Chord, Repast Approach

Mattia Milani*, Mat. number 203392; Andrea Fregonese†, Mat. number 205511; Francesco Torella‡, Mat. number 205752;
Department of Engineering and Computer Science, University of Trento
Trento
Email: *mattia.milani@studenti.unitn.it, †andrea.fregonese@studenti.unitn.it, ‡francesco.torella@studenti.unitn.it,

***Abstract*—This document present our implementation of the Chord protocol [1] in Repast[1] and how to use the software to do some experiments on your own. All the software presented is also public for future modifications[2]. The protocol is presented more deeply in [1], and we assume the reader has a general knowledge of how the protocol works and components roles. Our main goal is the replication of the protocol in an agent based environment that allow us to do some experiments.**

I. Introduction

Chord is a distributed hash table protocol. The main goal of the protocol is to assign key-value pairs to nodes in a peer to peer system in a uniform way. Chord provides only two operation, the insertion of an object and the lookup for a given objects. To map keys to nodes Chord uses consistent hashing, in our case the hash function SHA1. The nodes are organized in a ring topology and the ids of the nodes are an hash of their identifiers, in the experimental setting we use incremental ids, but in a real case they can be the IP and the port number of the nodes. When the user want to insert an element the Chord protocol will assign to it a key, given by the SHA1 of the object, and the object will be assigned to the node in charge of it, that is the successor of the key. Due to the properties of the hash functions the keys will be assign to nodes in a balanced way.

The protocol is very efficient, in a N it has a space complexity of $O(logN)$ routing information per node, and the cost of lookup/insertion is also $O(logN)$. Furthermore Chord allows node to join and leave the network in a dynamic way and also in a concurrent environment. The protocol works well also if the routing information are partially correct and it have methods that spread the information of new nodes or leaved nodes very fast. Chord one of the base peer to peer protocol and it can be used in a lot of areas.

In our implementation we replicated the Chord protocol using the Repast environment and we made some simulations. We created a Super node that allows us to communicate with the nodes of the network to test Chord in a lot of different scenarios. The Super node can schedule the execution of the main functions of the protocol like join, leave, insert and lookup in specific nodes. The details of the implementation are described in Section III

Our implementation includes a graphic representation of the system that shows the ring topology and the exchange of messages. We run our simulation for testing some scenarios of the protocol and we also provide different views to show different point of view of the executions. The details of the graphic simulation and how to use it are described in Section IV

Furthermore we collect some data from executions of the protocol and we made some plots to prove the correctness of our implementation, details are described in Section V.

II. Goals

Our goals as described in the previous section were two; The first one is to design and implement all the functionalities described in Chord paper [1] and built a system that is capable of maintain, consistently, a list of nodes and a list of resources that could be requested by the users. The system should be able to remain consistent also if more than one node join or leave the network simultaneously. Failure are managed by this protocol and the ring is kept consistent in a normal behaviour.

Our second goal is: once the system is up and running, we should be able to verify all the particular properties that make this protocol so smart and useful. For this purpose we design some specific tests to show that the nodes are, for example, equally distributed among the circle of the possible id space. Second test shows the number of keys stored in each node is on average proportional with the number of active nodes in the network. Last test we are going to present in this report is designed to show that the average path length is logarithmic with respect to the number of nodes. This property is extremely important to improve the scalability of the protocol.

The improvement that Repast offer to us is the possibility to show, at run time, the messages exchanged by the nodes in the system. This provides a better understanding of the algorithm and allows also experimentation with different parameters. With this possibility we want also to show an execution of the algorithm that demonstrate all the possible messages in the network.

III. Repast Development

Repast framework is used in this assignment in order to provide a simple and easily comprehensible visualization of the algorithm in real time. In this section we would like to describe the structure of the project and also the choices that we made in order to implement the different functionalities of the Chord protocol.

Our simulator is composed by actors that generate events and communicate with other components by the exchanging of messages. We decided to create three main agents:

[1]https://repast.github.io/

[2]https://github.com/franzunitn/DS2_Chord

- **Network Builder**: creates all the network needed for visualization
- **Super Node**: control the simulation step by step from a level above
- **Node**: multiple actors that receive events from the super node and handle these by reacting in order to accomplish some action

The first actor we are going to describe is the lowest in the hierarchy: the Node. Every node has an id composed by 160 bits, because the SHA1 [2] hash function is used in order to obtain an unique id. The space of all possible ids is thus 2^{160}, the cryptographic hash function SHA1 has the property of distribute *n* random ids equally in all the possible space id. This means that, on average, if *n* nodes are insert in the network than the distance measured in number of ids from one node and its successor is equal for all the nodes. This property is the reason why SHA1 was chosen, indeed is not simple to find a function that has the same behaviour. This concept is described better in section V.

A Node has also to store objects, but to know which objects belongs to a node, again, we use an id of 160 bits also for the item. With the same hash function we find where in the circle an object has to be positioned and the node responsible for that object is the one which id's succeed the object id. In other words every node is responsible for the range of ids between his predecessor in the ring and him.

A Node keep also some useful information to find nodes and objects in the ring. Indeed, it keeps a pointer to his direct successor and predecessor node, but also keeps a table of references called: Finger table. Without this table the only way to send a message or an object to a specific node is to traverse all the ring in between with a computational cost of $O(N)$.

In figure 1 it is possible to see the structure of the table, that is composed by two columns, for every row there is an index obtained by summing to the id of the node a power of two, form 2^0 to 2^{159}. For each of these index his corresponding successor node in the ring is stored. With this mechanism one node can half the distance between him and the target at each step and this is why the overall computational cost of reach a destination node in Chord protocol is logarithmic with respect to the number of node in the ring. This technique has also a space complexity of $O(LogN)$.

Summing up a node has to keep a set of objects (keys) and a set of neighbours, to keep this set consistent has to implement six functionalities. the first function we implemented is the join. If a node n want to join the ring has to know a node n_i that is already in the network, so n can ask to n_i to find his successor, if it does not know directly the successor of n it can forward the request to the closest preceding node of n_i present in his finger table.

In order to better simulate a distributed system all the functionality of a node are implemented as message passing procedure, so every method call is divided in three phases, the invocation of the event on the node target in the specific case of a join we call the method 'join' on the node n which sends a message to node n_i that if know the successor send a reply message back to n in the other case forward the message to a node n_j until it finds the right successor.

Once the node has joined the ring it has to keep a two consistent pointers respectively to its predecessor and successor node in the ring, this is guaranteed by two methods:

- **Stabilize**: this procedure consists in a node n that sends a request to the successor node n_i asking for his predecessor, if his predecessor is between node n and n_i means that a new node n_j has joined in the space between them. So node n has to set his successor to n_j and notify it that n is its new predecessor;
- **Check predecessor**: this procedure has the only task to detect if the predecessor of a node is failed. A message is sent periodically to the predecessor node and if it does not reply in time the predecessor is considered as failed and set to null. After a while the stabilize procedure will set the new predecessor.

Once a node has join the ring and has his predecessor and successor consistent is enough to guarantee that the protocols works, but a useful improvement as told before is given by the creation of a finger table for every node.

Concurrently with stabilize and check predecessor procedure every node create a finger table and keep it consistent with a procedure called fix finger. This procedure is called periodically and at each step every node update a row in its finger table, calculating the index as explained before and find the corresponding successor node. Again this procedure is done by message passing so every time the procedure is called a message is generated and sent to the nearest node of the index we want to find the successor (if the node does not know the successor directly) and once the node responsible for that index receive the request, has to reply with another message directly to the source of the request.

Now a node can join the ring and stabilize his successor and predecessor pointer and it is also able to find the closest preceding node in at most $O(\log n)$ steps, so every node has all the requirements to insert and search for objects in the network that are the two functions we are going to explain.

In order to insert an object, and from now on we refer to an object only as a key, a node has to find the node responsible of that key in the ring. This could be done just finding the closest preceding node of that key, if the node that receive the request is not directly responsible for that key. Once the correct successor node of that key has been found the insertion is done, the node add the key to the list of his keys. If a node receive a request of lookup for a specific key than first check if that key is in his competence range and if not just forward the request to the closest preceding node based on his finger table.

Last two procedure contained in a node are two way a node can leave the ring. A node may want to leave the ring, and in order to do it has to notify his predecessor and successor, and send all his assigned objects (keys) to his successor that from now on will be responsible for them. But if more than one node that are consecutive in the ring leave at the same time a node will transfer his object list to a node that has already leave, this could generate some losses and in order to avoid this behaviour we decide that when a node leave the network it will remain active for a short amount of time and forward possible incoming request of transfer object list to his successor (or predecessor). The amount of time a node remain active depends on how many concurrent consecutive leave the system will be capable of tolerate this could be a parameter of the system, but the probability of two consecutive nodes that leave together is extremely low so depends on the characteristic of the system.

The second case where a node leave the ring is because it had failed, in this case all the object in his object list are lost and simply does not respond to any request anymore, in this particular implementation it is impossible to recover to this state. The other node has to detect the fail and react to this, but this is guarantee from the stabilize and check predecessor procedure early discussed.

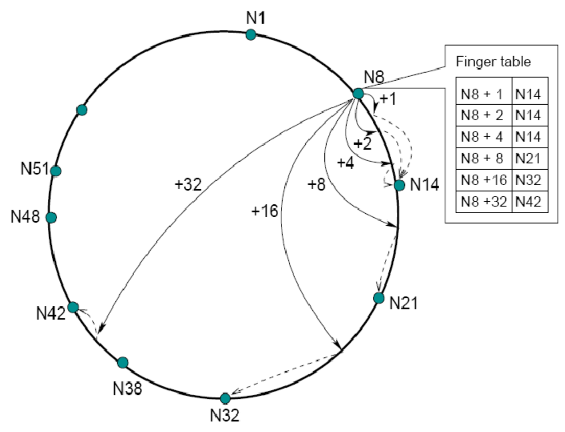
The main actor Node is complete and implement all the functionalities described above, but in order to test and run the simulation we need an actor that tells to a specific node actor what to do and when, this actor in our system is called Super node. There is only one Super node for each simulation and is structured as a set of tests, that could be of two type:

- **Unit testing**: These are a set of test in which we test a single functionality at a time, and for every functionality we test different cases in which different aspects are taken into account. For example in the case of a join of a new node we test normal join in different order as well as concurrent join. Or for example we test the fact that if we insert a key in the ring it must be still in the ring unless the node that store the key has fail.
- **Test scenario**: In order to obtain particular results explained better in Section V we have to create particular sequence of action in the network, in order to accomplish this task we design a method that contain an entire execution of the Chord protocol, from the joining procedure and in which order the nodes join the ring and also how the insertion of the keys is made as well as the lookup procedure, until the simulation end and the data are extracted and use to do some analysis.

The last actor in our simulation is the Network builder, its main task is to collect the parameter from the graphic interface and create the maximum number of node in the network, that is one parameter chosen by the user, after creating all the node the network builder display the ring based on the id of the node.

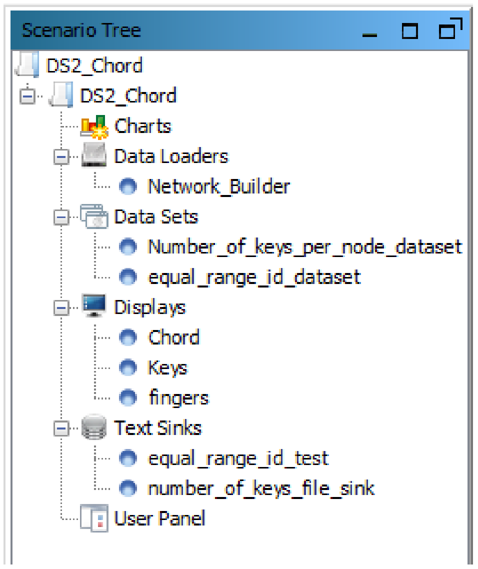
Could be that two node appear closer, and this is because the nodes are displayed based on the amount of ids they are responsible, so the more ids a node has in charge the more space there is between him and his predecessor.

Once the list of node is created and displayed, and once that all the node has a reference to their super node the task of the Network builder is concluded and from now on the simulation is guided only from the super node. In this section we briefly describe the actors contained in the simulator and for every actor we described the structure and the interaction that it have



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

Figure 1. An example of Finger Table of a Node



```
DS2_Chord
DS2_Chord
    Charts
    Data Loaders
        Network_Builder
    Data Sets
        Number_of_keys_per_node_dataset
        equal_range_id_dataset
    Displays
        Chord
        Keys
        fingers
    Text Sinks
        equal_range_id_test
        number_of_keys_file_sink
    User Panel
```

Figure 2. Scenario tree of the simulation

with the other actors in the system. In the next section we introduce the experiment made and their environment.

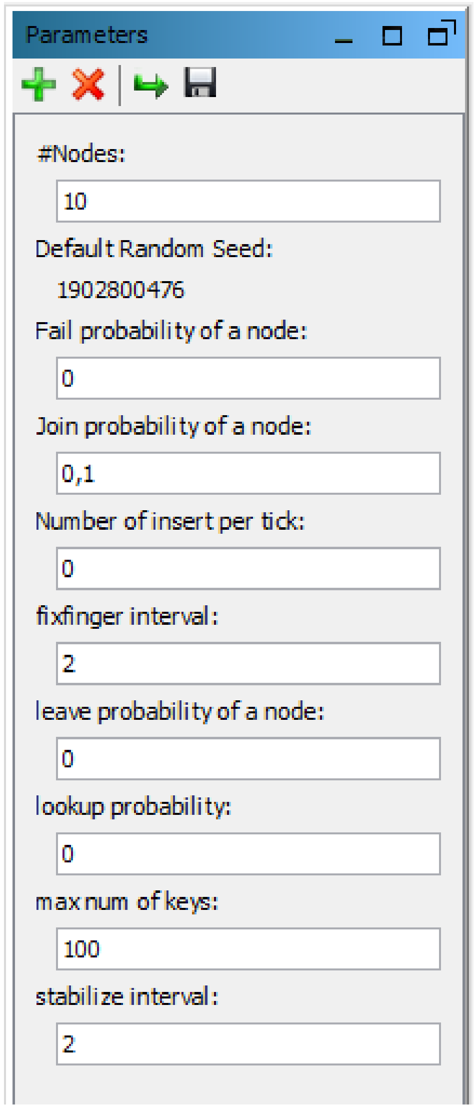
IV. How to

When we run our simulator Repast shows us the usual graphic interface where we can run some experiments with the Chord protocol.

A. *Scenario tree*

The first thing that we see, once we run the simulation is the Scenario tree presented in figure 2. The Scenario tree shows the classes that compose our simulation, we provide a brief description of all of these:

- Data Loaders: they are the main component of the simulation in our case we have only the Network_builder, it is in charge of creating the context and putting the agents in the space.
- Data Sets: we used the data sets to make some analysis on the protocol, they provide us some logs files that describe



Parameters

#Nodes:

10

Default Random Seed:

1902800476

Fail probability of a node:

0

Join probability of a node:

0,1

Number of insert per tick:

0

fixfinger interval:

2

leave probability of a node:

0

lookup probability:

0

max num of keys:

100

stabilize interval:

2

Figure 3. Parameters of the simulation

the behaviour of the protocol, these data set are created on demand during each simulation.

- Displays: they are the views in witch we put the agents to show different aspects of the simulation, they are described in detail in subsection IV-C.
- Text Sinks: they are used to write the files of the data sets.

B. Parameters

The other tab, showed in Figure 3, of the Repast simulation allows the user to set some parameters of the simulation, here we can tune the behaviour of the nodes in terms of probability, at each tick the nodes will take a decision following the probability parameters. we report here the most important:

- *#Nodes*: here is possible to define the number of nodes in the simulation, initially all nodes are inactive and they will join the network during the execution;

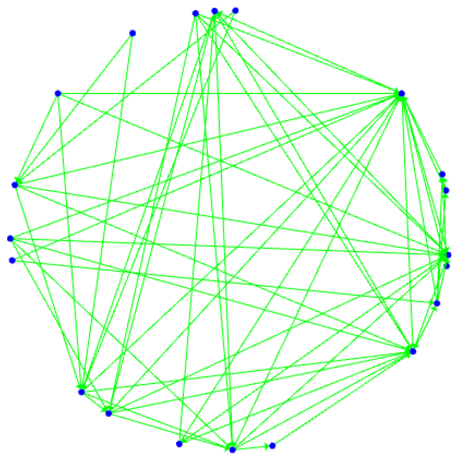


Figure 4. Finger tables pointers with 20 nodes

- *Probability of join*: the probability that a node joins the network in the next tick, this affect only inactive nodes;
- *Number of insert per tick*: the number of keys that are inserted in the network on each tick, the insertions will start when there are enough node that have already joined the network;
- *Lookup probability*: is the probability that a random key, already inserted, will be search by a random node.

C. Displays

We organize the simulation in three distinct displays, each one shows a different prospective of Chord.

node agent clearly is always present on each display and it changes color based on its status.

- Gray: the node is inactive, at the beginning of the simulation all nodes are in this status.
- Blue: the node is part of the network, a node becomes blue when it has completed successfully the join procedure.
- Red: the node is failed and it will not send or answer to any message anymore.

All nodes take place on the space based on its id as described in section III.

Now we describe, for each display, the different components and the semantic of each one.

- *Fingers*: this display is composed by a network that represent the Finger table pointers of each node, an example is showed in Figure 4. This view provide us an overview of the routing knowledge that each node have during the execution (green arrows);
- *Chord*: we create this display to show the exchange of messages that nodes have to send to maintain updated the network knowledge. In this display we have three distinct network one for each type of message of the protocol. The first one is the join network that represents the message exchange that the nodes do to make a new participant able to join the network, this network is colored in green. The second is the stabilize network that shows the exchange

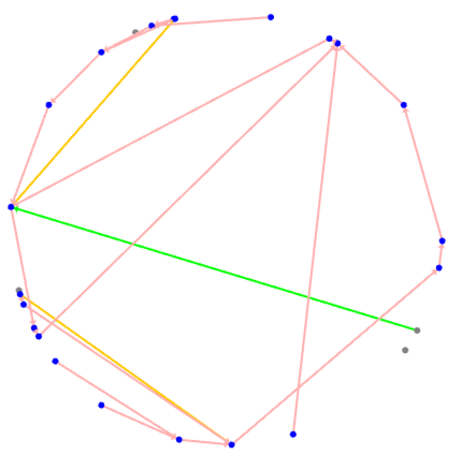


Figure 5. Exchange of messages with 25 nodes. The node lower left sends a join message to join the network.

of stabilize messages, it is colored in orange. The last one is the notify network that shows the exchange of notify messages and messages are colored in pink. This display provide us a simply way to understand how the protocol manage the dynamics and concurrent operations, with different numbers of nodes, also in terms of number of messages exchanged. An example of this display is showed in Figure 5.

- Keys: This display shows the exchange of information during the insertion and the lookup of the keys. The display is composed by one network for the insert procedure colored in green, a network for the lookup procedure colored in red and finally a network that represents the reply with the object to the node that initiated the lookup of a certain key and it is colored in cyan. Moreover when a lookup reaches the right node key-handler, the reliable node of that key becomes green for a couple of ticks. Similar for the lookup procedure, when the right node receives the request for the key of witch it is in charge it becomes yellow. An example of this display is showed in figure 6.

D. Simple experiment

We can set the parameters to show different aspects of the execution, for example if we leave the fail probability to 0 no node will fail during the execution. To launch a simple simulation we can set the number of insert per tick to 2 and the lookup probability to 0.2 and leave all the other parameters as default. It is advisable to set a tick delay in the run option to make the simulation proceed slowly. At this point we are ready to launch the experiment. During the execution we can see some interesting scenarios switching from a display to another. We will see the nodes joining the network in the chord display, the insertion and lookup of keys in the keys display and the evolution of the Finger tables in the finger display.

This section introduced the user to the main aspects of the simulator, how to interpret the different views and how

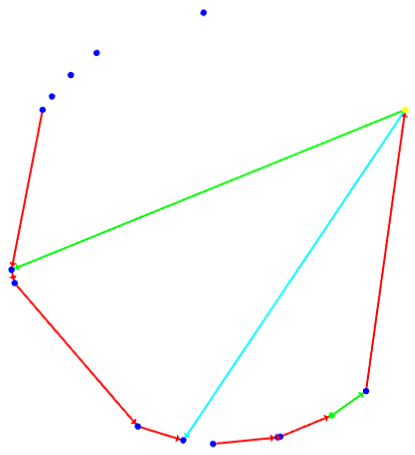


Figure 6. Exchange of messages for the insertion and lookup with 15 nodes. The green node have just receive a key to insert. The node upper right replies to the source of the lookup for a key.

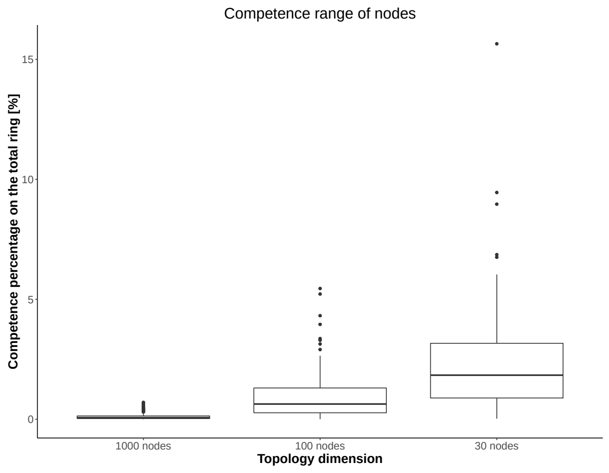


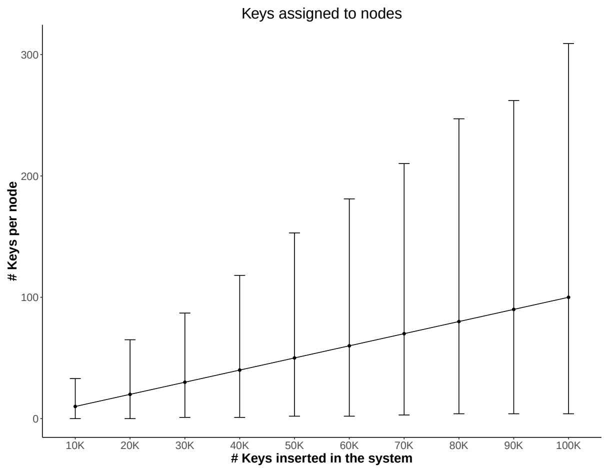
Figure 7. Competence range variance dependence on the number of nodes in the network

to launch a simple experiment. Now we will show some experimental results obtained with different parameters and some plots that represent the behaviour of the protocol.

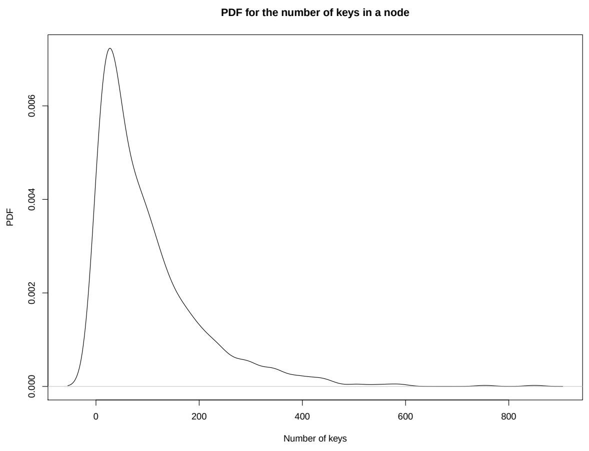
V. Results

For our experiments we designed an environment adapt to different situations. It's possible to change the number of nodes in the network and other probability factors that affects the evolution of the network. The results showed in this section are obtained and plotted thanks to the *Text Sinks* of repast.

Our first test relies on the capability of **SHA1** to split equally the space of the ring among all the nodes. This has been proven to be true for high number of nodes, like is described in Figure 7. Is possible to see that for an high number of nodes, the fluctuation around the mean are quietly not influenced, but as soon as we start reducing the number of nodes we have a bigger variance. Like described in Figure 7, in a network with 100 nodes we have the main portion of nodes that have

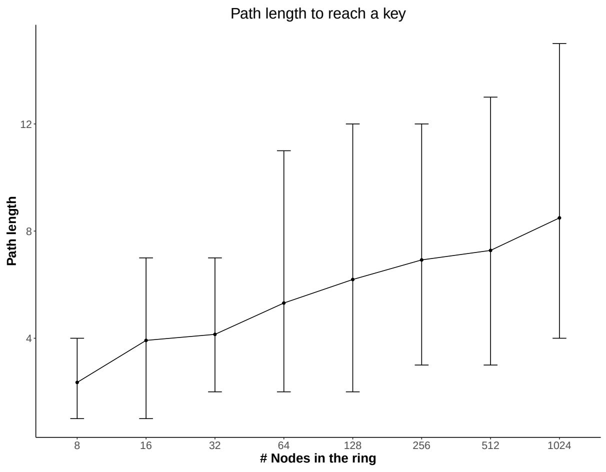


(a) Number of keys controlled by a node in a network of 1000 nodes, from 10 000 up to 100 000 keys increasing at each step of 10 000, confidence levels from 5% up to 95%

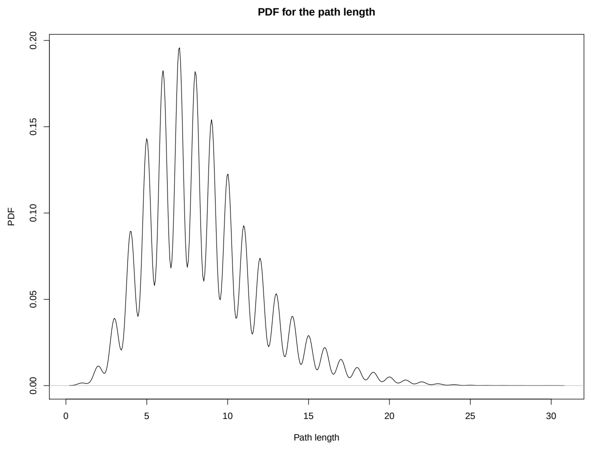


(b) PDF of the number of keys in a network of 1000 nodes with 100 000 keys inserted

Figure 8. How a network of 1000 nodes react to the assignement of keys



(a) Path length to lookup for a key in a network of different dimension, condifence interval from 5% up to 95%



(b) PDF of the path length for the lookup operation in a network of 1024 nodes

Figure 9. Results of test for the lookup operation

assigned around 1% of the network, but there is a lot of outliers that control a big portion of the network, some of them more than the 5%. This is more evident also in the 30 node case.

The second aspect we would like to study, now that we know that the space on the ring is equally distributed among all the nodes, is if the key distribution respect the same principle. We used **SHA1** also for the object encryption. So we want to know if on average inserting a crescent amount of key them are equally distributed among all the nodes in the network. For the experiment we used a network of 1000 nodes, we were not able to increase this number due to computational limits.

In Figure 8 are described the results of this tests. In Figure 8a is possible to analyze how the network react to an increasing quantity of keys. the average grows linearly. We can also take track of some outliers. In advance Figure 8b how the probability of having a large number of keys goes down very quickly.

Our last test concern the path to reach a key, thanks to the finger table mechanism this path should be shortest as possible. This is proven to be true thanks to the results described in Figure 9. On average the length of the path to reach a key grows lower than linearly. For this test we searched random keys (that we know are in present in the system) asking them to random nodes that are actually active in the ring. Thanks to Figure 9b we can see that the probability to have a path longer than 10 steps is really low in a big network of 1024 nodes.

Thanks to all this test we can conclude that our implementations is comparable to real systems that implement the protocol. The insertion of objects and the looking up operation are following what we expected from [1].

VI. Conclusions

This report start form the results achieved by [1] and pass through the implementation development with all the different choices that we made in order to simulate a distributed system using the Repast framework.

We start describing the structure of our implementation is section III and proceed to show a simple execution that uses the default parameters in section IV. To conclude testing the properties of Chord in section V. The peculiarity of Repast is that is possible to see if the execution is correct by only looking at the display evolving in time, indeed more than one bugs we found and correct were been discovered by looking at message exchange in the different displays. We demonstrate with our test that Chord is scalable and efficient, for sure has a wide range of application in distributed system, but could also being used in Actor system, for example in a scenario were more actors cooperate to complete a task and were every actor make available different resources, Chord could be used to keep tracking of which actor store a particular resource in a distributed way. Or maybe is applicable in a distributed self driving environment, to maintain information and the position of a self driven bus, arranging the stops in the ring, and passing the "bus" object from one stop to another one. Those are just few possible case that could be subject to future studies.

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: a scalable peer-to-peer lookup protocol for internet applications,” *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 1, pp. 17–32, 2003.

[2] D. Eastlake and P. Jones, “Us secure hash algorithm 1 (sha1),” 2001.