

A Nodejs package called `autocannon` was used to measure performance with is a benchmark tool. The metrics used were request per second, throughput, and latency. Test cases are written for the three different API'S which will be running on different compute nodes and ports, given a time between 10 to 60 seconds, 10 users will be making requests to the different the hosts for this given test cases and will output the results in the form of a table. The names of the test cases are `getNeighbors`, `getstorageItem` and `putstorageItem`. The figure 1 illustrates an example of `getstorageItem` running on one node.



Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	2 ms	2 ms	5 ms	6 ms	2.33 ms	3.55 ms	222 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	2071	2071	3899	4191	3676.19	623.76	2071
Bytes/Sec	864 kB	864 kB	1.63 MB	1.75 MB	1.53 MB	260 kB	864 kB

Req/Bytes counts sampled once per second.
40k requests in 11.03s, 16.9 MB read

Figure 1: an example output

The Figure 2 shows the number of requests per second given 1 to 16 nodes for the *GET STORAGE API*.

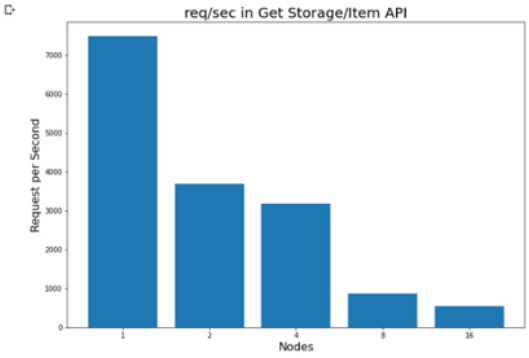


Figure 2: an example output of GET STORAGE API

The Figure 3 displays the number of requests per second given 1 to 16 nodes for the *PUT STORAGE API*.

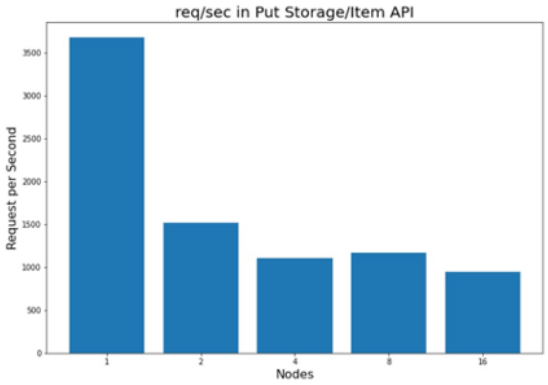


Figure 3: an example output of PUT STORAGE API

6 Future Work

In our implementation, each node keeps its own neighbours table only, which produces the lookup speed as $O(n/2)$ with n nodes in the worst case. To improve the lookup speed, we could provide a routing table for each instead of a neighbours table. To achieve this, we could create the routing table with logarithmic size S, where $M = 2^S$ and M is the size of the identifier space. So, each node will know about the successor$(N + 2^{(i-1)})$ for $1 = 1 \ldots M$, and hence the lookup speed will be reduced to $O(log\, n)$ in the worst case.

In addition, we also need to handle the maintenance of the ring. Periodic stabilisation and predecessor table can be utilised to make every route eventually correct. Also, we need to consider the cases when some nodes enter or leave the ring.

7 Conclusion

To implement the Chord system, first we utilise the consistence hashing with MD5 hashing function on both the node and data to distribute the keys evenly across nodes in the system. Then we sorted the keys in the hash table and create the neighbour table and data's successor table for each node. After that, we distribute the tables to corresponding node and start up the node in order to serve 3 APIs calls, which are GET, PUT request for data and GET request for neighbours. Through the evaluation, we could see that as the number of hosts increases, the number of requests a host need to handle decrease.