1. Introduction

The purpose of the assignment was to make a distributed hash table (DHT) with implementation of the Chord protocol [1]. The system should consist of a set of nodes, each of the should be able to store and retrieve values by the key via the HTTP protocol.

After the implementation of the DHT, the set of measurements should be done to evaluate the throughput of the system in PUTs+GETs http requests per second.

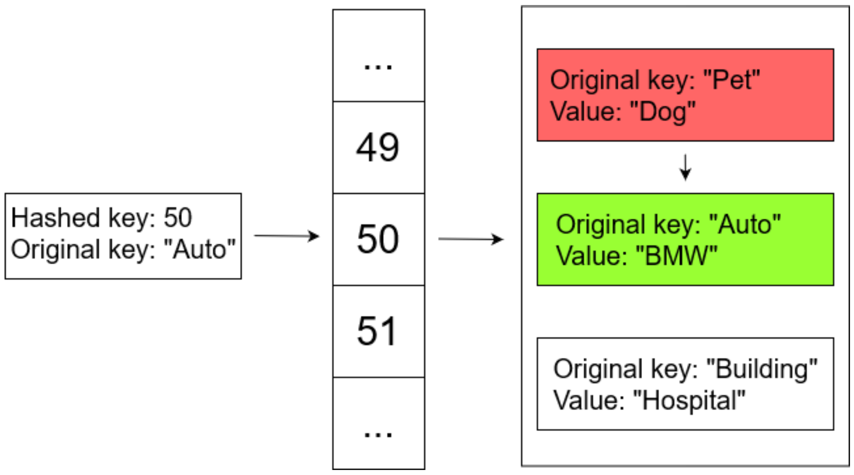
The solution was implemented in Go language with additional Viper package for working with configuration files.

2. Implementation

Due to the Chord protocol every node in the set gets an identification number which belongs to the key space range. This number is usually being computed with a hash function of the host name or IP, but for this work a small keyspace was chosen to show the collisions handling of putting and retrieving data with the same hashed value of the key, so nodes identification numbers are manually set at the launching of the process to avoid the collisions of node IDs. After the manual input of node ID, it's being multiplied by the value of *keySpaceSize / nodesCount*. So, every node would contain the equal amount of records.

Identification number is used for assigning the hashed key to the node. If the records hashed key is in range of *successorNodeId – nodeId*, the record can be put or retrieved from the node. Otherwise, the record is sent to the successor node.

To avoid the collisions every node contains an array of dynamic arrays which store information about records put in the DHT. On every attempt to retrieve a record by the key, this key is compared with the non-hashed key of a record and if they are equal, the value returns, otherwise the same operation is being performed on the next record in the array until it reaches the final element. This method is called separate chaining [2]. An example for searching a key is shown on a Picture 1.



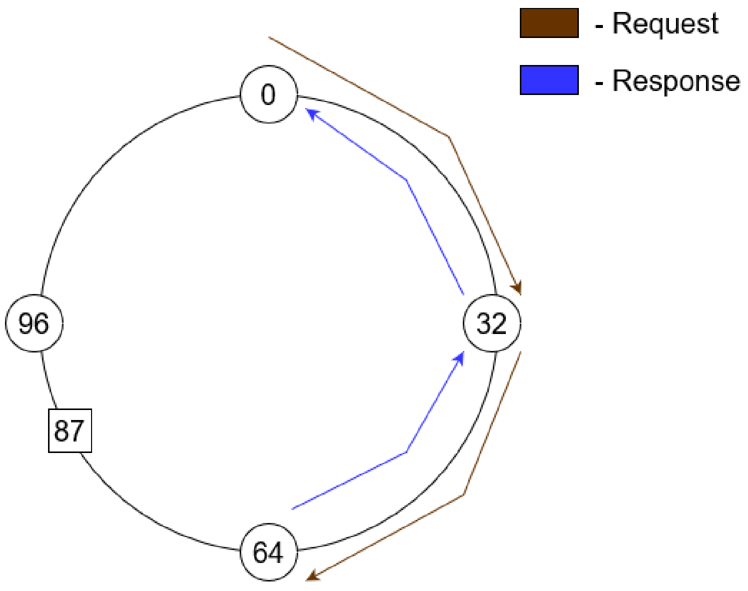
Picture 1. Searching a key in a chain.

The DHT data input is made with HTTP PUT request. Key of the record is the last part of the requests URI path, value is raw request body. If a record with such key already exists in a table, a message about existence of a record with such key returns.

Records value can be retrieved from DHT with GET request. If an array element related to the hashed key on the correct node was empty or contained a list of records, none of which had equal original key, a message about missing a record with such a key returns.

The Chord model can be presented as an “identifier ring” where nodes and table records are arranged. This solution implements simplified Chord algorithm without implementation of finger tables and nodes joining. Every node only knows information about their successor and predecessor, which is passed as command-line arguments.

An example of getting a value by the key is presented on a Picture 2 below.



Picture 2. Search of a hashed key with a value of 87 in a DHT with 4 nodes and keyspace of 128.

The hash function used in this solution is SHA-1. It is a cryptographic hash function which takes input message in bytes and outputs a 160-bit message. After that SHA-1’s output is being transformed to an integer in a range of a key space [3].

3. Experiment

The purpose of this experiment is to measure the throughput of the solution by sending HTTP requests during one minute and calculating how many requests per second were processed by the system. A client on Go was implemented for the experiment. It sends requests to one of the hosts with randomly generated keys to put or retrieve from the DHT. After five minutes it outputs the number of successful responses, number of errors and the amount of requests processed by the system per second.

There measurements were made on a system with different amount of nodes and were held three times each. All the measurements were performed on the UIT cluster servers. The average results are presented in a Table 1 below.

Number of nodes	1	2	4	8	16
Number of successful requests	2645	5651	5653	5640	5646
Number of failure requests	4898	4784	4213	4122	3534
Responses/second (including failures)	176	174	164	163	153
Responses/second (discluding failures)	94	94	94	94	94

Table 1. The average results of experiment 1.

The experiment have shown that the number of successful responses per second on a single host didn't change on a different number of nodes in the system. However, the number of failed requests was decreasing with the rising of number of nodes. This might be caused by a higher host latency but in this case the number of successful requests should have been decreasing as well.

Next experiment is pretty similar to the previous one, but the client now sends messages to random hosts. The results are presented in a Table 2.

Number of nodes	1	2	4	8	16
Number of successful requests	28225	48777	62080	34784	188082
Number of failure requests	22485	5456	0	0	0
Responses/second (including failures)	169	903	1030	580	301
Responses/second (discluding failures)	94	812	1030	580	301

Table 2. The average results of experiment 2.

The results show that with a twice enlargement of number of nodes, the throughput decreases almost twice. System with two nodes still does not manage to hold against client load, but system with four and more nodes can manage load without failures. The comparison of throughput of systems with different amount of nodes is presented on Figure 1.

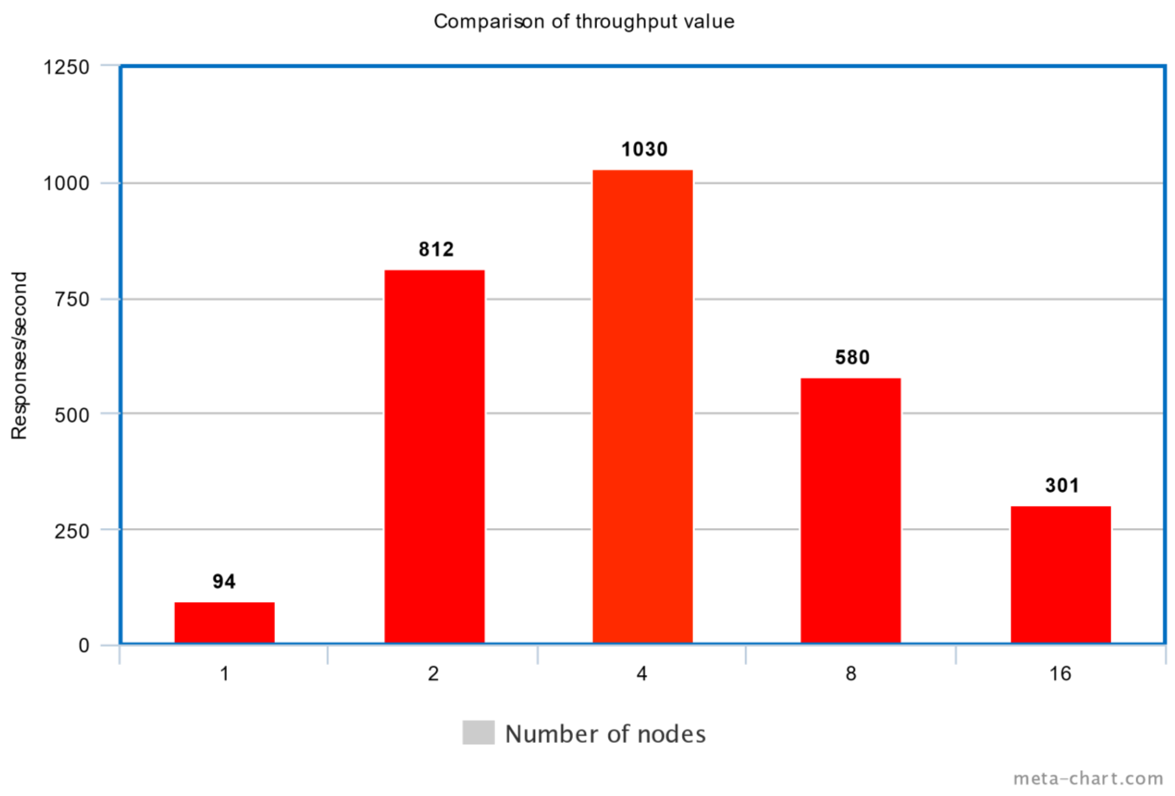


Figure 1. Comparison of throughputs on different amount of nodes.

4. Discussion

Current solution has a big number of improvements that could be made. One of them is implementing finger tables for a faster searching result. Current solution would be much slower with a big key space and a large amount of nodes than a solution with finger tables. This would also increase throughput of the system.

If this was a commercial project, the DHT would require a function of joining and removing nodes for simpler management of DHT resources during maintenance period.

The experiments have shown that the throughput of the solution is not optimal and strongly decreases with an enlargement of number of nodes. The research of this problem is required to find methods of optimization, besides implementing finger tables.

The SHA-1 algorithm used in this solution have been considered unsecure as the first collision was found in 2017, the information about algorithm weakness was provided by Google [4]. For the practical usage of the solution, more secure hashing algorithm should be chosen, for example – SHA-2 or SHA-3.

5. Summary

The presented solution successfully meets all the requirements of assignment, but still has a number of improvements that could be made to create a DHT table for usage in a real project.

The measurements of throughput of a system in number of HTTP requests per second were made and analyzed, analysis of measurements was provided in this paper.

6. Reference list

1. I. Stoica et al., “Chord: A scalable peer-to-peer lookup protocol for internet applications.” IEEE/ACM Trans. Netw., vol. 11, no. 1, pp. 17–32, Feb. 2003, doi: 10.1109/TNET.2002.808407.
2. Separate chaining collision resolution technique. OpenGenius. https://iq.opengenus.org/separate-chaining-collision-resolution-technique/
3. US Secure Hash Algorithm 1 (SHA1). RFC Editor. https://www.rfc-editor.org/rfc/pdfrfc/rfc3174.txt.pdf
4. J. Katz, H. Shacham, “Advances in Cryptology – CRYPTO 2017”. Pp. 570-596, Jul. 2017, doi: 10.1007/978-3-319-63688-7_19.