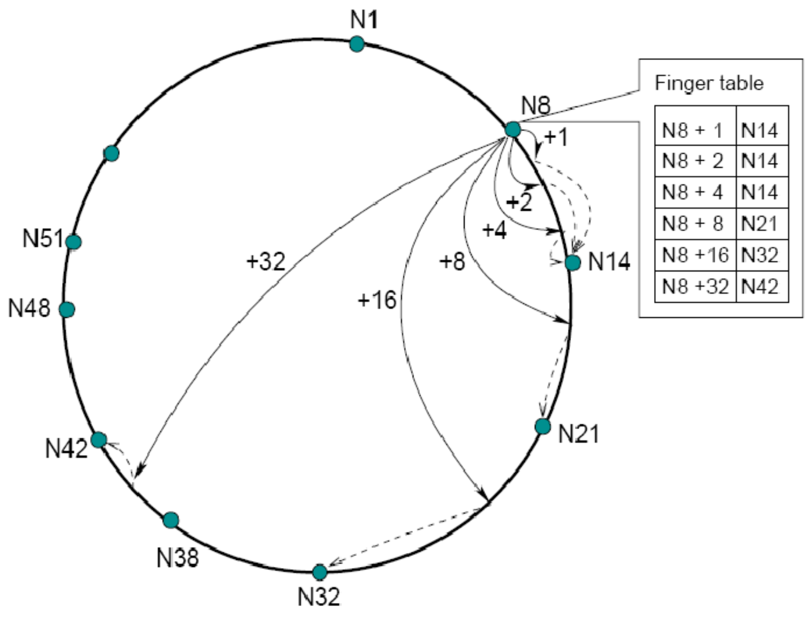
Big Data management

1st Exercise:

Simulate the Chord protocol



N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Alevizopoulou Sofia, dsc17002@uop.gr 2022201704002
Avgeros Giannis, dsc17003@uop.gr 2022201704003

Athens

May 2018

Contents

Table of Figures

At this assignment has been simulated the operation of a distributed file system implemented using the Chord infrastructure. A real-life dataset from the torrent site ThePirateBay has been used. Based on this dataset we are going to create a workload and perform measurements of different parameters.

1. The general idea:

The simulated system contains N different nodes (N is a system parameter given by the user) and each node stores a list of contacts as described in the Chord protocol (successor, predecessor, and fingers). Since it is a simulated and not a real distributed system, communication (i.e., message sending, message receiving) between the different nodes are conducted by writing in an appropriate share memory that is read by each node. At the end of the process, a node n sends a message to node n' by writing the message to a predefined area of n' that is reserved for incoming messages. Each node knows only the nodes that are present in its finger table, the successor and predecessor node.

The simulation is carried out in turns (the number of turns is a system parameter given by the user) where in the beginning of each turn all nodes read the incoming messages, perform the appropriate actions, and write their outgoing messages to the areas of the recipient nodes.

A node can fail or no, it is based on the stabilization protocol. A basic "stabilization" protocol is used to keep nodes' successor pointers up to date, which is sufficient to guarantee correctness of lookups. Those successor pointers are then used to verify and correct finger table entries, which allows these lookups to be fast as well as correct. In our implementation we have considered that nodes do not fail.

In the end we have experimented with different network sizes and different query loads (query load is a system parameter given by the user and it is the number of queries that random nodes will initiate) to get measurements. The experiments were run multiple times in order to take many results and to calculate the average (or median, or mean) of them. Measurements are regarding the number of messages needed to locate a single file, the observed latency (measured in the size of a chain of messages needed to locate a file), and the load of each node (in terms of requests for a file and messages routed).

2. The basic functionality that is implemented for the simulation is:

- creation of N random IP addresses and ports, that will be used by each node
- hashing IP and Port of each node with hash function sha1 (160 bits) will result in the placement of the node at the appropriate position in the Chord ring
- creation of the nodes and the population of their routing information (successor, predecessor, and finger table.
- Download a dataset to use at the simulation (name of movies)
- hashing the name of each movie with hash function sha1 (160 bits) will result in the placement of the specific file at the appropriate node in the Chord ring.
- Each file has a popularity measure that define how popular it is. A file is popular if it is requested from many nodes. The popularity measure has to follow a power-low distribution that has been created.
    - r=powerlaw.rvs(0.75, size=n_f) .
        - powerlaw is a power-function continuous random variable from the spice-stats library of python. The probability density function for powerlaw is: f(x,a)=ax^(a−1) for 0≤x≤1, a>0.
        - powerlaw takes a as a shape parameter.
        - powerlaw is a special case of beta with b=1.

    - popularity=[float(str(x)[:5]) for x in r]
        - Hold the first 5 digits
        - for the popularity we have follow the next logic:

        popularity= round(popularity*(100))
        For example, for a network with 10 nodes the popularity can be 0 or zero, a file will be requested once or never. This is an assumption that we had include in our functionality.

- capability to look up for files in the distributed file system based on the Chord protocol. If the node which initiates the query has the file then the query is initated from another node because this one already has that file
- Load queries that will be used to search files in the chord ring. Different query loads: 100, 1000 and 10000 file requests
- Data summarization measurements:
    - What is the number of messages needed to be proceeded in order a requested file to be found on the ring. In other words, the number of hops for a look up process. We use the median value as describes better the number of hops needed. The median is the value separating the higher half of a data sample, a population, or a probability distribution, from the lower half. It is known as the "middle" value. From the other hand average is not a good approach as influenced by the extreme values (very small or very big). Mode, describes the most frequent value at a set, it is the value that is most likely to be sampled. Mode is not a good approach always as sometimes can be very small or very big. For example, running the simulation, we got:
        - 110 nodes:

        - After 73 turns we got--- HOPS= 6.0 ---AVG HOPS= 21.3150684932 ---MODE HOPS [(4, 14)]
    - 100 nodes:
        - After 2 turns we got HOPS= 6.5 ---AVG HOPS= 6.5 ---MODE HOPS [(10, 1)]
    - 100000 nodes:
        - After 59820 turns we got HOPS= 12.0 ---AVG HOPS= 1566.5057673 ---MODE HOPS [(9, 6312)]
    - 1000 nodes:
        - After 413 turns we got --- HOPS= 7.0 ---AVG HOPS= 98.799031477 ---MODE HOPS [(6, 64)]

    Obviously, the median approach gives better results and verifies the O(logn) complexity of the lookup function. Average gives big hops most of the times. It happens because during the simulation maybe there is a requested file which demands many hops in order to find its location, this case is going to increase the average amount.

- What is the average number of messages exchanged per node for all nodes of the ring. (dht.msg_routed)
- What is the average number of messages exchanged per node, just for the nodes that have routed at least one message.
- What is the average number of message requests per node node for all nodes of the ring. (dht. file_reqs)
- What is the average number of message requests per node, just for the nodes that have at least one request for a file.

All these measurements applied first for each execution and after for all the simulation process.

3. ReadMe:

The implementation has been done with Python. The project consists of 6 python scripts that implement the simulation of the distributed system and one txt file that is used as the dataset (names of movies). The scripts will be described below:

1. Init_func.py

Variables:

- Ids: list with all nodes' ids of the ring, used at the distributed hash table
- allIps: list with ips in order to avoid duplicates
- Tr: dictionary with the node id (comes from hashing the ip and port of the node) and the corresponded IP and port of the node. Used at the distributed hash table, giving info about the IP and port of the node.
    - The above information is the holistic view of the system. It is used to initialize, create and distributed the information to the nodes.

Functions:

- def init_hashing_ring(size,m):
    - initiate the chord ring with the nodes.
    - The number of nodes that will be put on the ring has been given as argument from the user (1st arg).
    - The IP and the port of each node has been created randomly.
    - After the combination of IP and port will be hashed with the hash function sha1 and will return an index which is going to be used as node id. The index will be range [0,(2^160)-1].
      ->ID(node) = hash(IP, Port)
    - Returns a distributed hash table(dht) -> dht[str(ip)+':'+str(port)]=Node(ip,port,id,m). Each entry at the hash table dht is a node on the ring.

- def update_contacts_list(dht,m):

Set the successor, the predecessor and finger table for each node of the ring. As successor set the first node clockwise from the current node and as predecessor the previous node from the current node.

- Key2: is used to set the ip and port for the successor and predecessor node.
- Key: is used to set the ip and port for this node at dht table.

- def fingers(dht,m,i,max_value):

Set the finger table of each node in the chord ring.

- finger[i] = successor (n + 2 ^(i-1))
- For the finger table we have :

finger[i] = successor (n + 2^(m-1)) where n is the current node and m is the number of bits of the identifiers (m=160 bits in our case).
The i^th entry on node n will contain successor ((n+2^(i-1))mode(2^m).
For example for m=6 the finger table of node Id=8 will be:
Finger[0]=successor(N8+2^0)
Finger[1]=successor(N8+2^1)
Finger[2]= successor(N8+2^2)
Finger[3]= successor(N8+2^3)
Finger[4]= successor(N8+2^4)
Finger[5]= successor(N8+2^5)

- if the desired value we are looking for to complete the finger table is bigger than the max node id then we have to divide it and keep the remaining (e.g. 63 max value, 42 node, N42+16 for the 4th item, 66 mod(63+1)=2. Max_value+1 is for the fact that we start counting from 0 up to 2^m-1 and the 2^m should point to the start of the ring.

- def generate_ip():
  Generate random ip with random.randint from python library. Check if this ip has been generated again in order to avoid duplicates.

- def generate_port():
  Generate random port with random.randint from python library.
- def generate_id(ip,port,m):
  generate hash key with hash function sha1 after hashing the IP and port of each node and return an index. The index will be in range [0,(2^160)-1].

2. Share_memory.py

A class with a dictionary and 2 methods:

Class member:

- shr = {}: dictionary that simulates the share memory of each node. By using ip:port, each node can communicate to any other node and write to a memory a node can read.
- In order to improve the speed of the execution, we store the information for each round (each lookup process) based on the node which is responsible to send and check for messages. With that approach, instead of iterating through all the nodes, we specifically call the only send/check messages that should be called.
  - s_ip=-1 #sender ip
  - s_port=-1 #sender port

  - c_ip=-1 #checker ip
  - c_port=-1 #checker port

Methods:

- def send_msg(msg, to):

write a message for a node at the appropriate area that has been set for this node.

- def read_msg(to):

return the context that is saved on share_memory for that node and delete the memory in order to free the space.

- def send(dht)::

Call specifically the node that should send new message.

- def check(dht):

Call specifically the node that should check for new messages.

3. Filenames.txt

A txt file with the name of the movies. The context of the file has downloaded from ThePirateBay.

4. Contact_list.py

A class with the contact list of each node contains: successor, predecessor and the finger table.

5. Node.py:

Node is as a class.

Class memebers:

- file_reqs=dict() : Dictionary containing the message requests per file for each node. It is used to calculate measurements.
- msg_routed: integer, number of a messages a node has exchanged. It is used to calculate measurements.
- succ[node_id,ip:port]: the successor of the node with is IP and port
- pred[node_id,ip:port]: the predecessor of the node with is IP and port
- fin=list() : the finger table of the node
- fin_tr=dict() :dictionary that contains the nodes that are in the finger table with their IP and port →info about {node_id:Ip,Port)
- ip: the ip of the node, it is generated by a random function
- port: the port of the node, it is generated by a random function
- id_: the id of the node on the chord ring. The node has been located at the ring based on this id. This id has been created after hashing the Ip+Port of the node with hash function sha1.
- m: number of bits of the identifiers, M defines the size of finger table, for example, m=160 then each finger table will have 160 nodes.
- Found: Boolean value used for look up functionality
- msg=list(): Generally a list that has the form [file_id,init_node]. Init node contains the information ip:port. Need it to achieve the communication between the node that has the file and the node that initiated the file query. In case the node happens to have the file, it adds a third element in the list, the communication info for the first node to communicate with the file owner.

- recipient: the node id that is going to receive the msg, not the node id that requested first the message
- q_answer: Ip:Port of the node that requested the file first. Initialized when the node has that file, in order to inform the simulation process of read/ send messages to insert the file owner port and IP into the message that will send to initial node. This variable help us to understand what type of msg we have to send (with 2 or 3 args). Set only at the nodes that have the requested file.

Functions:

- def look_up(self,file_):

a node lookup for a specific file with its id (hashing key of its name). Firstly, check whether the file is between the node and his predecessor. In this case the query initiates from another node because this one already has that file (return -1). At that case we check also if the predecessor's node is the last node of the ring and the is the first node of the ring. Otherwise the file isn't located at the current node so the lookup process will be continued with the next node that should search for the file.

- def find_successor(self,file_id,current_node_id,init_node):

There are 3 cases:

- 1st Check whether the file is between the node and its predecessor. We also include the case where predecessor is the last node
  and current node is the first one of the ring. In this case the initial node has the file requested
  - 'self.q_answer'=set the node Ip and port as this node answers to the query

  - self.msg=[file_id,init_node,self.id_] ->set the message with the appropriate format (3 args) as this node answers to the query
- If the file isn't located at that node check whether the file is between the node and its successor. We also include the case where successor is the first node and current node is the last one of the ring.
  - self.msg=[file_id,init_node] ->set the message with the appropriate format (2 args) as this node doesn't answer to the query
  - self.recipient=set the node Ip and port of the successor node as the message will be procced to this node
- the file isn't located at this node neither its successor, find the closest preceding node of this node. The recipient variable will be set with the closest preceding node.
  closest_preceding_node(self,file_id,current_node_id) will be called.
- Messages are set at each node with info about which is the recipient of the message, which asked the file and at which node it is located. These messages, will be checked after with "check_msg" function in order to find which node has the final answer of the query.
- Set the Ip and port of the current node at shr_mem.share_memory.s_port and shr_mem.share_memory.s_Ip. This is the node that is responsible to send messages. With that approach we specifically call the only send

messages that should be called, instead of iterating through all the nodes of the ring. This info is set at each round of the simulation (at each lookup)
    - shr_mem.share_memory.s_ip=self.ip
    - shr_mem.share_memory.s_port=self.port

- <u>def closest_preceding_node(self,file_id,current_node_id) :</u>
  - we tested a variation of the algorithm. Due to the fact that the algorithm had the worst time scenario when the id was smaller than the node id. In case the file id is smaller than current node id we find the max node from the finger table. By that, we speed up the process of searching because the search will be driven to finish the cycle and start searching from nodes with id close to zero.(e.g. current node id is 8 and search for file 5 . Go to max neighbor of 8 and send the message at that node. If the same condition applies to that node also, the it will forward it to its max node id from the finger table and so on. We will reach a node with id less than 5 but bigger from 0 so the search will continue from there much faster than leaving the original algorithm execute as it will proceed to the successor node until it reaches a node greater than zero.

```
if file_id<current_node_id:
        max=-1
            for neighbor in self.fin:
                if neighbor<max:
                        return max
                else:
                        max=neighbor
```

  - return the closest preceding node from the finger table of the node. Starting from m down to 1, search a node with ID bigger than the current node
  - if there is no node at the finger table in range [current_node, file_id], it means that the file is between max node and first node of the hashing ring. So, return the successor of this max value node.

- <u>def check_msg(self):</u>

check every time if the current node has the same ip and port with the ip and port that are related with the current file search. Find the context of the message from the shared memory that is used for the current node.
  - Read the message from shared memory that is allocated for that node
  - If the ip and port that are contained at the node is the ip and port of the current node then, this is the node that has requested the file.
  - If it’s not the node that has requested the file then check if this file has been requested again form that node and increase the “file_reqs” variable of the node. It is used to calculate the measurements at the end of simulation. Find the successor of the node in order to proceed the message

- <u>def send_msg(self):</u>

There are 2 cases:

  - answer to the query of the initial node. This is the last step, now the initial node has the answer where the file is located
  - it's not a msg from the node that requested the file. So, just procced the message to the recipient.
  - At both cases the messages are sent to shared memory that is allocated for them.
  - At both casesset the Ip and port of the current node at shr_mem.share_memoryc_ip and shr_mem.share_memory_c_port. This is the node that is responsible to check for messages. With that approach we specifically call the only check messages that should be called, instead of iterating through all the nodes of the ring. This info is set at each round of the simulation (at each lookup)
    - shr_mem.share_memory.c_ip=
    - shr_mem.share_memory.c_port=

6. File.py

- <u>def read_file(name,n_f):</u>

download a file with movies title from ThePirateBay and save them at txt file "filename.txt"

- <u>def generate_hash_name(name,m):</u>

generate hash key with hash function sha1 after hashing the title of each movie and return an index. The index will be in range [0, (2^160)-1]. For example if m=6 then node id can take values in range [0, (2^160)-1].

- <u>def distribute_files(file_list,dht,m,n_f):</u>

Distribute the files based on their hash key on the chord ring.

  - Each file has a popularity measure that define how popular it is. A file is popular if it is requested from many nodes. The popularity measure has to follow a power-low distribution that has been created.
    - r=powerlaw.rvs(0.50, size=n_f) .
    - popularity=[float(str(x)[:5]) for x in r]
      for the popularity we have follow the next logic:
      popularity= round(popularity*(100))
      This is an assumption that we had include in our functionality.
  - Each entry at the dictionary "files" will contain the hash id of the file (its name hashing by sha1) and its popularity measurement, an integer that will define how many times this file will be requested.
  - Every node can save files with id smaller than its node id. If a file has an id bigger than the bigger node id in the ring, then it will be saved at the first node of the ring. For example, there is a chord ring with m=6 and num_of_nodes=10. The nodes of the ring are: [1, 8, 14, 21, 32, 38, 42, 48, 51, 56] and there is a file with id:60. This file will be saved at the first node of the ring, the node with id:1.

4. Main functionality (main.py)

1. Arguments will be given from the user:
    a. Number of nodes: how many nodes will have the chord ring
    b. Load_query: how many queries will be run
    c. Executions: how times will run the simulation
2. First, the chord ring is initialized as it is described in init_func. init_hashing_ring.
3. set succesor, predeccessor and finger table for each node of the ring as it is described in init_func. update_contacts_list and init_func.fingers
4. Make an instance of share_memory. It will be used from nodes in order to write messages for each node.
5. The number of files is predefined and it is: 342884
6. The number of bit identifiers is predefined to be 160 (from sha1 hash function)
7. A list which contains all the he context of the txt file with movies name (read the txt file and save its context at a list). It is described at files.read_file.
8. Distribute the files, define a popularity for each file with the power law distribution
9. All the below steps will be run for "execution" times, for "load queries" and each loaded query as many times as defined from its"popularity".
10. The simulation will be finished when it will have run as many times as the execution (user argument).
11. Choose a random file that will be searched in the chord ring. Hash its name to produce its key and search at the list which contains all the files that has been distributed at the ring in order to find its popularity. Lookup for this file as many times as the popularity. If for example popularity is 4 then this file will be searched 4 times. Each time will be searched from another random node.
12. If a file has popularity equals to zero then take another random file for searching.
13. Take a random node that will lookup for a random file. If the initial node has the requested file then imitate the query from another random node of the ring.
14. Continue the searching until you find the node that file should have been located.
15. Find from init_func.tr dictionary the IP and the port of the node that is searching for the file.
16. The current node checks its messages until the initial node received a message that contains information about the ip and port of the node where the file is located. If the receiver of the message is the current node then the query responded otherwise it has to be processed at the successor node. The message of a node that hasn't requested the file has format: "file id requested, IP:port of the node that is requested it". On the other hand, if the node has requested the file the its message has format: "file id requested, IP:port of the node that is requested it,IP:Port of the node that the file is locating"
17. A variable has been used to calculate the hops that needed form the first node until the file will be found. General there is a logarithmic growth of lookup costs with number of nodes in network: log(n). The worst-case scenario is m-1 hops.

18. A node can fail or no, that means that chord protocol will respond to the node for its requested file but may this file there isn't at the node this time. At that point we have a failure. At fail cases we don't do anything. We assume that we have no fail.
19. If the file hasn't been found continue sending messages.
20. Calculate the average measurements:

    Per execution

    a. hops: the messages until a file will be found at the ring.
    b. avg_msg: sum all of msg_routed in the ring / number of nodes that have routed at least a message.
    c. avg_msg_all: sum all of msg_routed in the ring / number of all nodes of the ring.
    d. avg_file_reqs: sum of file_reqs at the ring / number of nodes that have requested at least one file.
    e. avg_file_reqs_all: sum of file_reqs at the ring / number of all nodes of the ring.

    Of the final simulation:

    After summing up all previous measurements and divided by the execution times we got the final measurements of the simulation:

    - final_avg_msg=sum(results_avg_msg)/float(simulation_run)
    - final_avg_msg_all=sum(results_avg_msg_all)/float(simulation_run)
    - final_avg_file_reqs=sum(results_avg_file_reqs)/float(simulation_run)
    - final_avg_file_reqs_all=sum(results_avg_file_reqs_all)/float(simulation_run)
    - results_hops=np.median(temp_hops)

    where simulation_run is a counter of how many files have been looked up at the ring totally.

5. Execution of the commands:

The user has to give a set of arguments at the start of simulation. These are: number of nodes that will be located in chord ring, the query load and how many turns will be executed. The simple that will be executed with the user arguments is main.py. A simple run could be:

Python main.py 10 100 2

That means: nodes=10, query_load=100, execution=2 times

6. Load Balancing

Load Balancing: Inside a chord ring we can notice that some nodes are not balanced. This can be happened if at a node are located much more files than other nodes. These nodes will be more popular as the files that are located there will be requested from many nodes. Another case is if a very popular file is located just at one node of the ring so all the other nodes that will request it they have to send messages with this node. All these cases can slow down the performance.

A way to load balancing at the nodes of the ring could be to use Uniform distribution when initializing files id. Supposing, we introduce the possibility that nodes have access to file name- id correlation. At the beginning of file id initialization, we can introduce a Uniform distribution that will span the equally probable id's on the id space. If, for example, m equals to 6, then the uniform distribution will span the ids through the space 0->2^m. Due to the nature of the uniform distribution that will give equal probability for each item, we will ensure that the id's will equally be distributed in the nodes, thus it will be more difficult for a node to have much more files compared to another node.

7. Questions and answers:

1. Number of messages for a networks size: 10, 10^2, 10^3, …

What happens to the number of messages when the network size (number of nodes) increases? Produce a graph showing the number of messages for networks with 10^2,10^3, 10^4, 10^5, … nodes.

Running our simulation for different network sizes we got the following results written on the table. We can see that increasing the network size from 10 to 100 and after to 1000 and so one the hops per lookup duplicated. This can be explained by the fact that the lookup process increases logarithmic, (logarithmic function).

Network size	Queries executed	Hops	Messages per node (only for nodes at request chain)	Messages per node (all nodes of the ring)	File reqs per node (only for nodes at request chain)	File reqs per node (all nodes of the ring)
10	10	3	2,6059523809	1,97	1,60654761905	1,21
100	63	6	3,95154063154	2,1793650794	1,88753469386	1,26476190476
1000	522	9	21,1130884518	15.,278122605	1,6405593159	1,31359386973
10000	3764	11	40,2211080965	23,2237513018	9,83732052127	6,31705908608
100000	59820	12	622,091922829	520,450822904	475,230972013	392,532400371

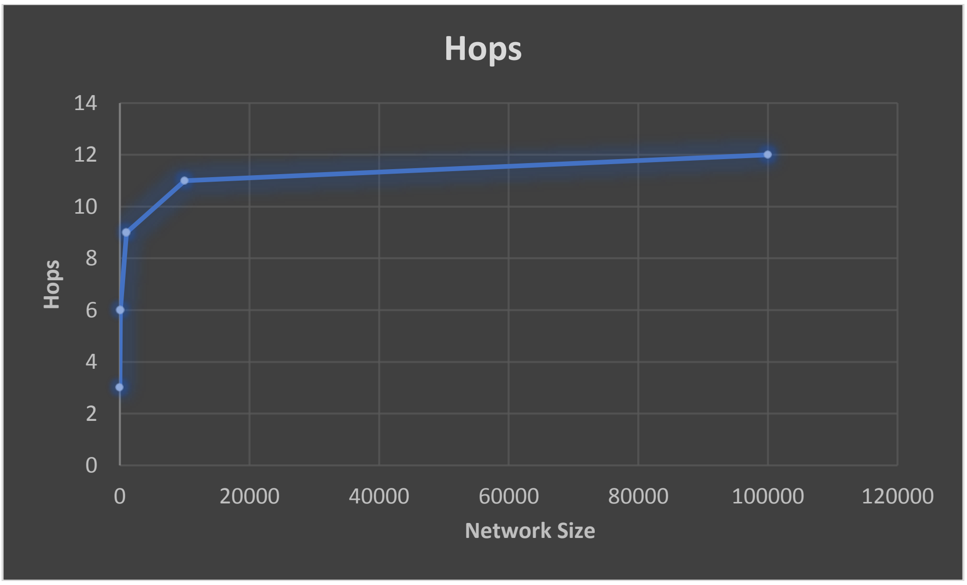


Figure 1:Hops per lookup process for different network size

The above graph visualizes the hops needed for a requested file at different network sizes. As we see the lookup function has O(logn) complexity where n is the network size. This is what we expected from our simulation. The different hops from requests of files to be increased logarithmic based on the number of nodes inside the network.

2. Load (in terms of routing requests)

What is the load (in terms of routing requests) of each node for a given network size (select the maximum network size that delivers results in a reasonable amount of time).

The maximum network size that delivers results in a reasonable time (less than 5 minutes) is that which consists of 10000 nodes. For this network size as it is shown at the above table, the messages routed per node increased as the network size increased. We have calculated 2 metrics, the first one is the average number of messages have been sent from nodes that have routed at least one message. The second metric is the average number of routed messages per all nodes of the chord ring. The second metric (the one for all nodes) is smaller than the other. The network size becomes bigger the number of messages per node increased as more messages needed to be sent in order to locate a file and as a result more nodes take part at the chain of messages.

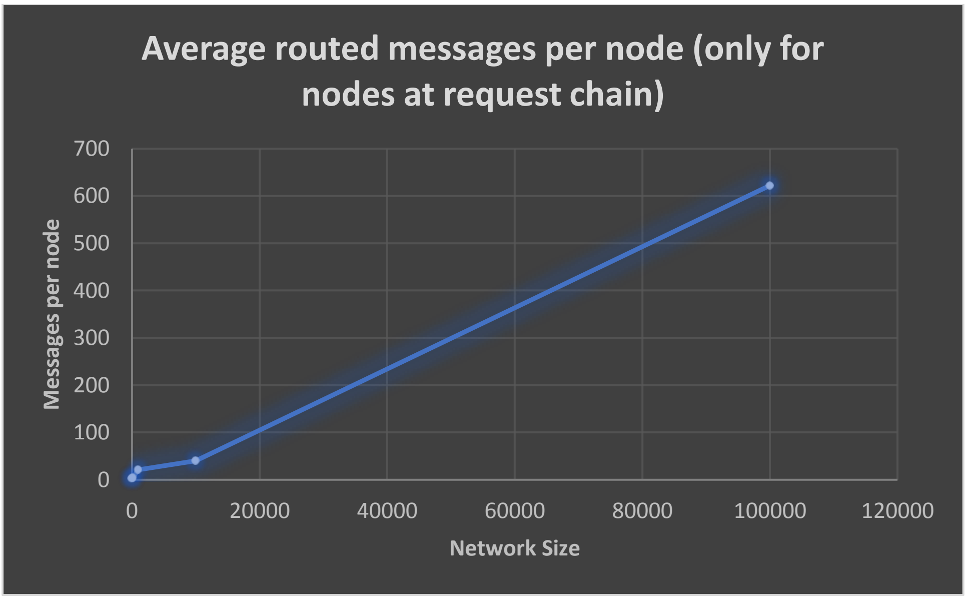


Figure 2:Average messages routed per node for different network size (only for nodes at the request chain)

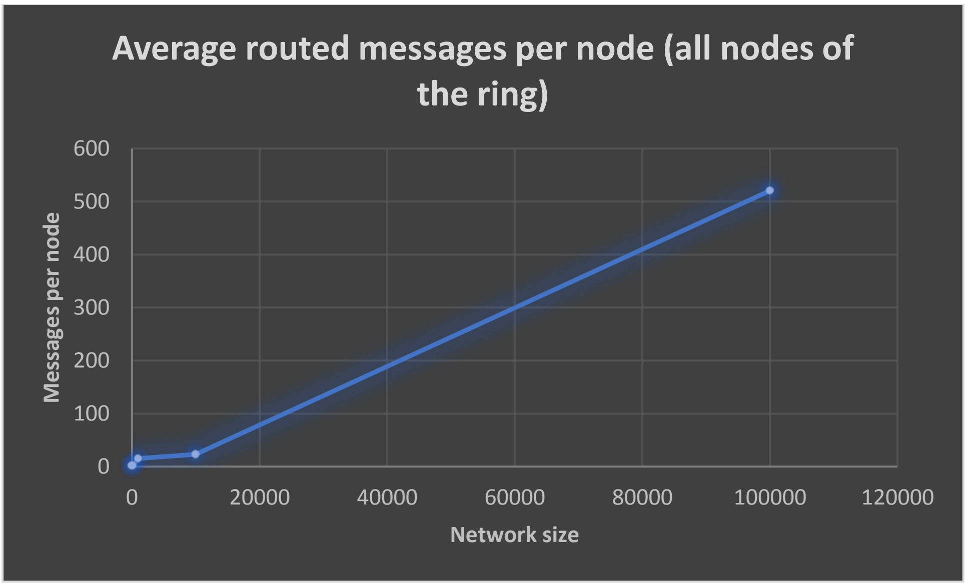


Figure 3:Average messages routed per nod for different network size (all nodes of the ring)

3. Load (in terms of file requests)

What is the load (in terms of file requests) of each node for a given network size (select the maximum network size that delivers results in a reasonable amount of time).

The maximum network size that delivers results in a reasonable time (less than 5 minutes) is that which consists of 10000 nodes. For this network size as it is shown at the above table, the file requests per node increased as the network size increased. We have calculated 2 metrics, the first one is the average number of requests just for the nodes that have at least one requested file. The second metric is the average number of messages per all nodes of the chord ring. The second metric (the one for all nodes) is smaller than the other. The network size becomes bigger the number of messages per node increased as more messages needed to be sent in order to locate a file and as a result more nodes take part at the chain of messages.

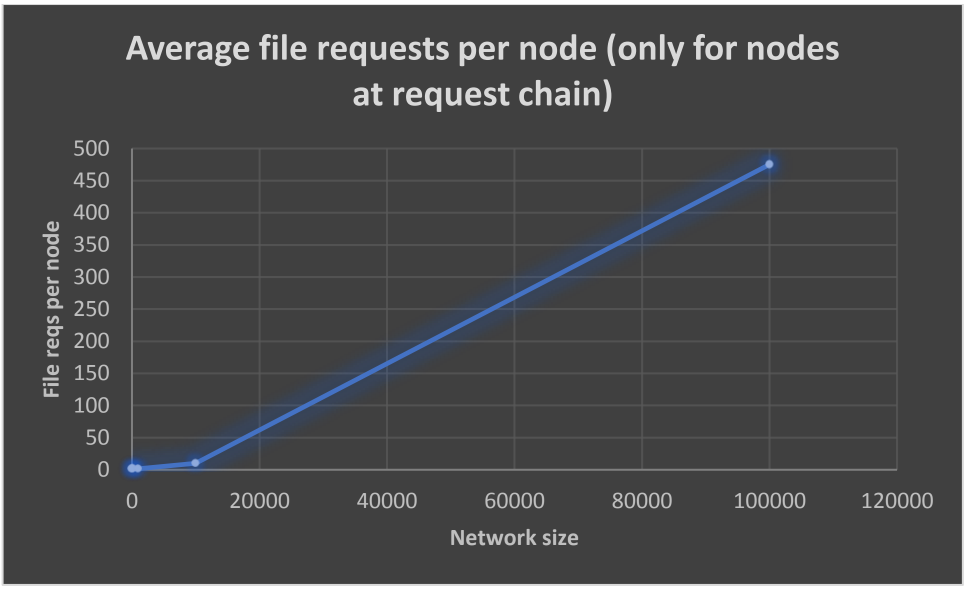


Figure 4:Average file requests per node for different network size (only for nodes at the request chain)

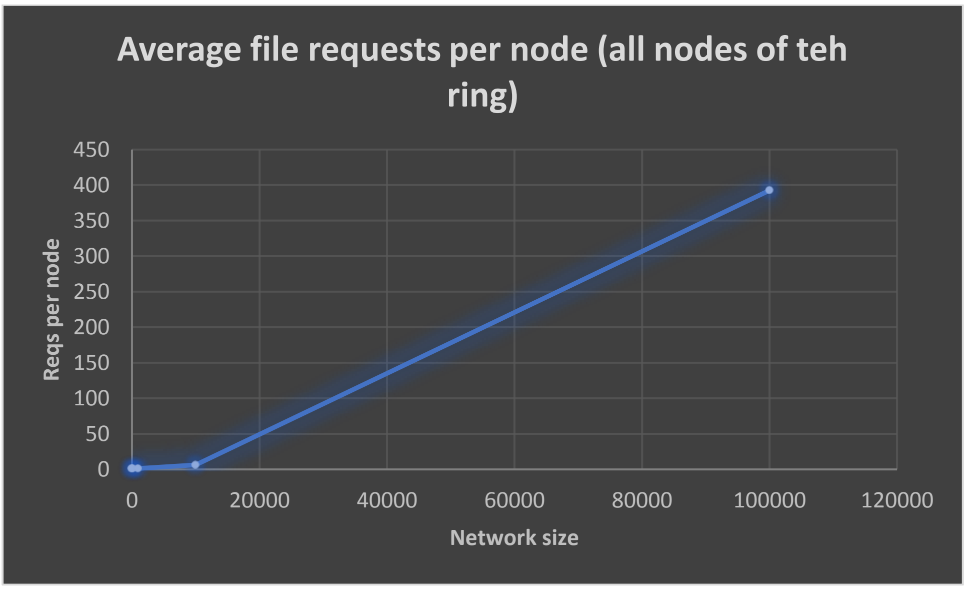


Figure 5:Average file requests per node for different network size (all nodes of the ring)

3. Load queries

Load queries that will be used to search files in the chord ring. Different query loads: 100, 1000 and 10000 file requests for network size 1000 nodes. The “turns: column shows how many files requested at the ring.

Network size	Queries load	Turns	Hops	Messages per node (only for nodes at request chain)	Messages per node (all nodes of the ring)	File reqs per node (only for nodes at request chain)	File reqs per node (all nodes of the ring)
1000	10	465	8	58,6825819227	47,6806473118	12,1358686254	10,1301268817
1000	100	4697	9	350,159396267	347,675015542	309,9219678	308,112650415
1000	1000	43346	9	633,041064604	632,29746055	582,264534335	581,619590689
1000	10000	482591	10	41401	41365,88614	40995,30908977	40950

From the above table we see that the number of hops follows the logarithmic function log(n). In more details the number of hops doesn’t increase when the number of queries increased, it depends only on the network size.

From the other hand we see that when number of queries increases the average number of messages and file requests per node increased, too.

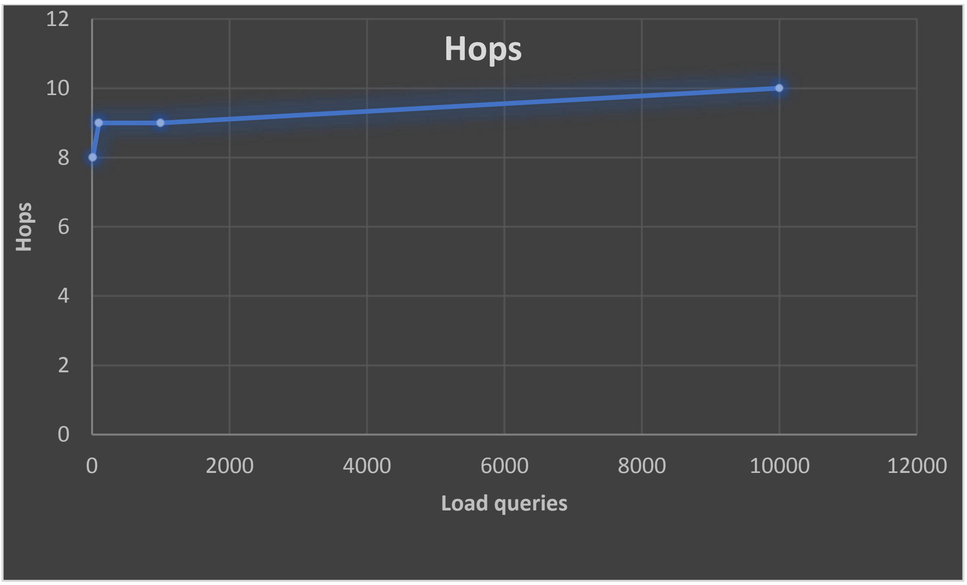


Figure 6:Hops per lookup process for different load queries

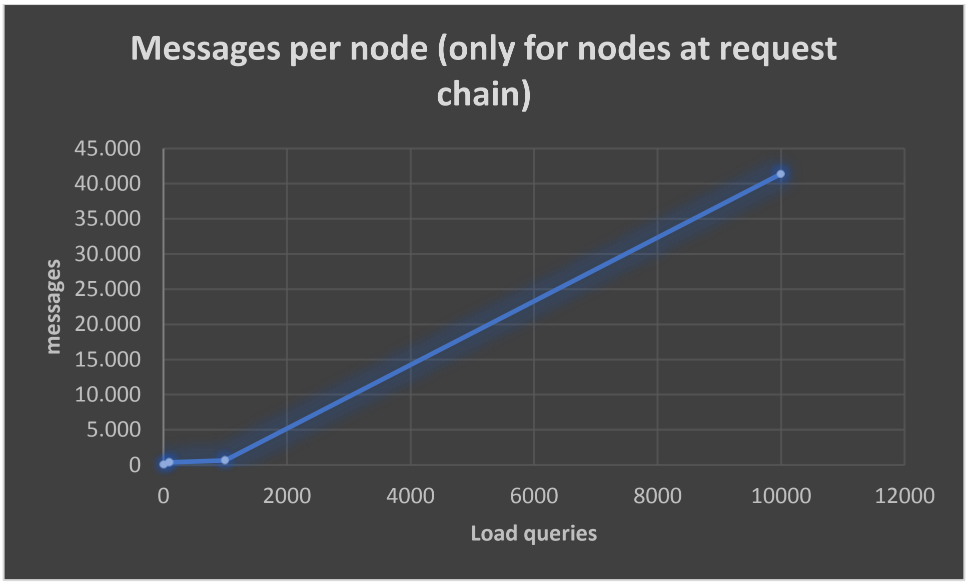


Figure 7: Average messages routed per node for different load queries (only for nodes at the request chain)

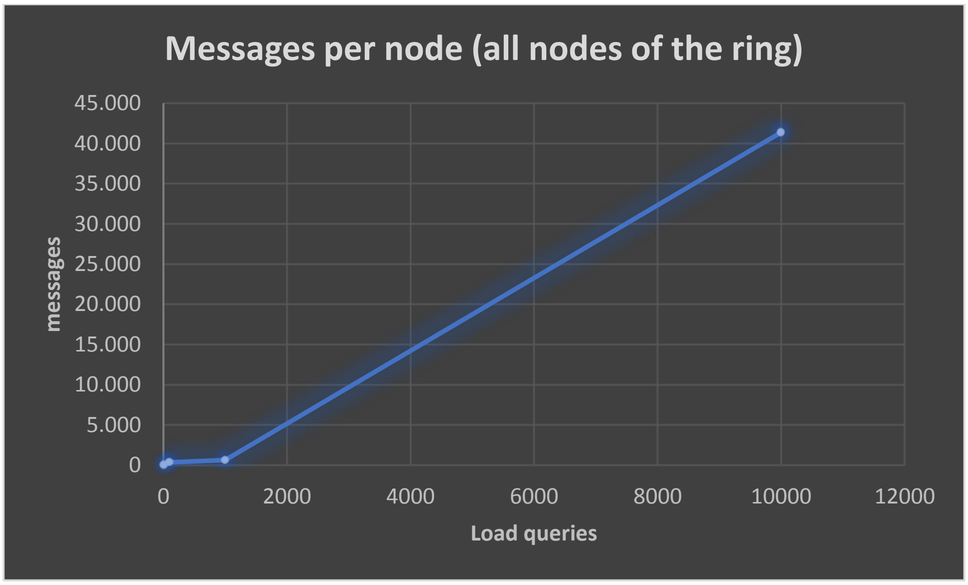


Figure 8:Average messages routed per node for different load queries (all nodes of the ring)

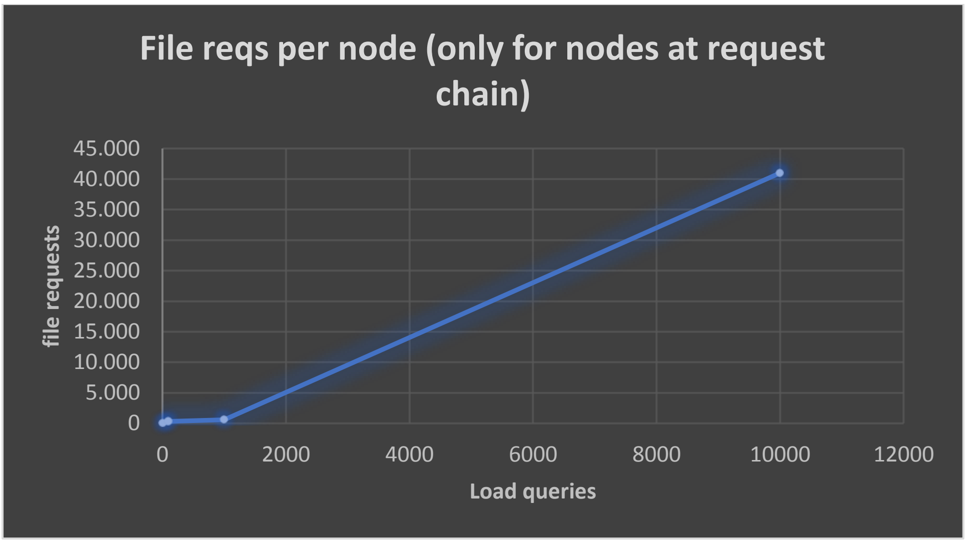


Figure 9:Average file requests per node for different load queries (only for nodes at the request chain)

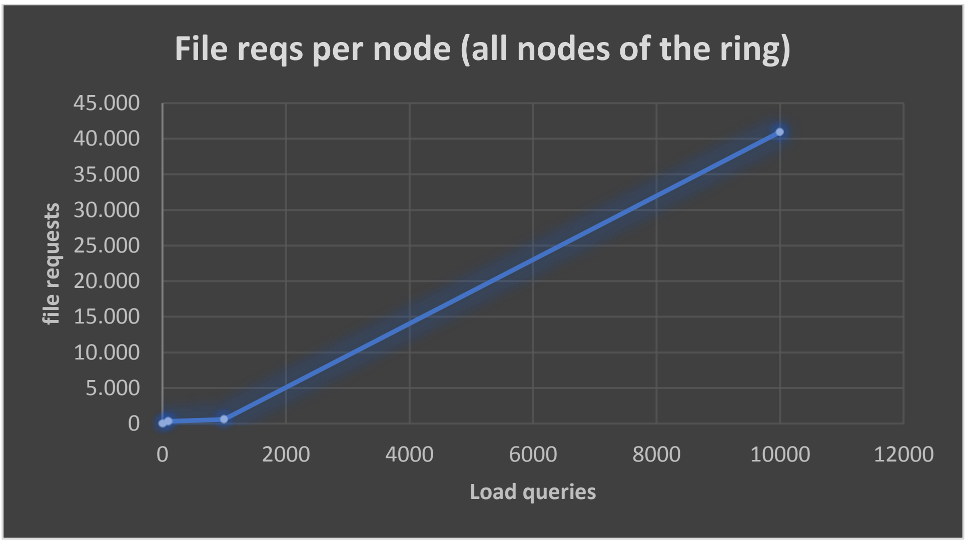


Figure 10:Average file requests per node for different load queries (all nodes of the ring)

4. Algorithm Variation:

We tested our implementation by adding a variation at the lookup functionality (def closest_preceding_node(self,file_id,current_node_id). Due to the fact that the algorithm had the worst time scenario when the id was smaller than the node id. In case the file id is smaller than current node id we find the max node from the finger table. By that, we speed up the process of searching because the search will be driven to finish the cycle and start searching from nodes with id close to zero.(e.g. current node id is 8 and search for file 5 . Go to max neighbor of 8 and send the message at that node. If the same condition applies to that node also, the it will forward it to its max node id from the finger table and so on. We will reach a node with id less than 5 but bigger from 0 so the search will continue from there much faster than leaving the original algorithm execute as it will proceed to the successor node until it reaches a node greater than zero.

```
if file_id<current_node_id:
        max=-1
            for neighbor in self.fin:
                if neighbor<max:
                        return max
                else:
                        max=neighbor
```

Below are shown some executions with and without the variation and the time needed for each execution. At all cases the average messages which send from the nodes of the ring and also the average number of files that requested from each node of the ring are less in case we use the algorithm variation.

➢ Without the variation, basic implantation of the Chord protocol:

Network size	m	Queries load	Turns	Hops	Messages per node (only for nodes at request chain)	Messages per node (all nodes of the ring)	File reqs per node (only for nodes at request chain)	File reqs per node (all nodes of the ring)
10	6	10	100	4	19,1623888889	18,457	13,7997103175	13,296
real 0m6.493s user 0m6.040s sys 0m0.180s								
100	7	10	506	6,5	55,6300674723	54,7162450593	45,6683398184	45,03
real 0m13.772s user 0m9.700s sys 0m0.208s								

➢ With the algorithm variation:

Network size	m	Queries load	Turns	Hops	Messages per node (only for nodes at request chain)	Messages per node (all nodes of the ring)	File reqs per node (only for nodes at request chain)	File reqs per node (all nodes of the ring)
10	6	10	100	4	17,6292380952	16,821	12,9837857143	12,406
Real 0m8.646s user 0m8.156s sys 0m0.240s								
100	7	10	454		17,6407034901	17,2295154185	10,8872240129	10,7135462555
real 0m9.466s user 0m8.608s sys 0m0.228s								

The results from the above table are represented below in these graphs. It is obvious that using the algorithm variation the execution time of the simulation has been reduced a lot as well the number of messages and requested files per node.

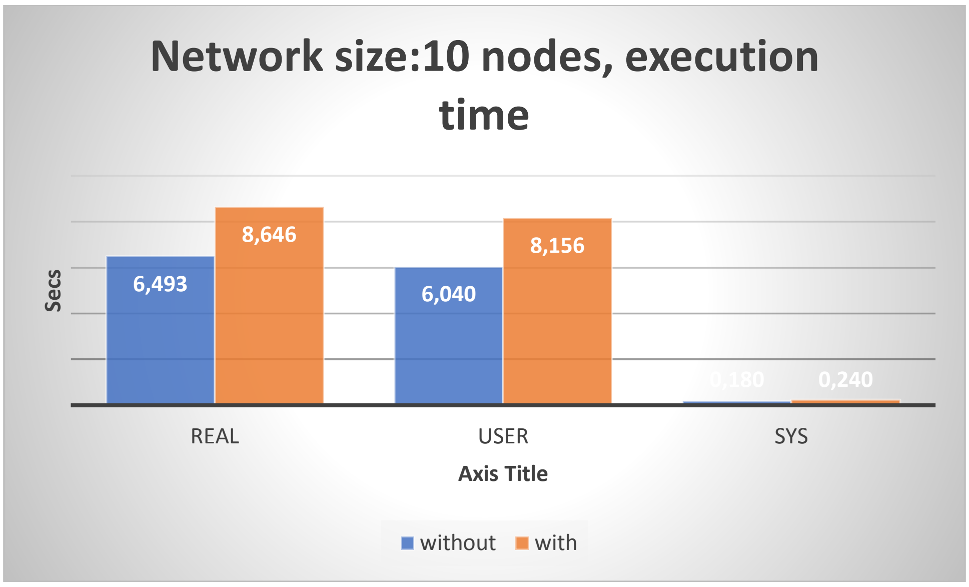


Figure 11:execution Time for Network size:10 nodes

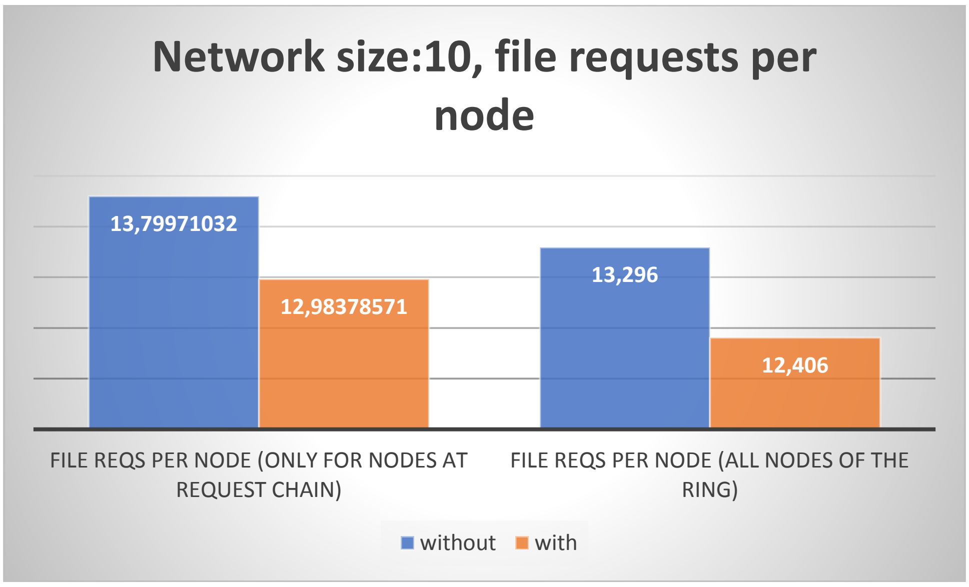


Figure 12:Network size:10-Average file requests per node

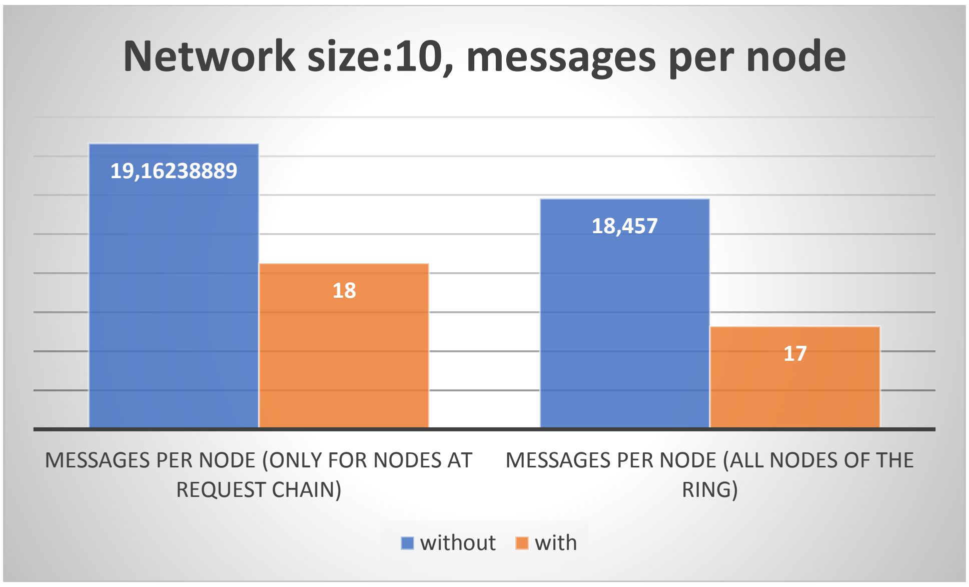


Figure 13:Network size:10-Average messages routed per node

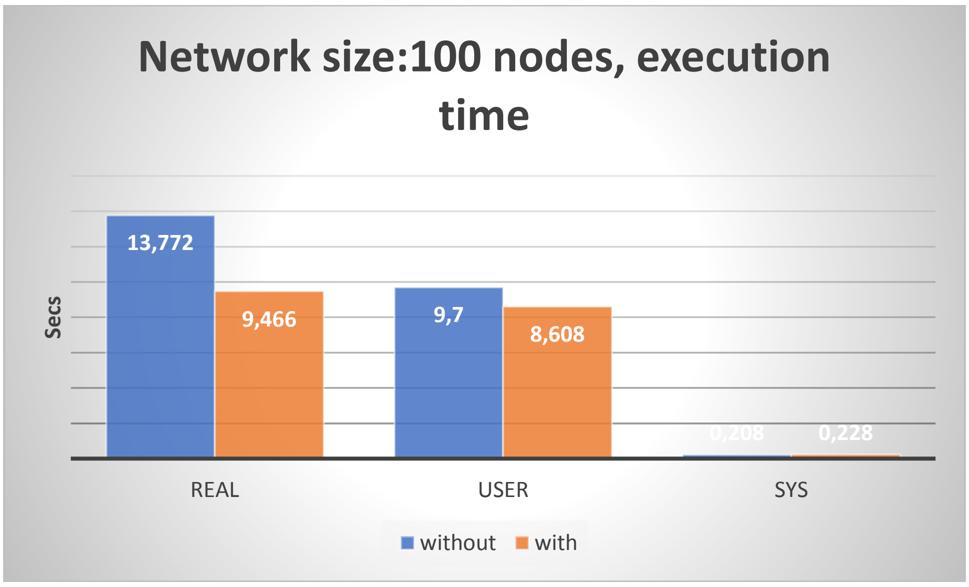


Figure 14: :execution Time for Network size:100 nodes

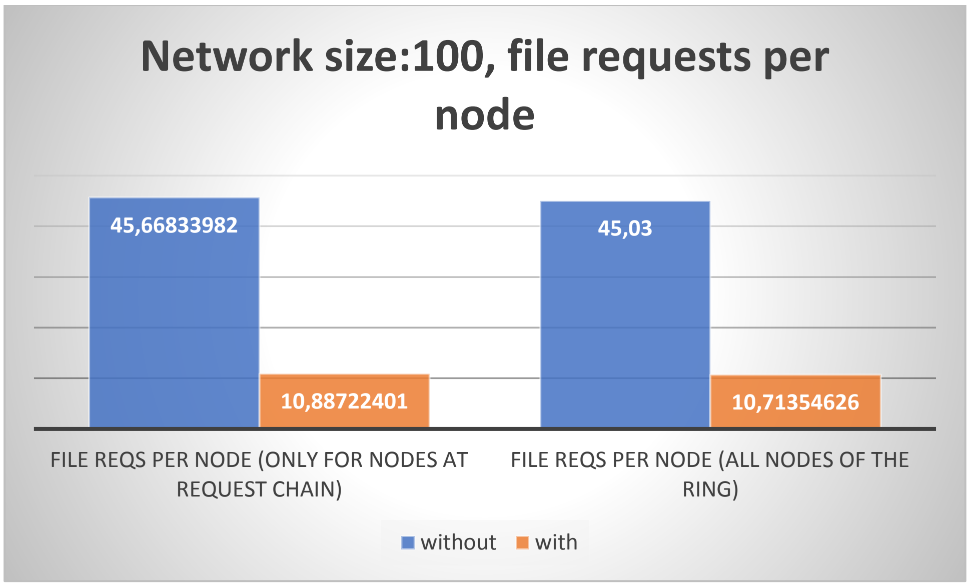


Figure 15:Network size:100-Average file requests per node

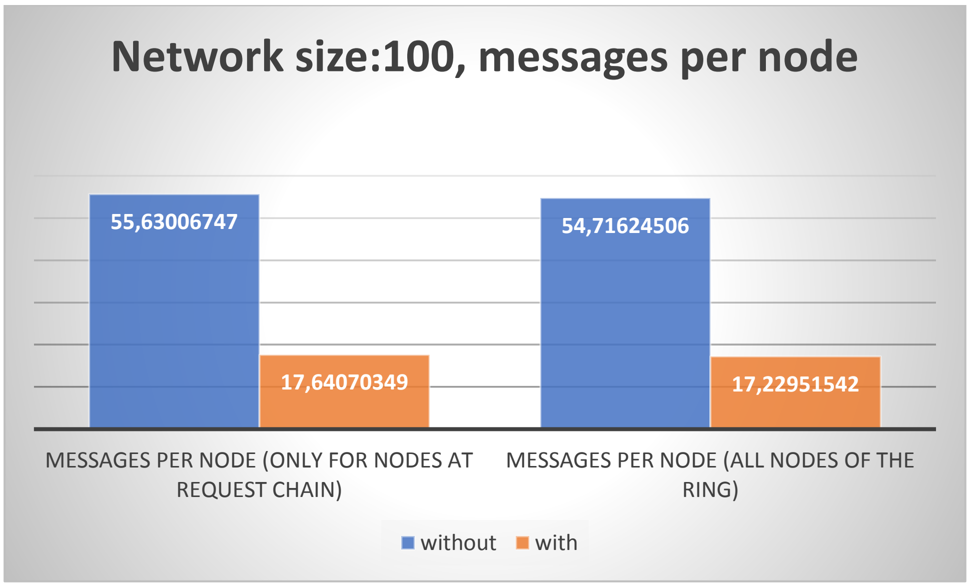


Figure 16:Network size:100-Average messages routed per node

5. Sum up

At this assignment we have implemented a simulation of the chord protocol. A list of files is distributed at the nodes of the ring [0,2^160 -1] (not actually distributed, just used their hash key to locate them on the ring). Using has function sha1(160 bits) each node has got a key by hashing its IP:port. Also, each file has a key by hashing its name by sha1 (160 bits). Each file has a popularity measure that define how popular it is. A file is popular if it is requested from many nodes. The popularity measure follows a power-low distribution that has been created.

An instance of share_memory is used from nodes in order to write messages. Messages of nodes, contain info about the file id which is requested, which node requests it and the ip:port of the node if this node has the file. By checking the messages of node, we can understand if this node has the file requested or no. For this reason, we check the messages of the specific node of the ring and not all the nodes of the ring in order to make the process faster. If the receiver of the message is the current node then the query responded otherwise it has to be processed at the successor node.

The look up functionality has been implemented like there is no fail during the lookup. We assume that every file that is requested there is at the ring. Each file will be requested by a random node of the ring. If the initial node has the requested file then it will be requested form another node. In the end we calculated some metrics about the number of messages and requests needed for a query execution. We see that when the network is big the average numbers of messages and requests per node increased, whereas the overall numbers of hops required to locate the file at the ring has a logarithmic growth of lookup costs based on the network size.

Last but not least, using an algorithm variation while searching for the closest preceding node in case that the requested file id was smaller than the node id we manage to reduce the execution time of the simulation as less hops are needed. This is the worst-case scenario for the complexity of the simulation.