Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Daniele Giuliani
Student ID 203508
daniele.giuliani@studenti.unitn.it

***Abstract*—In today's modern world of IT, distributed systems have become ubiquitous, especially given the need to store and manage large amounts of data which cannot be handled by a single machine.**

One of the most common operation done in such systems is called *lookup*, which is the problem of finding the node in the system that is responsible for storing a specific data item. In this paper we analyze Chord [1], a distributed lookup protocol that addresses this problem.

We will present an in-depth analysis of the protocol evaluating its performance through the use of a simulator implemented using the Repast Simphony framework. We will start by describing in detail how the implementation was done and what are some of the difficulties encountered in the process, then we will present the simulator obtained and a brief tutorial on how to use it. Finally the result obtained from the analysis will be presented.

***Index Terms*—Chrod, Distributed Lookup, Distributed Systems, Repast, Java**

I. Introduction

In the last few years distributed systems have become almost ubiquitous. Distributed storage systems are a particular type of distributed systems used to store data, these have become increasingly popular especially due to the rapid growth of the rate at which data is being produced by all the digital devices connected to the internet. Such a large amount of data cannot be stored on a single machine, thus creating the need for such systems. One of the most common operation performed in such systems is a *lookup* which means locating a node in the network that stores a specific data item. The protocol presented by [1], is called Chord and is a distributed lookup protocol that tries to achieve this operation while also providing some guarantees in terms of scalability.

In order to evaluate the performance of such protocol, while also providing some insight on how it works, we decided to implement a simulator using the framework provided by the Actor-based modelling system Repast Simphony. This framework provides API for the Java programming language and it is especially useful because it allows to easily implement some visualization aspects without the need to use directly Java graphic libraries.

In the first section of the paper we will discuss the implementation of the protocol in Repast Simphony, describing in detail some of the most important architectural aspects. Then we will present the simulator, explaining how the visualization works, following with a section describing the parameters of the simulator and how they can be tuned in order to obtain specific behaviours. Finally, in the last section, we will present the results of the different analysis performed on the protocol.

All the code, for both simulation and analysis, is publicly available in the Github repository[1].

II. Implementation

To improve the readability and ease-of-comprehension of the code base, we decided to follow an implementation that resembles the one described by the authors of [1]. The main actor of the simulator is the class Node, which implements all the functionality of a node taking part in the Chord protocol.

During this section we will describe some of the main components of our implementation as well as the architectural choices that were taken during the development process.

A. Message-Passing Framework

In the paper, the authors describe the protocol by using pseudocode assuming that the interaction between nodes happens through the use of Remote Procedure Call (RPC). For example: supposing that a node *n* is executing the code for the `find_successor()` method, than when it is required to get the successor of *n'* it will simply call the remote procedure `n'.successor` and resume the execution from that point as soon as the RPC terminates. Unfortunately repast does not provide a way to implement RPC-like mechanisms, for this reason we opted to implement a message passing architecture which also better fits the modelling of distributed systems.

The message passing, shown in Figure 1, is implemented through the use of a buffer called `messageQueue` and the methods `send()` and `receive()` implemented inside the node. When a node calls the `send()` method it passes the message to be sent and the destination of the message, then the method automatically computes the tick in which the message needs to be processed by the receiver, therefore allowing to easily simulate network delays, finally the message is placed inside the queue of the receiving node. Each node when executing, will call the `receive()` that will check for the presence in the queue of messages that need to be processed. In case there are, the method will return them one at the time. It is important to note that each node interacts with the `messageQueue` only by using the primitives `send()` and `receive()`, this way we abstract completely

[1]https://github.com/daniele122008/chord

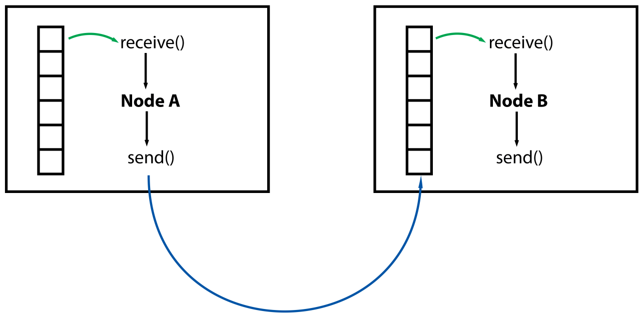

Fig. 1. Message passing architecture of the simulator

how the message passing is done, and can easily change the implementation in the future if required. As already said, this abstraction layer also takes care of simulating network delays, by storing inside of each message the tick in which they must be processed. Furthermore when the network delays are disabled each "non-local" message (meaning that the sender and the receiver are different nodes) sent at tick X will be processed at tick $X + 1$.

B. Execution suspension and Pseudo-Stack

Repast does not provide native support for a mechanism to send a message and suspend code execution until the reply arrives, for this reason we had to implement from scratch a custom structure called `suspendedRequests` that simulates the method call stack found in memory: when the execution of a method must be suspended to obtain some information (e.g. asking a node to provide its successor), an entry is added to this stack. When the reply for the query arrives, the stack is checked to see which method was suspended, and the execution is resumed. For this reason, each query is associated with a universally unique identifier (UUID) that is used as a key to identify the stack relative to a specific request, this way a node can still process other requests while it's waiting for the reply for a different query from another node. To further clarify this behaviour, we present a simplified example, while refereeing to the pseudocode in Figure 2.

In order to perform the lookup of an ID the node will call the method `find_successor()` which executes `find_predecessor()`, which in turn calls the method `closest_preceding_finger()`.

Since these method might not be local, to perform them we need to send a message to the target node, asking for their execution. Each time a call is not local these steps must be performed:

1) Suspend execution.
2) Add entry to the list of suspended requests.
3) Send "appropriate" (we will define later what this means) message to node that must execute code.
4) When reply arrives, get correct entry in list of suspended requests.
5) Resume execution.

```
// ask node $n$ to find $id$'s successor
$n$.find_successor($id$)
  $n' = find\_predecessor(id)$;
  return $n'.successor$;

// ask node $n$ to find $id$'s predecessor
$n$.find_predecessor($id$)
  $n' = n$;
  while ($id \notin (n', n'.successor]$)
    $n' = n'.closest\_preceding\_finger(id)$;
  return $n'$;

// return closest finger preceding $id$
$n$.closest_preceding_finger($id$)
  for $i = m$ downto $1$
    if ($finger[i].node \in (n, id)$)
      return $finger[i].node$;
  return $n$;
```

Fig. 2. Pseudocode of Chord as described in [1]

An "appropriate" message, means a message containing all the necessary information for the receiving node to correctly complete the request. In the implementation we have defined messages such as `FindPredecessorMessage` following a naming convention that resembles the action being executed.

In our implementation, the steps just described are also done when the method call is local. This way we don't have to distinguish between local and remote call, allowing us to use a single method instead of having to create a local and a remote handler. This provides more uniformity and a source code which is easier to comprehend. It must be noted that although we use the message passing scheme even for local calls, no network delay is applied to messages sent from a node to itself, therefore the node is able to process the message in the same simulation tick in which they were sent.

When a node initializes an operation (e.g. starts performing a lookup for some ID) a new UUID representing that specific query is generated. This UUID will be used as a key in order to identify the suspended requests relative to that specific operation. Furthermore, if the node needs information from other nodes in order to complete the operation, it will sent a request message containing the same query ID. This way it's extremely easy to track how the requests relative to a single operation move throughout the system, which is especially useful when performing the analysis.

The structure and the behaviour just described are represented graphically in Figure 3.

C. Successor List and Stabilize

For our simulator, we decided to implement the more advanced version of the protocol where each node stores a list of successors, instead of a single one. This allows to deal with failures, since when a node is no longer able to communicate with its successor, it can immediately switch to the next one by using such list.

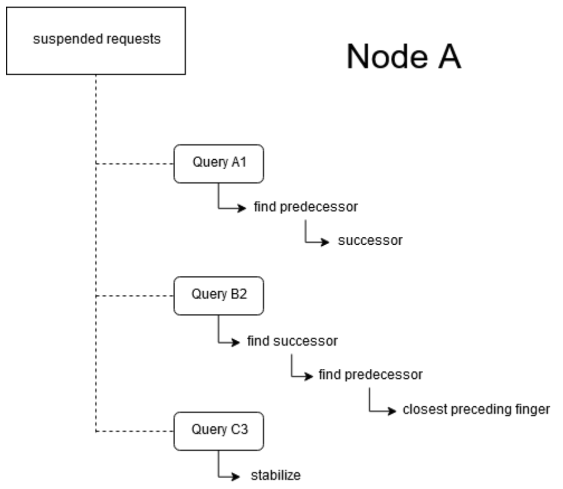

Fig. 3. Pseudo-stack of the implementation to store suspended and resume requests

```
n.stabilize(id)
  if (predecessor.isDead)
    predecessor = nil;
  if (stabilizeNotAlreadyInProgress)
    n' = successors[0].predecessor;
    if (successors[0].isDead)
      remove successors[0]
      shift successors array back
    if ($n' \in (n, successors[0])$)
      successors[0] = n';
    successors[1..end] = successors[0].successors[0..end-1];
    successor[0].notify( )
```

Fig. 4. Pseudocode of new stabilization procedure engineered

Although the use of a successor list is very beneficial, it requires the use of a different stabilization procedure, as the one described on the paper [1] is correct only in the case of a single successor. Unfortunately the authors only mention this modified version of the `stabilize()` method and do not describe such procedure, for this reason we had to engineer it ourselves. The pseudocode describing the new stabilization process in shown in Figure 4.

Conceptually it is composed of five operations:

1) Checking if the predecessor of the current node is still alive. If it does not respond, set the predecessor to *nil* (the stabilization of other nodes will eventually provide a new predecessor for the current node).
2) Ask the immediate successor (first entry in the successor list) to provide its predecessor (if the immediate successor is dead, switch to the next one in the list).
3) If a new node has joined, update the immediate successor and the successor list to reflect that.
4) Ask the immediate successor to provide its successors in order to update our successor list.
5) Notify the immediate successor about our existence (as in the normal stabilize procedure).

In order to increase the redundancy of the system, each nodes communicates to all of its successors the values stored. This way, if the node fails, the successors are used as replica in order to prevent the loss of information.

D. Code Repository and Packaging

In this subsection we describe how the code is organized inside the repository. The project is divided into five packages, we will briefly describe the content and the purpose of each one of them.

a) chord: This package contains the core component of the simulator. The `Node` class contains the logic of the protocol, while the `ChordBuilder` class is an implementation of a custom context builder used to initialize the simulation process in Repast Simphony. The `Configuration` and `Helper` classes contain respectively the code to load the simulation parameters and a set of utility function used all around the project (e.g. computing $log_2(x)$, obtaining the current tick of the simulation, etc.).

b) messages: This package contains all the different type of messages that can be sent around the system. Since each type of message might need to store different information, different classes are provided. Most of the messages are composed of two classes: a request and a reply counterpart, with the exception of the `NotifyMessage` which does not require a reply from the receiving end. All the message classes extend the abstract class `Message`.

c) requests: This package contains all the classes describing the different types of requests that will fill the `suspendedRequests` structure. Each different request type might need to store different information. All request classes extend the abstract class `Request`.

d) visualization: This package contains all the code regarding the visualization aspects of the simulator. We will explain it more in detail in Section III.

e) analysis: This package contains all the classes used to collect data for the analysis of the protocol. The `Collector` class contains custom data structures that are used to collect information, and acts as a meta-actor that queries the nodes during each tick of the simulation to collect appropriate data. The `Analysis` class, allows to enable and disable each individual analysis, by default the analysis are not performed in order to reduce the load on the simulator. All the other classes are implementation of custom data sources from Repast Simphony, which allow to create dataset in the CSV format based on the data collected by the `Collector`.

III. Real time visualization

In order to provide an easy comprehension at a glance on how the protocol works, we exploited the API provided by Repast Simphony to create a real-time visualization of how the events unfold. As we can see from Figure 5, the ID space is represented as a ring onto which are placed the different

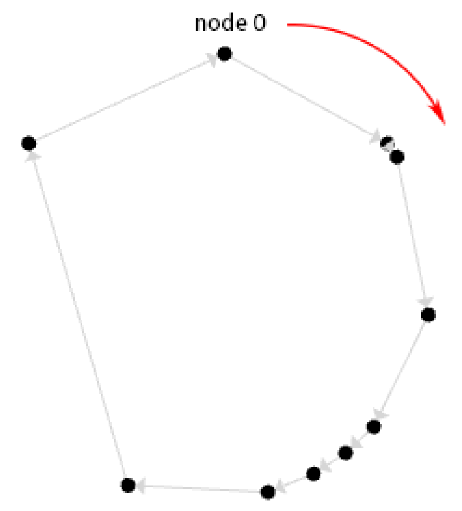

Fig. 5. Initial visualization of simulator, with placement of nodes and successors

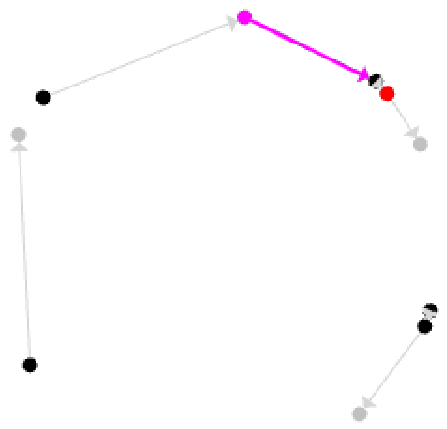

Fig. 6. Visualization of the different node colors reflecting their state

chord nodes. The IDs are placed in an increasing clockwise formation: the lowest ID is placed at the top (12 o'clock), by following the ring clockwise we arrive at the bottom (6 o'clock) which represents the center of the ID space.

A node is represented as a circle of a solid color which depends on its state:

- *Black Node*: the node is participating in the protocol.
- *Gray Node*: the node is not currently participating in the protocol, it might have crashed or it might be trying to join the system.
- *Colored Node*: the node is currently performing a lookup.
- *Red Node*: the node was performing a lookup that failed.

Figure 6 shows an example of these cases.

During the simulation we visualize some of the lookups performed by the nodes. To avoid cluttering the interface with unnecessary information, only a single lookup at the time is visualized even tough many might be executed simultaneously in the system. Lookup operations are represented by the color cyan, yellow and magenta. A switch between these colors indicates the end of a lookup and the beginning of a new one.

An edge represents a link between two nodes, the color of the edge describes the type of the link:

- Light-gray edges: represent the successor of a node. They allow to visualize the current state of the chord ring, the joining process performed by new nodes and can be useful to check whether nodes have become isolated.
- Colored edges: represent a communication, happening between two nodes, in order to perform a lookup operation. A bright color indicates a request message, while a darker shade of the same color represents the corresponding reply. This allows to intuitively visualize the chain of requests and replies, while also showing the traversal of the chord ring during the lookup process.

We can see an example of request message Figure 6.

It is important to note that in the first 100 ticks of simulation, no colored arcs will be visualized. In fact this interval of time is used, at the beginning of every run, in order to stabilize the network and allow the nodes to exchange information and fill the respective tables. During this period no lookups, crash and join requests are generated.

The simulator provides parameter in order to customize some aspect of the visualization, such as how long a crashed nodes remains inactive before being removed from the view. The use of these parameters is explained in the next section.

IV. Parameters of the Simulator

In the following section, we will explain the parameters that can be tuned in order to personalize the simulation. Given the modular structure of the software, many different analyses can be performed by tweaking these parameters appropriately. Since Repast Simphony automatically sorts them alphabetically by their name, we have decided to name all the parameters with a number and a letter at the beginning, thus giving us the ability to group them in a more appropriate manner. Together with the description of each parameter, we will also provide a default value, which we suggest using in order to model a generic execution of the protocol under normal conditions, which provides an intuitive explanation on how it works.

- *Max Number of Nodes [256]*: maximum number of nodes that can be present at any given time during the simulation. This also corresponds to the size of the ID space. A low number is suggested in order to improve the visualization, as the different nodes will appear further away from each other instead of being really close.
- *Min Number of Nodes [5]*: minimum number of active nodes that must be present at any given time during the simulation. Useful to keep the simulation going and avoid

a scenario where all nodes crash, therefore terminating the run. When the minimum number of nodes is reached, no more nodes will be crashed until new ones join the system.

- *Initial Number of Nodes [10]*: number of nodes present in the system at the beginning of the simulation. Again a low number is advised for better visualization.
- *Synchronous Mode [enabled]*: establishes if the model should run in Synchronous or Asynchronous mode. When the synchronous mode is enabled, each message set is received in the following tick, random transmission delay do not occur.
- *Max Transmission Delay [10]*: maximum transmission delay between the send of a message and the receive of the same by another process. This parameter influences the simulation only if the model is running in Asynchronous mode.
- *Successor List Size [5]*: size of the successor list of the nodes. The authors of [1] suggest using as size $log_2(N)$ where N is the number of nodes in the system. Higher values can be used to improve fault tolerance, lower values can be used to destabilize the system.
- *Average Stabilize Interval [30]*: average number of ticks between each stabilization. Since this is an average is possible that much less, or much more, ticks will elapse between the execution of two stabilization procedure.
- *Average FixFingers Interval [30]*: average number of ticks between each fix fingers execution. Since this is an average is possible that much less, or much more, ticks will elapse between the execution of two fix fingers procedure.
- *Request Timeout Threshold [50]*: number of ticks after which a request is considered in timeout and is removed. This value should be greater than $2 * log_2(N_{max})$, where N_{max} is the maximum number of nodes, in order avoid deleting a request which might be still ongoing.
- *Stabilization Timeout Threshold [3]*: number of ticks after which a stabilization request is considered in timeout and is removed. This value can be much smaller than the previous one since these requests involve only an immediate answer from a single node (instead of multiple nodes). Lower values increase the responsiveness of the system to failures. This value should be greater than $3 * MaxTransmissionDelay$ if the system is running in Asynchronous mode, in order to avoid considering failed a request that is only delayed.
- *Crash Probability [0.001]*: the probability (between 0 and 1) of a process to crash during every tick of the simulation.
- *Join Probability [0.001]* the probability (between 0 and 1) of a new process to join during every tick of the simulation.
- *Lookup Probability [0.2]*: the probability (between 0 and 1) of a process to perform a lookup operation during every tick of simulation.
- *Default Random Seed*: default parameter of Repast. It describes the random behaviours, by using the same seed we can reproduce a specific run of the model.
- *Vis 1 - Timeout Showtime [10]*: number of ticks for which a node is shown in red color after its lookup request has timed out.
- *Vis 2 - Crashed Nodes Showtime [50]*: number of ticks before an inactive node is removed from the view. This value should be greater than the Request Timeout Threshold, in order to avoid preemptively removing form the view, a node that is still performing the joining process.

During section V, each analysis that uses specific parameters will describe them and explain why this is the case. If the parameters are not shown, the default values can be assumed.

V. Analysis

In order to make our analysis more effective, we employ the ability provided by Repast Simphony to define custom data sources used to extract the data in a CSV format. As described in Section II-D, all of the implemented custom data sources can be found inside the `analysis` package of our application. To enable different analysis, the reader needs to modify the attribute `ACTIVE` inside the `Analysis` class and recompile the project, after this the custom data sources can be added as datasets inside the Repast simulator and the CSV file will be created at the end of the simulation run. Each custom data source extracts information form specific structure of the `Collector` agent, the CSV files are then processed in Python using the *Pandas* library. Plotting is done using the Python libraries *Seaborn* and *MatPlotLib*. All the code regarding CSV processing, data manipulation and graph plotting can be found under the *python-scripts* folder inside the repository, together with all the data collected for the analysis performed and a high definition version of the plots presented in this paper.

Each one of the analyses performed describes different aspects of the protocol. Some helped us detect bugs in our implementation, while others allowed us to discover possible ways of improving the protocol itself.

A. Lost Values and Probability of Data Loss

In order to test the redundancy of the system, we performed an analysis of the number of values lost when a percentage of the nodes in the system crashes simultaneously. It's important to remember that, as described in II-C, each node asks all of its successors to store a copy of the values, for this reason the loss of data occurs only when $Successor_{size}+1$ consecutive nodes crash simultaneously (we need a specific node, as well as all of its successors to crash). If the crash is not simultaneous, then the remaining nodes will notice the crash and start replicating again the data, thus preventing the loss.

Figure 7 shows the number of values lost per run, with a system containing 32 nodes, half of which crash simultaneously. This study was repeated for 100000 runs, while using a list of successor of size $5 = log_2(32)$. As we can see, there is a high variability in the number of values lost throughout the runs, this is due to the fact the the nodes are placed randomly

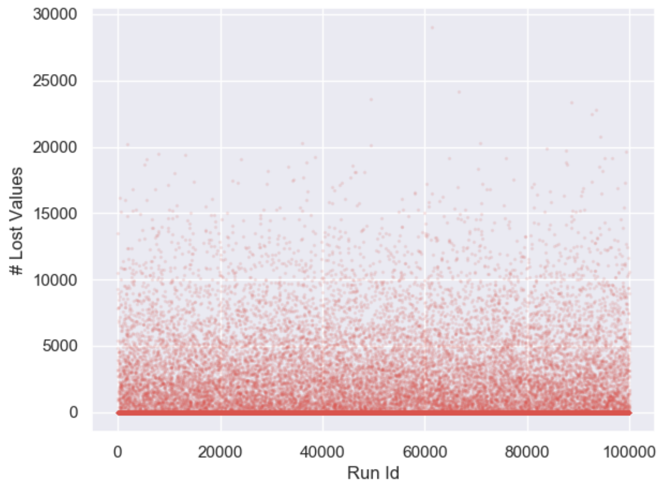

Fig. 7. Number of values lost in a system with 32 nodes with 5 successors each, during a simultaneous crash of 50% of the system

around the chord ring at the beginning of the simulation: the way the nodes are placed highly affects the number of values lost. This can be easily seen by looking at Figure 8: in both these cases 6 nodes crash, but in the first case the node that loses data is responsible for a small portion of the ID space, while in the second, the node is responsible for a huge chunk of the ID space.

Nevertheless Figure 7 shows that in the great majority of runs no data was lost, this is represented by the extreme density of points creating a solid line on $y = 0$. This can be easily understood by looking at Figure 9 that shows the probability of loosing data as we increase the percentage of nodes crashed simultaneously. These results were obtained by computing the fraction of runs that lost data over the total number of runs, each probability was computed over the course of 100000 simulations.

Computing this probability theoretically is complex: in a system with 32 nodes where 50% of them crash (meaning that 16 nodes crash) the possible number of configuration that we obtain by picking different nodes are:

$$\binom{32}{16} = \frac{32!}{16!(32-16)!} \approx 6.2 * 10^{21}$$

But the number of configuration that actually lead to data loss (which are the one where $Successor_{size} + 1$ successive nodes crash) are much smaller. Although this number is difficult to compute, by using such a high amount of runs we are confident to have obtained a good approximation of the probability of data loss.

In the end this analysis shows how robust the protocol really is, even with a 30% crash of the system we have no data loss. Furthermore a scenario where a large amount of nodes crash at the same time is really unlikely to happen, provided that the underlying network connecting the nodes is reliable enough.

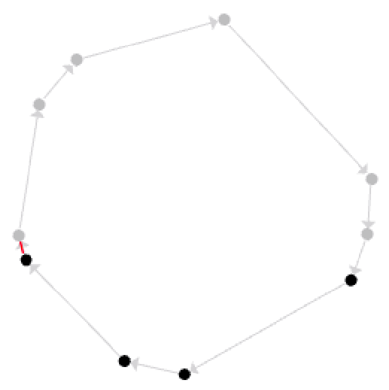
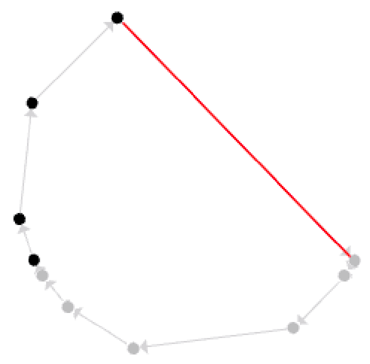

Fig. 8. Chord ring with 6 nodes crashed, the part highlighted in red represents the part of the ID space for which values were lost

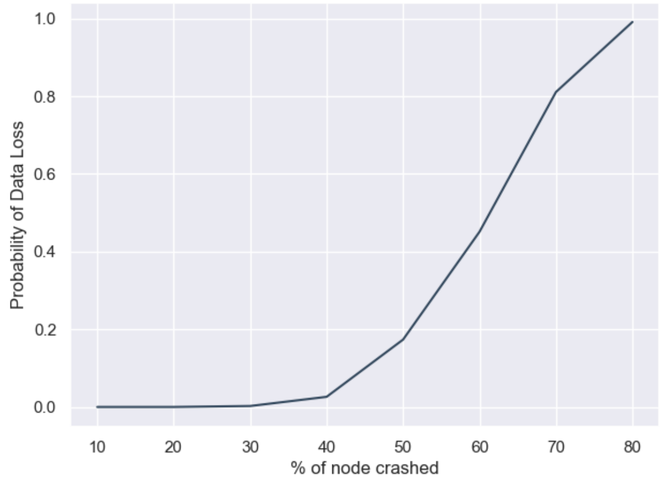

Fig. 9. Probability of losing data in a system with 32 nodes with 5 successors each, as the percentage of failed node increases

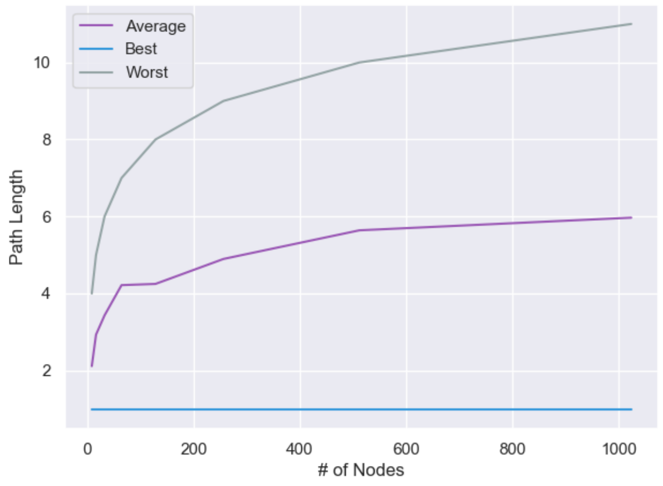

Fig. 10. Number of nodes contacted during lookup operations, in a system with $2^{14} = 16384$ possible IDs

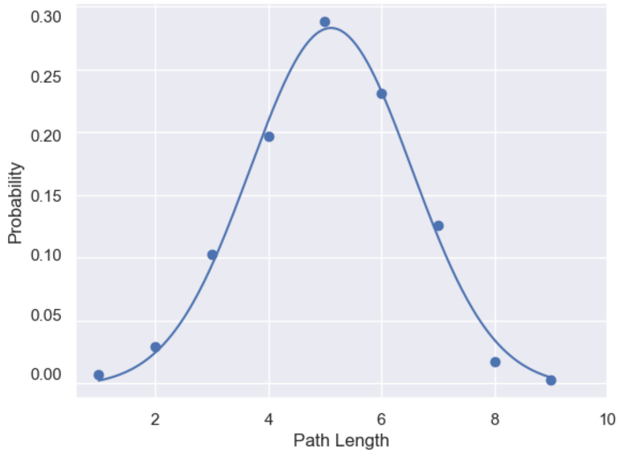

Fig. 11. Probability density function of the path length for lookup operation in a system with 256 nodes

B. Lookup Path Length

To evaluate the efficiency of the protocol, we decided to repeat the analysis performed by the authors of [1] and compute the number of nodes that are contacted during a single lookup operation. This analysis also allowed us to asses whether our protocol was working correctly or not. To perform it, we created a simulation scenario where the ID space can contain at most $2^{14} = 16384$ different IDs, and performed many lookup operations as we increased the number of nodes. Figure 10 shows the number of nodes contacted per lookup on average, as well as worst case and best case scenarios.

In the best case the number of nodes contacted to perform the lookup is only one. This happens when the node that needs to perform the lookup of a key is also responsible for that key. As we can see, in the worst case the number of nodes contacted follows an $O(log_2(N))$ growth, where N is the number of nodes in the system, on average the number of nodes needed are even less.

This confirms that our implementation of the protocol is working as intended, while also showing the scaling potential that Chord has.

It it important to note that for this analysis the timeout before considering a lookup failed was set to a high value (5000), this was done in order to make sure that if the protocol was working incorrectly, and many nodes were contacted, this behaviour would be registered in the data collected. Otherwise if the lookup failed because of the timeout we would have ended up recording only a fraction of the number of nodes needed for that lookup, therefore collecting biased data and invalidating our analysis.

To better evaluate the efficiency of the protocol under normal condition of operation we decided to perform a more in-depth analysis by collecting data about a high number of lookup operations and computing the probability density function for the path length when the system contains a fixed number of nodes (256). The result are shown in Figure 11.

As we can see the worst cases where $log_2(256) = 8$ nodes must be contacted are very few, with probability around 2%. Instead the vast majority of the lookups are solved by contacting five nodes. This result aligns with the one presented on the paper [1] and confirms the great scalability properties of Chord.

C. Churn Rate and Lookup Fail

In order to test the robustness of Chord and evaluate its ability to perform queries as the network is constantly changing, we decided to analyze the number of lookups carried out correctly while a predetermined amount of nodes crashes and enters each round. During this analysis we consider a lookup successfully completed if we are able to contact the successor of the desired identifier. Furthermore, in case a node does not respond, we do not attempt to perform the query by following another path, but simply consider the lookup as failed, this is done in order to estimate a worst-case baseline for the performance.

Figure 12 shows the probability of a lookup to fail as the churn rate (probability of a node to enter/exit the system each round) increases. As we can see with a churn rate of 0.01 (meaning that on average a node enters/exits the system every 100 ticks) the number of failed lookups is almost zero, because the network has enough time to fix the finger tables and successor entries. Unfortunately by increasing the churn rate we obtain a sudden spike in the number of lookups failed, which increases to about 11%. This is not in line with the results obtained by the authors of [1], which show a much lower percentage of failed lookups. Furthermore by increasing the churn rate past 0.05, after a number of simulation rounds, the system becomes so unstable that the nodes become all isolated and the simulation ends.

This discrepancy in the results is due to multiple factors. First of all, our random generation of lookups, joins and

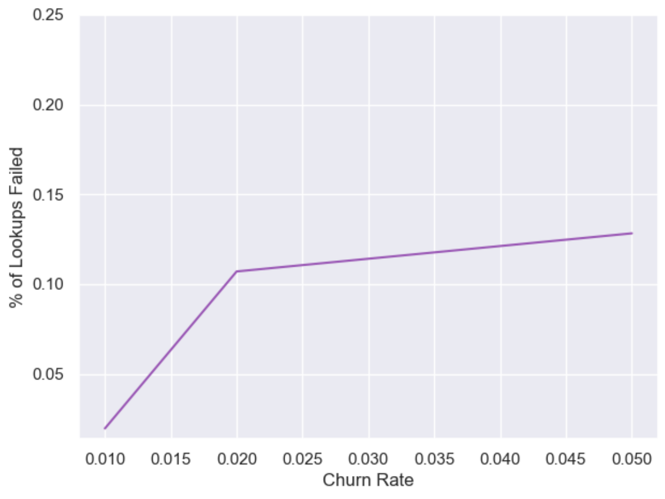

Fig. 12. Probability of lookup fail in a system with given churn rate

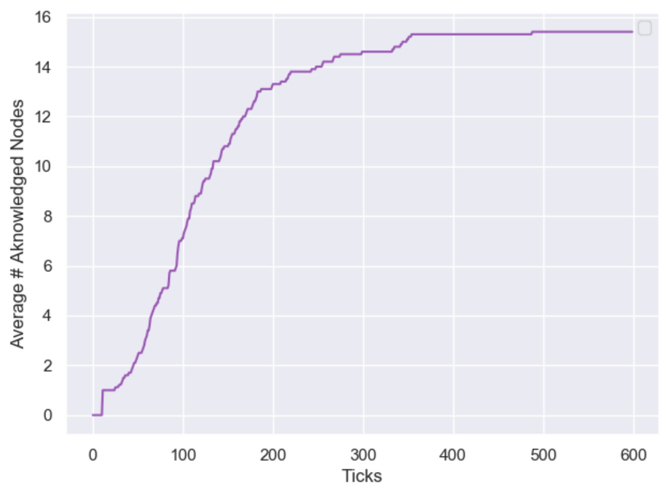

Fig. 13. Number of nodes which become aware of a newly joined node as time elapses

crashes is slightly different then the one done in the paper [1] which follows a Poisson process, instead we use a random number generator to perform one of these operations with a specific probability each round. Furthermore our implementation gives up on some reliability aspects in order to obtain a more flexible and easy to comprehend code. In fact, the paper does not explain how the implementation of the *stabilize* method works, when using a list of multiple successors instead of a single one. The correctness of this method is fundamental for achieving a working implementation, for this reason we decided to implement a simplistic version which is easy to understand but might not be the most efficient. As already mentioned, the pseudocode for the stabilize method is shown in Figure 4.

In our implementation, the execution of the stabilization procedure is done on average every 30 rounds. It is important to understand that the stabilization procedure is not instantaneous, in fact in a best-case scenario, when there are not network delays, it takes 6 round to complete. This is due to the fact that three request/reply interaction must be done, one to check the liveness of the predecessor, one to get the predecessor of the current successor and the last one to get the successors list. Furthermore, in a single node, only one stabilize procedure at the time can be active. This is done in order to avoid the possibility of updating incorrectly the successor list and ending up in an inconsistent state, due to random network delay which could create a situations where responses to an "older" execution of the stabilization procedure, are processed after a "newer" stabilization procedure has already been done. Although this could easily be avoided by time stamping locally the different stabilization requests, we opted for a conceptually simpler version, given the fact that one of the goal of this study is precisely to obtain an implementation of Chord which is easy to understand.

D. Node Join and Knowledge Spread

To better evaluate how responsive the system is, we decided to analyze how quickly the knowledge about new nodes is spread around the network. To this end, we simulated a chord network for a while allowing it to stabilize, then we initialized a join procedure for a new node and collected information describing in which tick nodes become aware of the new participant. This procedure was repeated several times and the result were averaged to get a more robust measurement. To avoid disrupting the results, the random generation of crash and fail was disabled, and both the stabilization and fix finger procedure were executed every 30 ticks on average. The size of the ID space chosen for this analysis was 2^{14}, while the initial number of nodes was 2^8 with a list of successors of size 8.

The results are shown in Figure 13. The sudden spike at the beginning, represents the fact that the joining process was completed and the newly joined node has notified its successor. It is interesting to notice that even though the stabilization and fix fingers procedure are executed on average each 30 ticks, it can take much longer for some nodes to become aware about the new node (insert the new node in their finger table or successors list). This is due to the fact that a single stabilization procedure might not be sufficient to spread the knowledge around, as mentioned briefly by the authors of [1]. Furthermore, the interval of execution of the stabilization and fix fingers procedures is probabilistic, this means that a few nodes might take much shorter or much longer that this, which explains why the number of acknowledged nodes slightly rises even after more than 300 ticks have passed. Regardless, most of the nodes (approximately 87%) is aware of the newly joined node after 180 ticks, which is consisted with how the protocol should behave. After 500 ticks the knowledge has reached all the network, meaning that no more changes occur inside the fingers table or successors lists of the nodes.

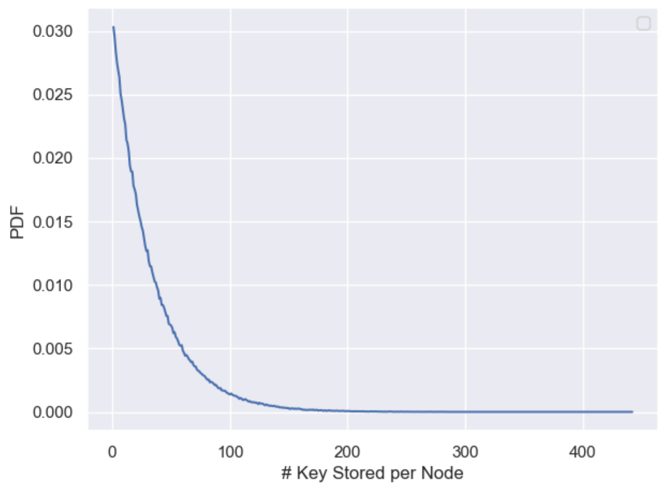

Fig. 14. PDF of the number of keys stored by each node of a network with 2^9 nodes and 2^{14} keys

E. Load Balancing

To test the ability of Chord to balance the load evenly among the nodes we performed an analysis of the keys distribution, by generating IDs randomly and assigning them to the nodes. The experiment were performed on a network with 2^9 nodes and 2^{14} possible keys.

Figure 14 shows the result obtained. As we can see they are very similar to the ones of the original paper [1], with the only difference being that our curves results much smoother because we computed an average over a thousands runs. As we can see there is a big variations in the number of keys that are stored in each node: In all of our runs, we have at least a node that does not store any key, while in some of the runs few nodes end up storing 90% of all the keys in the network, which is reflected by the PDF.

This high unbalance in the load is created by the position of the nodes inside the ID space, as we already explained in Section V-A. As the authors of the paper [1] explained, this problem can be solved through the use of consistent hashing and virtual nodes.

VI. Conclusions

As analyzed in this paper, the Chord protocol results being very robust: data loss is a rare occurrence even in the face of a simultaneous crash of a large chunk of the network. Furthermore Chord results salable and ideal for system with a high number of nodes, although we could only test this up to a limited number of nodes (around 10000) due to the fact that Repast Simphony does not provide multi thread support and the simulation slows down when a large number of nodes is selected.

Given that Chord is a distributed lookup protocol, in a real-world scenario specific data items are associated with keys. The results we obtained are based on the simulator created, where the keys represent the identifier as well as the data item. Although this was sufficient for our evaluation purposes, in a practical application the data items might be significant in size therefore requiring large amount of data to be replicated around. If the churn rate is too large, the replication process might not be fast enough to keep up with the loss of nodes. In such a scenario, it could be useful to assign different roles to nodes with specific characteristics, for example, nodes that are more powerful and stable (participate in the protocol for longer intervals of time) might be elected as "super peers" with the ability to store greater amount of data, therefore diminishing the load on less powerful nodes and preventing useless replication steps.

This hierarchical approach could also be used to facilitate the joining process of new nodes. In our simulation, when a new node wanted to join the system, an existing node was randomly chosen as an entry point for the new node. On the other hand, in a real-world scenario, a peer-sampling service is needed. It is important that this service runs on nodes which are relatively stable through time, in order to allow new nodes (with no prior knowledge to the network) to acquire an entry point. For this reason "super peers" seems to be the most logical choice for the placement of such service.

Additional optimization to the stabilization procedure can also help rendering this protocol even more robust, and given the modular architecture of the simulator, it is fairly easy to implement new procedures or swap out some components in order to test the protocol in different environments.

VII. How to Install

In order to run the simulator:

- Execute the installer `chord.jar`, this can be done via the terminal by typing *java -jar chord.jar*, or by double clicking on the *.jar* file if a default association with Java already exists.
- Follow the installation wizard provided by the Repast Simphony Model Installer.
- Execute the the file `start_model.bat` or `start_model.command`, respectively for Windows-based operative systems and Linux-based ones.
- Customize the parameters of the model in the parameters panel if necessary, then initialize and run the simulation.

Java 11 or newer is required to correctly execute the simulator. If the simulation does not start, consider updating your Java Runtime Environment installation.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan (2003). Chord: A scalable peer-to-peer lookup protocol for Internet applications. IEEE Transactions on Networking, Volume 11, 10.1109/T-NET.2002.808407