Chord Algorithm Implementation

Assignment 1

Big Data Management

Msc in Data Science

2017-2018

Vasiliki Konstantopoulou, email: dsc17011@uop.gr, AM: 17011

Angeliki Mylonaki, email: dsc17014@uop.gr, AM: 17014

Code implementation: https://github.com/DataScienceMSc/Chord_DS.git

Running the Executable

For the first Assignment we have implemented a simulation of the Chord algorithm. The programming language we used is python so the user can execute it in every platform supporting python.

To run the executable the user must insert the number of nodes as an argument.

For example to run the program using the VM's Mint operating system provided, the steps are:

- open a console in the folder containing the executable file Chord.py
- Type on the console "python Chord.py --N *number*", (*number* is the number of nodes)

Nodes creation design

Every node gets a unique hashed id. This ID is created by using the SHA1 method provided in python for hashing and encryption, divided with the maximum number of nodes that the ring can hold and finally getting the mod result from this division to limit it between 0 and the maximum number of nodes decided.

As for the maximum nodes in the ring, we were unable to create a 2^160 ring because of lack of hardware supporting so big numbers for hashing, creating many more problems described later. As a result, we decided to make a dynamic ring which can handle the next available power of two after the number of nodes typed by the user.

Movies requests design

The algorithm searches for movies which follow the power of law distribution using an implementation provided as a package by scipy of python. Analytically, in the beginning of the program the algorithm randomly generates numbers, between 0 and the maximum number of nodes, that represent hashed movies' IDs.
Due to hardware limitations that does not support too big numbers (as described above), we assumed that the dynamic ring created, supports dynamic number of movies. For example in a ring that supports 2^10 nodes, the maximum number of movies supported is 2^10. As a result, the hashed ID of a movie corresponds to the number of line where the movie exists in an excel file.

Statistical description of Results

Observed Latency

The deliverable folder contains the excel files described below:

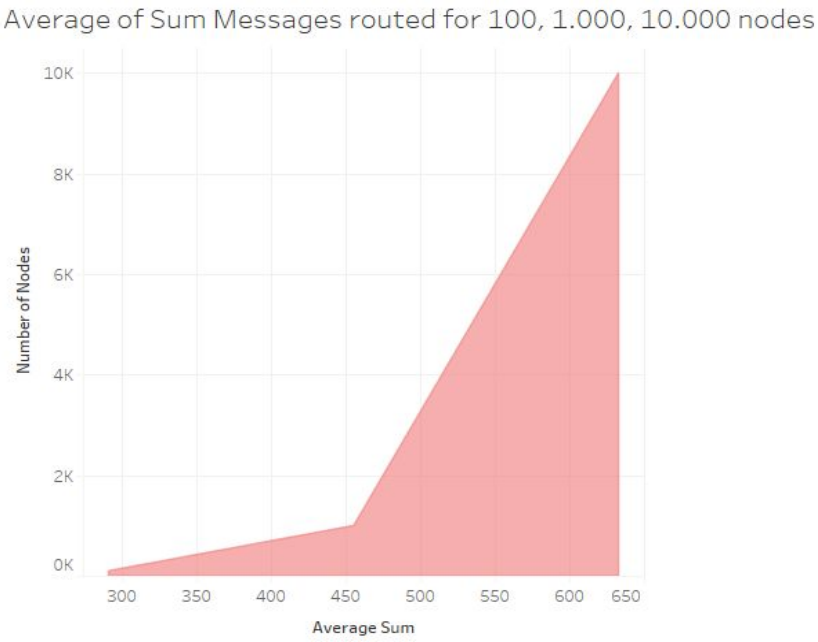
- The load_MaxNodes_MM_requests_RR.csv files contain the average messages routed and the average messages served for each node for 5 experiments. (MM is the number of nodes that could exist in the ring and RR is the number of requests) - three excel files created from running the program providing 100, 1.000, 10.000 nodes as an argument with fixed number of requests 100, 1.000, 10.000 for each group of experiments.

- The hops_MaxNodes_MM_requests_RR.csv files contain the averages of the average hops a request may make from the beginning of searching till it is found in the right node - three excel files created by running the program providing 100, 1.000, 10.000 nodes as an argument with fixed number of requests 100, 1.000, 10.000 for each group of experiments.

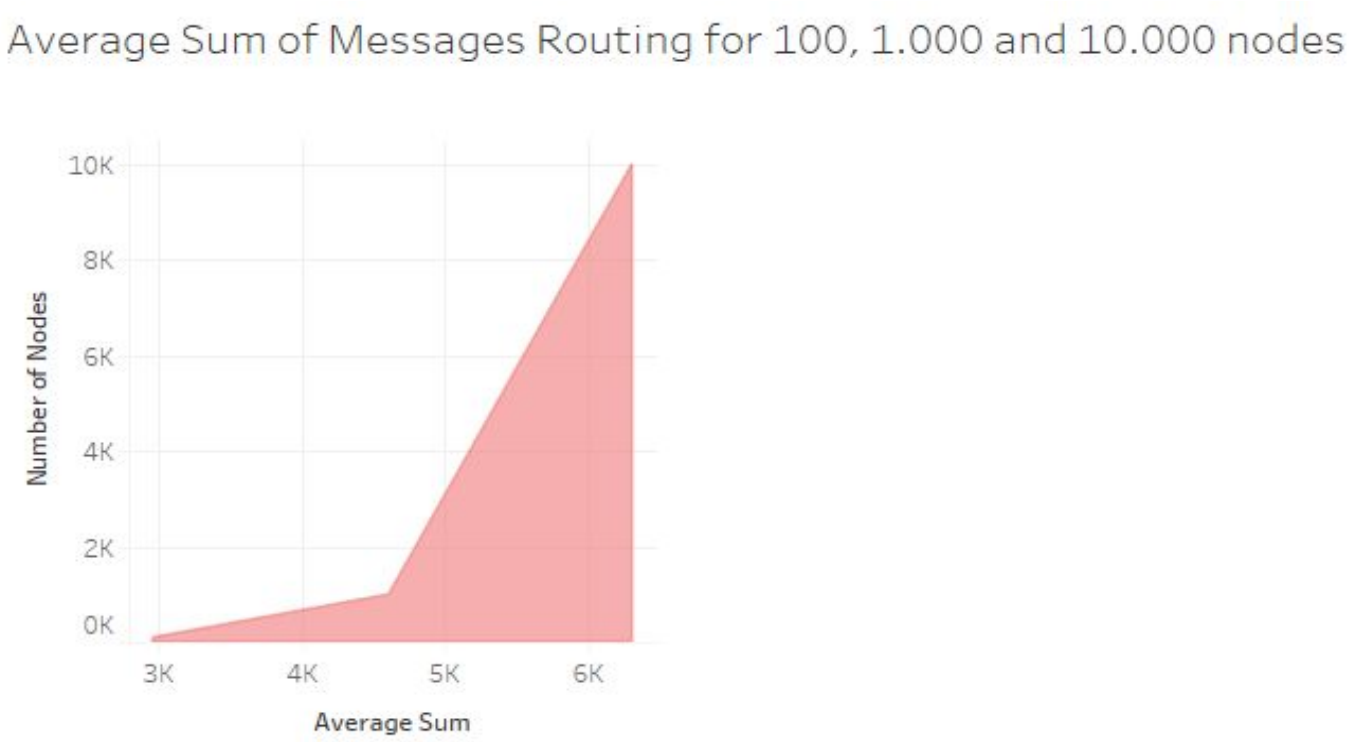
  (totally 18 excel files delivered)

Below we provide three graphs showing the changes in the number of messages as the network size increases.

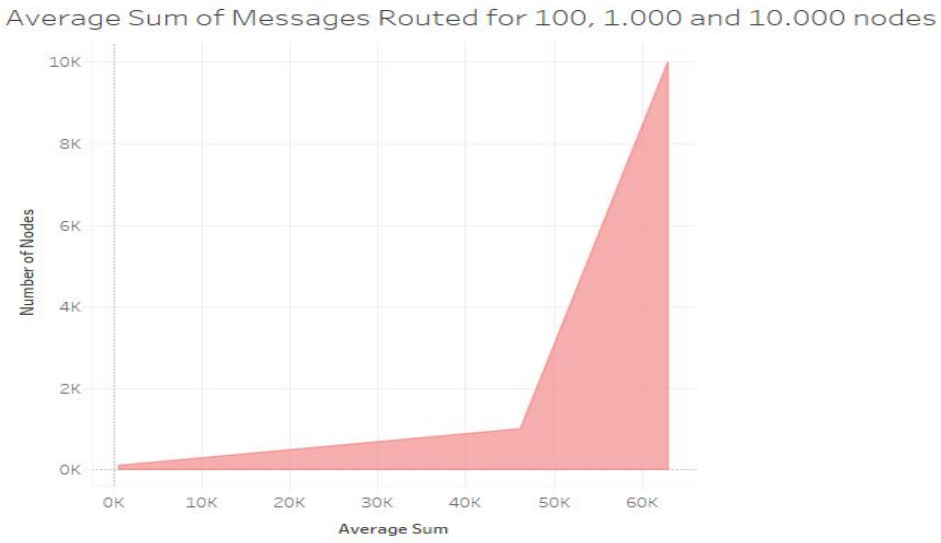
For 100 requests:



For 1.000 requests:



For 10.000 requests:



From the graphs above we can see that the sum of routed messages needed to serve all requests is significantly increased along with the number of nodes. More specifically,

for 100 nodes, it takes 650 route messages for all requests to be served, but for 10000 nodes it takes up to 6.000 messages to serve the same amount of requests.

Along with this, during our experiments, we noticed that a great amount of time is spent on creating the finger tables and searching into them. This leads us to understand that the more nodes present in the Chord system, the more messages are needed to locate a single file. This routing “flood” could cause overhead to the system, as a lot of time and resources are spent on the node communication.

In addition to that, when experimenting with various network sizes and requests amount, we came to the conclusion that the factor that mainly affects the time needed to serve all requests is the amount of nodes in the system and not the requests themselves. This means that a system with 100 nodes serves 10000 requests quicker than a system with 10000 nodes, thus considering the system size should be of a great importance when designing it.

Last but not least, it seems that a relatively empty system may behave worse than a fuller one. This happens since a lot time is spent on creating and searching into a node’s finger table as mentioned above, which ends up containing information for most of the present nodes in the system. This means that there is a lot of time spent on calculating a “smart” finger table when the search within the system can be almost “serial”. Conclusively, it is important for the designer to choose the perfect ring size in order to result with the most effective performance.

Load Balancing

Since the file requests are user-defined, we may end up in a situation that a node has a lot of requests to serve but most of the nodes are relatively stale. In this assignment, this balance problem, is approximated by the popularity distribution used to generate the requests.

The unfair load distributed to each node needs to be taken into consideration in a real system since bottlenecks may arise, that will make it unable to serve incoming requests. It is this exact reason that a load balancing technique shall be introduced. A possible solution is described below.

A possible solution to this problem could be the virtual creation of nodes, that do not really exist on the Chord system. In order to better understand that, you can consider the following scenario:

During a period of monitoring of the file requests arriving to the system, we see that requests 10,11,15 are generated more frequently than others.

We can also see that all these requests are served by the same node with id:17. This results in this node having a great amount of requests to serve. At the same time, there exist nodes that have no pending requests at all..

In order to avoid such situations, we could implement a load balancing protocol that creates virtual nodes, extracting resources from nodes with little to zero load, and assign them to serve a part of the requests of the suffering node. In this way the load is distributed.

This “protocol” could also be expanded to be dynamic, so that the virtual nodes are only created on demand and are removed once they have no pending requests.

Summarization

During the implementation of this assignment, we experimented with the Chord Distributed system and studied its behavior for multiple network sizes an request load.

We can conclude to the following:

- The factor that affects the system’s behavior the most is the number of the maximum possible nodes
- The more nodes in the Chord system, the less files each node is responsible for
- A great amount of time is spent on calculating the finger tables, which greatly affects our implementation’s performance
- The total amount of messages needed to serve all requests are increased along with the number of nodes present in the system
- Having a relatively “empty” chord, with large number of possible nodes (large *m)* but few present nodes, is possibly not a great idea

Concluding, if the communication between the nodes is done in an effective manner, the chord is relatively “full” and a load balancing technique is introduced, then “Chord” can be of a great design choice for a system’s distributed needs.