INF3200: Mandatory Assignment 1

Yasiru Rathsara Witharanage

September 27, 2021

Abstract

In an ever-growing tech-world, the necessity of storing and accessing data is vital. With the expansion of industry and users, datastores are being required to address additional requirements apart from its main objective, such as scalability, replication, accessibility etc. To this end, distributed datastores 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 and number of requests in the batch.

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. Figure.1 shows one of such structures with 16 nodes used in the experiments of this study along with generated node ids and hostnames.

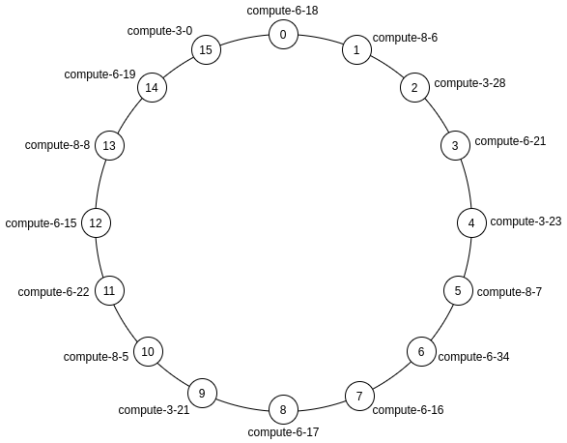

Figure 1: Ring structure with 16-nodes

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
```

`max_num_of_nodes` here should be equal or larger than the number of nodes being tested. `request_timeout_sec` refers to the waiting time of a single HTTP request by a node in seconds. Moreover each node has a TTL in minutes for the self-termination configured by `ttl_min`.

3 Evaluation

Several 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 of the project.

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. 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, thus minimizing the erroneous influence of a particular node on a test attempt. This set was iterated as many times as required until the number of requests being tested was matched.

Table 1 and 2 provide details about 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

4 Results

Following are the average execution times obtained per each query type (GET and PUT) while varying

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

number of queries in the batch and number of nodes in the cluster (Table 3). Note that throughput experiments in clusters with higher number of nodes were limited due to file descriptor limits. Each attempt was done 3 times and the resultant average total latency value was calculated for the average time.

No. of nodes	No. of requests	Latency-GET(ms)	Latency-PUT(ms)
1	16	0.2708	0.3542
	100	0.1066	0.1133
	1000	0.0580	0.0553
2	16	0.3750	0.3958
	100	0.1267	0.1233
	1000	0.0630	0.0607
4	16	0.5833	0.6042
	100	0.1900	0.2033
	500	0.1007	0.1240
8	16	0.9167	0.9375
	100	0.2467	0.2767
	500	0.1653	0.1407
16	16	1.5417	1.4583
	100	0.3833	0.3733
	500	0.1693	0.1460

Table 3: Results of experiments

5 Discussion

Following histograms (Figure 2 and 3) with error bars for standard deviation [**2**] were derived by using the above results where y-axis implies the calculated average time and x-axis denotes the experiments in the form of (number of nodes, number of requests).

Line graphs in Figure 4 and 5 were constructed for more precise interpretation and comparison of experiment results.

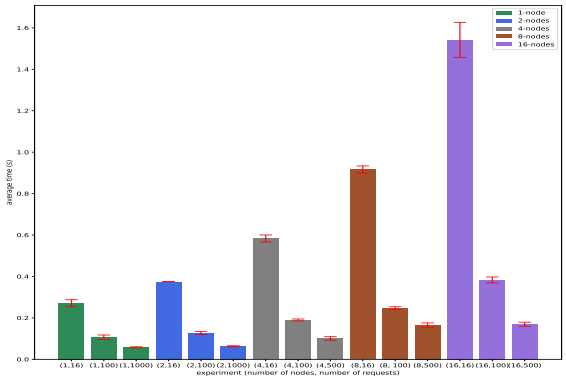

Figure 2: Histogram of average times for GET requests with different parameters

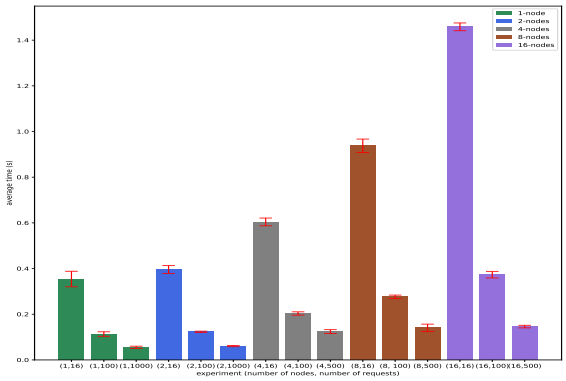

Figure 3: Histogram of average times for PUT requests with different parameters

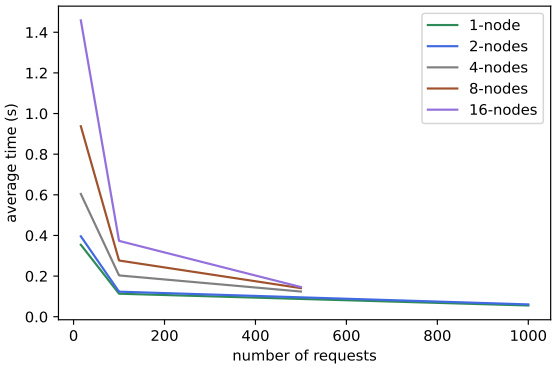

Figure 4: Line graph of average times for PUT requests with different parameters

By observing these charts, it is obvious that response times increase with number of nodes in the network for a particular batch of requests

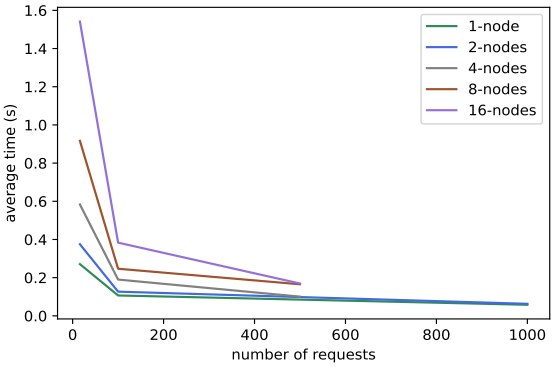

Figure 5: Line graph of average times for GET requests with different parameters

and can this be explained by the rise in redundant HTTP calls among nodes when a network expands. However, this increasing factor seems to be diminishing when the size of the request batch increases.

Furthermore, for a given network with a fixed number of nodes, latency decreases as opposed to number of requests in the batch. A possible reason for this observation could be that the capability of a node to process query requests asynchronously while the rest are still in progress or being circulated in the network and thus maximizing the throughput relatively.

This entire behaviour is consistent across both request types (GET and PUT) despite the minor variations in latency. Lower average times are highly possible if the communication among nodes can be replaced with a better approach.

6 Conclusion

Finally, it can be concluded that the implemented key-value store is more appropriate when the number of queries is higher in terms of the performance for a given network of nodes. In contrast, number of nodes in the network has a negative impact on the query latency.

Thus, the distributed key-value store should be designed with a preferable number of nodes such that it addresses the trade-off between performance and other quantitative/non-quantitative measures of a distributed system (scalability, accessibility, load of queries, fault tolerance etc.) depending on the requirements of a given scenario.

References

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan (2001) *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications* , MIT Laboratory for Computer Science.

[2] Lee, Dong and In, Junyong and Lee, Sangseok (2015) *Standard deviation and standard error of the mean*, Korean journal of anesthesiology.

A Source Code

Github repository