INF3200: Mandatory Assignment 2

Yasiru Rathsara Witharanage

October 24, 2021

Abstract

In the context of distributed systems, the term stability is trivial as the existence of individual nodes are not predefined and fixed. Hence it is significantly important to address the issue of inconsistent structure dynamically while minimizing the adverse impact on performance of the system thus providing transparency to the end-user. This paper discusses a solution to this complication of ever-changing participants of a key-value store based on chord [1] along with the experiments conducted to measure its performance and behavior.

1 Introduction

In this study, a distributed hash table was implemented based on chord which uses a ring topology where each node in the system is only aware of its predecessor and successor initially. A node is responsible for storing and retrieving of any key-value pair that belongs to its bucket space based on node id. These keys and node ids are first hashed using SHA256 [2] algorithm in order to minimize the collision of two separate nodes and to utilize the potential of supporting an enormously large distributed store.

Main objective of this study is to build a chord network resilient to dynamic changes which can be occurred due to joining and leaving of nodes. This adaptation also includes coping against crashing of nodes without any prior notification and re-construct network leaving these out-of-reach nodes. Moreover, it should be able to detect these left out nodes and include them in the network whenever they are reachable again.

Following HTTP endpoints are exposed to serve store functionalities as well as to cope with dynamically changing structure of server nodes. Note that nodes are not aware of cluster information and cluster-info endpoint circulates an HTTP request around the ring structure to collect necessary data.

- PUT /storage/key - stores a value provided in the raw body
- GET /storage/key - returns the value of a key (returned in raw body)
- POST /join?nprime=HOST:PORT - joins the network using an existing node
- POST /leave - leaves out the requested node from network
- GET /neighbors - returns the neighbours of a node
- GET /node-info - returns a JSON response of requested node information
- GET /cluster-info - returns cluster information

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 and 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.

When joining the network, a request is sent to the new node along with an existing node in the cluster and thus a join is initiated to this entry point where it starts circulating the request until the corresponding successor of new node is reached. This successor then updates its new predecessor while notifying the ex-predecessor to update its successor. Once this circulation is successful, neighbors of the new node will be returned by the response and updated in the joining node.

On the other hand, a node notifies both its predecessor and successor upon leaving, to update their neighbors accordingly. Since these two requests are independent of each other, they can be executed in parallel thus saving an overhead latency of multiple requests. In both these scenarios, bucket space will be adjusted with respect to the network change.

Moreover, a probing mechanism is implemented in each node to detect any out of reach condition of its predecessor and automatically rectify with active nodes by eliminating the crashed node. Once a crash is recognized, the successor will circulate an internal HTTP request until the last active node is found and thus updating a new connection between these two nodes. Immediate successors also keep track of these disconnected nodes to add once they become reachable again and this check is done in the order of proximity, i.e. closest lost node will be checked before the rest and so on. However, this system does not support the migration of existing key-value pairs when changing the components of network.

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.

```
# service configs
port: 52520
request_timeout_sec: 5
ttl_min: 180

# neighbor configs
probe_interval_sec: 10
detect_crash_timeout_sec: 2

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

`request_timeout_sec` refers to the waiting time of a single HTTP request by a node in seconds and each node has a TTL in minutes for the self-termination configured by `ttl_min`. `probe_interval_sec` is used as the time interval in which each node is checking if their immediate predecessors are reachable. A separate HTTP client is used to detect whether a successor is not responsive in finding the last active node triggered by a node crash. This detection is done with a timeout configured by `detect_crash_timeout_sec`.

3 Evaluation

Several experiments were carried out to evaluate performance and stability of network with dynamic changes in the structure. First experiment was conducted to measure the time consumption of network to stabilize after varying number of joins of nodes. This was achieved by sending sequential join requests to the cluster and logging timestamps with each node's action. Total time to stabilize was taken as the time difference between initial join action and the last neighbor update by a node. Finally cluster information response was considered to ensure that the network consists of the desired number of nodes.

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

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 (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).

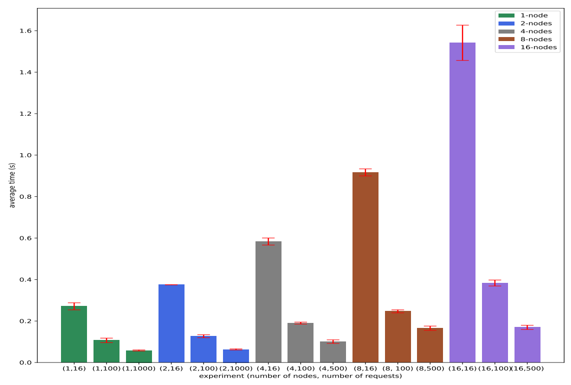

Figure 1: Histogram of average times for GET requests with different parameters

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

By observing these charts, it is obvious that

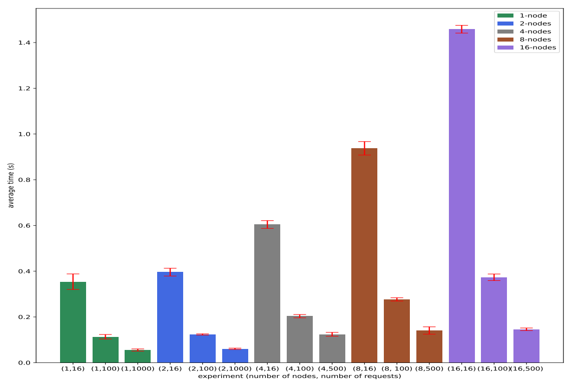

Figure 2: Histogram of average times for PUT requests with different parameters

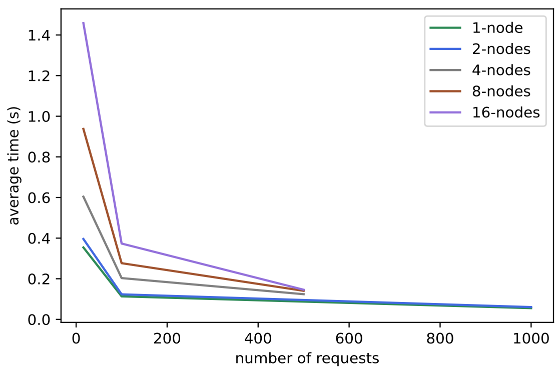

Figure 3: Line graph of average times for PUT requests with different parameters

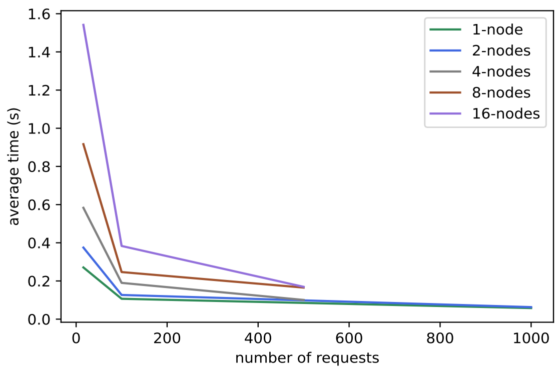

Figure 4: Line graph of average times for GET requests with different parameters

response times increase with number of nodes in the network for a particular batch of requests 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