INF3200: Mandatory Assignment 1

Yasiru Rathsara Witharanage

September 26, 2021

Abstract

In an ever-growing tech-world, the necessity of storing and accessing data is vital. With the expansion of industry and users, data-stores are being required to address additional features apart from its main objective, such as scalability, replication, accessibility etc. To this end, distributed data-stores have been more useful and applicable. In this study, I have implemented a distributed hash table using chord [**1**] and conducted few experiments to observe how its performance differs with cluster's scale.

1 Introduction

The primary objective of this study is to build a system of distributed nodes acting as a single key-value store with the assistance of a distributed hash table 'chord', which uses a ring network of individual nodes with distributed memory each with the knowledge of only a partition of the entire network. This store exposes the following HTTP endpoints for querying and debugging purposes.

- PUT /storage/key - to store a value provided in the raw body
- GET /storage/key - to retrieve the value of a key (returned in raw body)
- GET /neighbors - to retrieve the neighbours of a node

2 Project Description

2.1 Design

Each individual node in the network is given an id based on the SHA256 hash value of its own hostname. Thus, specific instances were chosen to resemble and organize the ring structure such that there are no multiple instances for the same id. Fig.1 shows one of such structures with 16 nodes used in the experiments of this study along with generated node ids and hostnames.

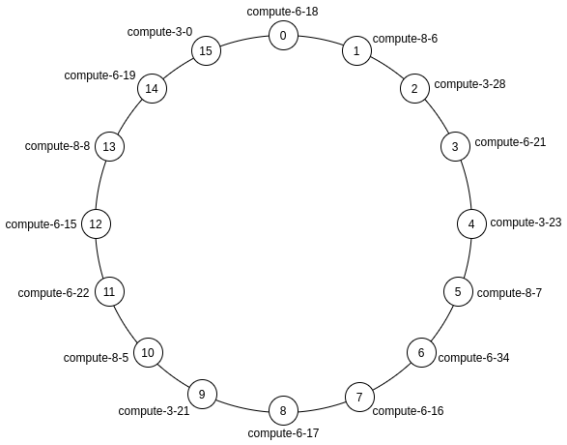

Figure 1: Ring structure with 16-nodes

Each node is only aware of its neighbours, i.e. predecessor and successor. These values can be configured before the execution of a node.

Whenever a query is received by a node, it initially checks whether the requested key belongs to itself and if not, it proceeds to the immediate successor. This process is iterated until the correct node for key is discovered and the corresponding response is returned along the same route.

2.2 Implementation

Golang was used for the implementation of chord with a flat-hierarchical code structure and different layers for each functionality. HTTP endpoints listed previously were implemented in HTTP layer while node layer contains details (node id,

predecessor, successor) related with the node along with validating key function. SHA256 was used for hashing of both key and node id to match with relevant buckets. A separate HTTP client was used in neighbour layer to proceed any request with a key that does not belong to the corresponding node. All key-value entry pairs are stored in an in-memory map which is specifically designed for concurrent use.

A sample set is given below for the configuration values of parameters required by the implementation but however user is allowed to provide his own desired values for service configurations. Predecessor and successor ports can be given null if they are same as the port of current node.

```
# service configs
port: 52520
max_num_of_nodes: 16
request_timeout_sec: 2
ttl_min: 180

# neighbour configs
finger_table_enabled: false
neighbour_check: false
predecessor: "compute-1-1"
predecessor_port: ""
successor: "compute-1-1"
successor_port: ""

# logger configs
colors_enabled: true
log_level: "TRACE"
file_path: true
```

3 Evaluation

Experiments were carried out to measure and evaluate latency variation under different circumstances. A separate client was implemented for this testing purpose and can be found under tester directory.

In these experiments, a number of requests was sent and total latency of the whole batch was measured in order to calculate the average latency per request in different scenarios. Each attempt was made 3 times and the resultant average total latency value was considered for the calculations. This entire process was done several times by changing load (number of requests) and number of nodes in the cluster.

A set of 15 dummy requests was used in these attempts where each request consists of a key belonging to a separate bucket. This set was iterated as many times as required until the number of requests being tested was matched.

Below shows the tables containing hardware specifications of nodes and sets of nodes used in each experiment.

Node ID	Hostname	Model	CPUs	Cores	Processors
0	compute-6-18	Lenovo P310	1	4	4
1	compute-8-6	Lenovo m93p	1	4	8
2	compute-3-28	Dell T3600	1	4	4
3	compute-6-21	Lenovo P310	1	4	4
4	compute-3-23	Dell T3600	1	4	4
5	compute-8-7	Lenovo m93p	1	4	8
6	compute-6-34	Lenovo P310	1	4	4
7	compute-6-16	Lenovo P310	1	4	4
8	compute-6-17	Lenovo P310	1	4	4
9	compute-3-21	Dell T3600	1	4	4
10	compute-8-5	Lenovo m93p	1	4	8
11	compute-6-22	Lenovo P310	1	4	4
12	compute-6-15	Lenovo P310	1	4	4
13	compute-8-8	Lenovo m93p	1	4	8
14	compute-6-19	Lenovo P310	1	4	4
15	compute-3-0	Dell T3600	1	4	4

Table 1: Hardware specification of nodes

Number of nodes	Node IDs
1	10
2	0, 7
4	2, 5, 8, 11
8	0, 1, 4, 6, 8, 9, 10, 13
16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

Table 2: Nodes used in each experiment

4 Results

Following are the average execution times obtained per each query type (GET and PUT) while varying number of queries in the batch and number of nodes in the cluster. Note that throughput experiments in clusters with higher number of nodes were limited due to file descriptor limits.

Number of nodes	Size of load (No. of requests)	GET - Average time (ms)	PUT - Average time (ms)
1	16	0.2708	0.3541
	100	0.0956	0.0995
	1000	0.058	0.062
2	16	0.345	0.3125
	100	0.12	0.11
	1000	0.065	0.064
4	16	0.5625	0.625
	100	0.19	0.21
	500	0.102	0.118
8	16	0.8958	0.9375
	100	0.25	0.2667
	500	0.158	0.144
16	16	1.5208	1.5625
	100	0.38	0.38
	500	0.158	0.146

Table 3: Results of experiments

5 Discussion

6 Conclusion