Implementation of Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications

Giacomo Callegari
Master in Computer Science
University of Trento
M.207468
giacomo.callegari@studenti.unitn.it

Fabio Molignoni
Master in Computer Science
University of Trento
M.203201
fabio.molignoni@studenti.unitn.it

***Abstract*—The report contains a description of the development and the results obtained in the implementation of Chord [2]. The algorithm was implemented using "Repast Simphony" [1], an open-source agent-oriented modeling framework. The simulator generated using Repast was then used as the basis for the analysis of the algorithm. In this project, we focus mainly on exploring how well the algorithm scales. In the final part, a comparison between the results obtained in our implementation and the one described in the paper is provided.**

1. Introduction

Chord is a protocol that implements distributed lookup. It addresses the problem of finding the node that stores a desired data item efficiently. The protocol provides just one operation: given a key, it maps the key onto a node. The algorithm leverages the fact that both nodes and items have an associated hash value. A node in Chord requires information about $O(\log N)$ other nodes in the system. The lookup of a key requires $O(\log N)$ messages to be executed. The protocol was developed assuming that nodes can join and leave arbitrarily; for this reason, a section of the algorithm deals with maintaining information of each node consistent even in a dynamic system. The authors of the paper state that Chord provides a solution to five different problems: load balance, decentralization, scalability, availability and flexible naming.

Chord works by assigning to each item and each node of the system a hash using a pre-defined hash function. In the original paper, consistent hashing is provided as a $m-$bit SHA-1 identifier. For a node, the IP address is used as input of the hash function. Using the identifiers associated with each node we can construct an *identifier circle*, where each node has a connection to its successor, i.e. the node which has the nearest bigger value. The assignment rule of Chord states that key x is assigned to the first node whose identifier is equal to or follows x in the identifier space. Chord provides scalability by imposing that each node only knows a small subset of the entire network. In particular, each node only knows its *fingers*, i.e. nodes that handle keys bigger than $x + 2^k \mod m$ for $k = 1, 2, ...$ The power of Chord is that the size of the finger table is log-linear and, therefore, it can scale well as the number of nodes in the system increases. The finger table also allows the lookup to be log-linear: at each step of the search we remove half of the possible nodes that may be responsible for the ID since the node sends the lookup to the closer node in its finger table.

In the *Architecture* section a detailed description of the implementation is presented. In particular, it is described how the Chord algorithm was shaped in according to the Repast framework requirements. This section also contains a brief description of the simulator and a guide on how to install it on the machine. The *Analysis* section contains the analysis done over the implemented algorithm using the simulator provided by Repast. Since one of the main features of Chord is scalability, we focus our attention on how the protocol behaves as various parameters scale up. In the final *Conclusion* section a final evaluation on the implementation is provided.

2. Architecture

In figure 1 is presented the architecture of the implementation.

`Node` is the main class of the project. This class implements the behavior of one node of the system. Each node runs the Chord algorithm and is identified by a unique ID. In the paper, the ID of the node is computed as the SHA-1 hash of the IP of the machine on which the algorithm runs. Since we thought that this approach was unnecessary in a controlled simulated environment like ours, the IDs of the nodes are randomly picked from a uniform probability distribution in the range $[0, 2^m)$. Since the goal of the hash function is to evenly distribute nodes across the domain, our implementation does not impact in any way the execution of the algorithm. Using this approach, we also enforced nodes to take different IDs. In this way, we removed the possibility of collapse, i.e. two nodes having the same ID. The `NodeContext` class inherits from Repast's `Context` and was created to group all nodes in a single collection

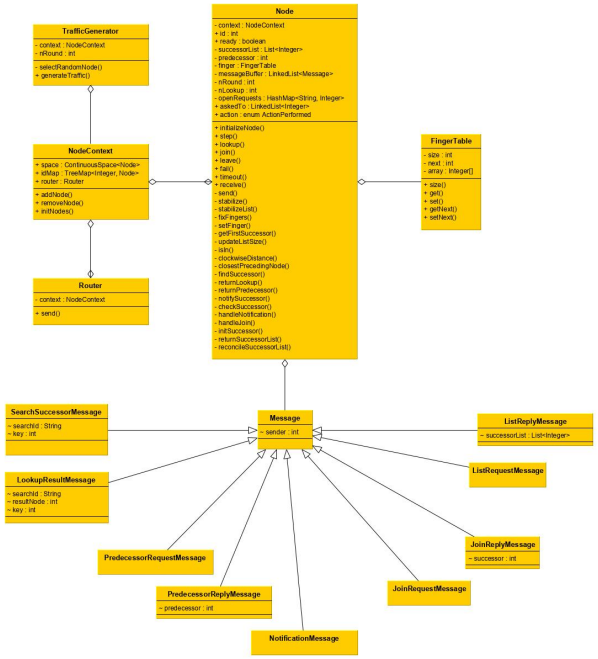

Figure 1. Architecture of the Chord implementation

of agents. Other agents (i.e. `TrafficGenerator`) are located in the main context of the simulation.

A node can communicate with another node only using messages. Since Repast does not provide a built-in mechanism to manage message communication, we implemented custom classes. `Router` is the middleware that makes communication happen. A node that wants to send a message invokes method `send(message, nodeId)`. The router adds the message to the `messageBuffer` of the receiver. Using this implementation, we allow the node to be agnostic of the implementation of other nodes since they communicate only through the router. At each step of the execution, each node reads all the messages present inside the `messageBuffer`, a buffer that contains received messages. Once a message has been read, it is removed from the buffer.

The lookup of the Chord protocol works recursively: if a node is not responsible for a particular key it will ask the node in its finger table that is closer to that value. This process is repeated until the node responsible for the key is found. At that point, the response is sent back to the nodes. In the original paper, the lookup is handled synchronously, since the method has a `return successor.find_successor(id)`. This is not directly implementable in our architecture, since nodes can only communicate through messages and messages are read once per round. This means that a single lookup spans over several rounds. In our implementation, each lookup has a unique ID, which is used to perform several lookups at the same time. A node that receives the request for a lookup and is not responsible for that key sends a `SearchSuccessorMessage` to its successor. This message is used to ask another node to perform the lookup. A node that receives a `SearchSuccessorMessage` first adds an entry in the `openRequests`. This hash-map keeps track of all lookup requests that the node received and has yet to solve. It maps each lookup ID to the ID of the node that performed the request. The node that is responsible for the key sends a `LookupResultMessage` to the requester. A node that receives this type of message has to send the result to its requester. To do so, the node extracts the ID of the requester from the `openRequests`. A node that has no requester in the hash-map is the initiator of the lookup. Using this approach we can perform the lookup asynchronously, since the synchronicity issues are handled through `openRequests`.

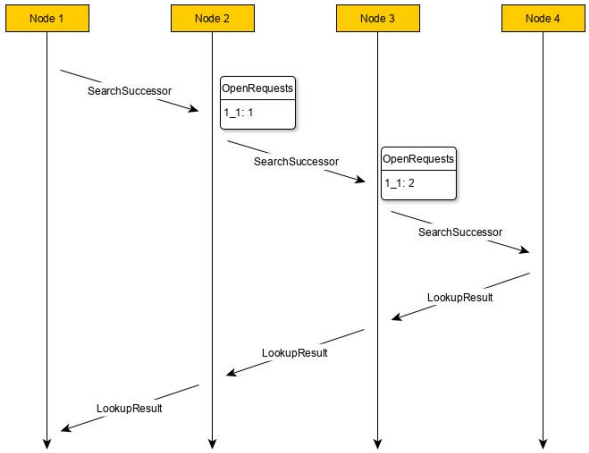

Figure 2. An example of lookup execution

In figure 2 is presented an example of execution of lookup. Node 1 wants to perform the lookup. Through the `search_successor` algorithm it contacts node 2 sending a `SearchSuccessor` message. Node 2, which doesn't know who is responsible for the key, adds the request to the open requests and forwards the lookup to node 3. Node 3 does not know the responsible for the key, so it adds the request to the open requests and forwards the lookup to node 4. The successor of Node 4 is the handler of the key, so it sends the response to node 3 through a `LookupResult` . Node 3 receives the message, checks openRequests and send the response to node 2. Node 2 checks the openRequests and sees that the request comes from node 1, so it sends the result to node 1. Node 1 was the initiator of the lookup, it has no entries in openRequests, so it returns the result. Note that a possible optimization of this approach would be, for the node who knows the handler of the key, to return the result directly to the original requester. This could be implemented just by adding an extra parameter in the message that tells who is the original requester. We decided not to implement it to be consistent with the algorithm defined in the paper: in the paper, the algorithm performs a call `v.find_successor(id)`. This is a recursive request, so the response to this invocation would be recursive as

well. This means that the response is passed through each intermediate node before arriving to the initiator. For this reason, in our implementation, the response is also passed through all the intermediate nodes.

A class called `TrafficGenerator` was created to generate traffic. The goal of the class is to periodically ask to perform a lookup. Each time the class asks for a lookup it selects a random node and selects a random key: this allows us to have stochasticity in the lookups that are done during the simulation. `TrafficGenerator` can also simulate nodes joining, leaving and failing: each of these events can happen with a probability that is specified as a simulation parameter. The traffic generator is also an agent of the simulation, since it uses a scheduled method provided by Repast.

A large part of the code is responsible for the dynamic composition of the system: as nodes join or leave, the pointers have to be updated (i.e. stabilized) with a certain frequency in order to guarantee lookups that are correct and efficient. The most basic stabilization is the one concerning the immediate successor of each node, but the finger table must be refreshed as well. Node failures are handled through the use of a successor list, which provides additional robustness. The traffic generator simulates the aforementioned events with a certain probability, selecting a random node of the system each time.

A node n starts the joining procedure when the method `join()` is called by the traffic generator: the node sends a `JoinRequestMessage` to another node n', which is already part of the system and which was randomly selected by the traffic generator. Upon receiving the message n' calls the method `handleJoin()`, which starts a lookup of the ID of n: a custom request ID is used. When n' obtains a response, it sends it to n through a `JoinReplyMessage`. Finally, n calls `initSuccessor()` and sets that lookup result as its successor.

Existing nodes must stabilize periodically to be aware of new arrivals. By calling `stabilize()`, a node n sends a `PredecessorRequestMessage` to its current successor s, which in turn responds with a `PredecessorReplyMessage` that contains s's predecessor p. Node n handles such message through `checkSuccessor()` and decides whether p should be its new successor; then, it sends a `NotificationMessage` to its successor. When receiving a notification, a node may decide to update its predecessor to n if it is closer than its current predecessor. The method `fixFingers()` refreshes one finger at a time: the k-th entry is updated by performing a lookup of the ID $n + 2^k$.

Node failures are handled through a successor list: if the immediate successor has failed, the list is traversed until an active node is found. Node n stabilizes the list by sending a `ListRequestMessage` to its current successor s: this node responds through a `ListReplyMessage` that contains its own successor list. Then, in the method `reconcileSuccessorList`, n maintains s as immediate successor and copies the received list. Normally the last element is discarded, but it may be kept to increase the size of n's list as more nodes join the system: in fact, the list has a logarithmic size. Now nodes call `getFirstSuccessor()` to obtain the immediate successor: this method also removes the failed entries from the list.

2.1. The simulator

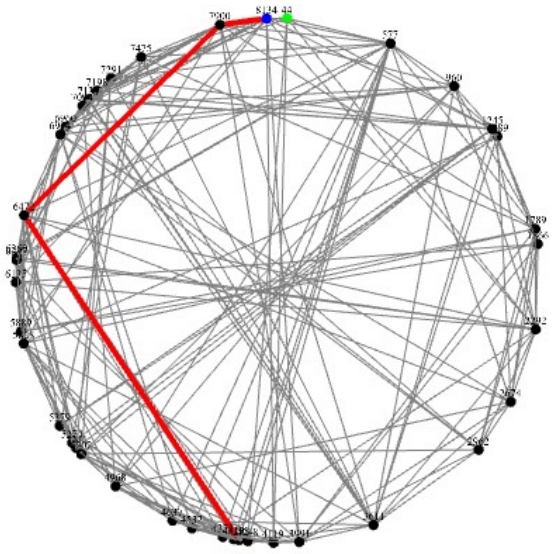

Figure 3. Visualization in the simulator

Figure 3 shows the visualization of the protocol in the simulator of Repast. Each node is represented as a dot, the number is its ID. The grey links represent the finger connections that each node has with other nodes. Red links compose the path that the lookup took to be solved. Nodes can take four different colors at each round:

- Black: the node was idle in that round.
- Red: the node asked another node to perform the lookup.
- Green: the node received a response to the lookup request it previously performed.
- Blue: the node's successor is responsible for the key.

Several parameters of the simulator can be tweaked to analyze different configurations of the system:

- **ID space**: the ID space allows you to define the range of the IDs. The maximum ID value is 2^m, where m is the input of this field. For example, if $m = 10$ then the ID space will have a range of $[0, 1024)$.
- **Number of nodes**: number of nodes initially in the system. Note that the system expects $\#nodes < 2^m$. This restriction was inserted to avoid ID collapse. If this constraint is not respected, an error will appear in the console and the execution will not begin.
- **Wait rounds**: the system performs a lookup request every x rounds. The node and the requested key are

chosen randomly. You can change the frequency of lookup requests using this parameter. If set to 1, the system will perform a new request at each round; a value of 0 deactivates external lookup requests.

- **Stabilization rounds**: The number of rounds between two stabilizations of a node's immediate successor. There is also a separate parameter for the stabilization of the successor list and the finger table. If the parameter is 0, the stabilization is disabled.
- **Probability of joining/leaving/failing**: Each of these events has its own probability, which can be tuned to simulate the dynamic composition of the system.

2.2. Simulator installation

We used Repast Simphony to create a self-contained JAR installer of the simulator, which can be found at the following link https://drive.google.com/open?id=1FQUvV0HONWukeIfaEcJV1CDF1wJFjE8f. The installation is very simple: after opening the installer, a step-by-step guide will be shown to the user. Then, the simulator can be started by running the batch file in the installation folder, which also contains the Java source code.

3. Analysis

In the analysis we used the simulator created through Repast to evaluate our implementation. Since the goal of the paper was to create a scalable peer-to-peer lookup protocol, we decided to focus mainly on evaluating whether this statement is true.

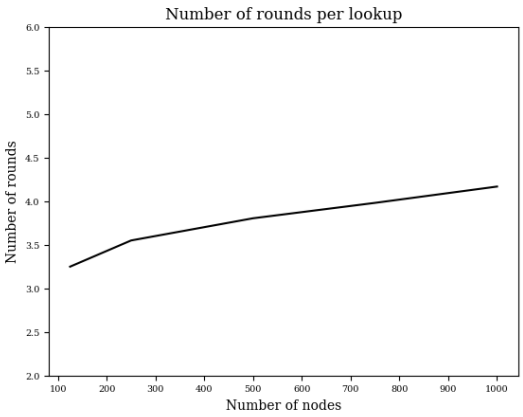

Figure 4. Average number of rounds required to perform a lookup

The first parameter we decided to analyze is the number of rounds required to perform a lookup as the number of nodes increases. Since the protocol is scalable, we would assume that adding new nodes would not increase that much the computation time of a single lookup. To perform this evaluation, we set a fixed $m = 20$ and we took an average time measurement with different amounts of nodes. The results are presented in figure 4. With a network of $n = 125$ nodes we get an average round time of 3.25 rounds per lookup, while with a network of $n = 1000$ nodes we get an average round time of 4.17 rounds per lookup. As the network size increases so does the required round time. The behavior of this graph is logarithmic, which means that the system can scale well, and therefore adding new nodes to the system does not significantly impact the performance of the lookup. This result was expected: since the lookup execution is $O(\log n)$, then the round time should be logarithmic as well.

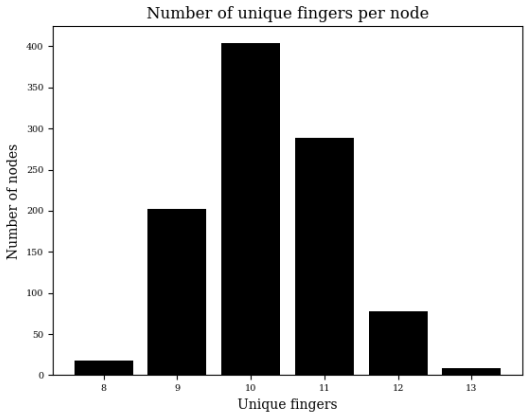

Figure 5. Number of unique fingers stored by each node

Another information that we wanted to evaluate was the number of unique "fingers" that each node has. The paper states that each node stores only $O(\log n)$ different nodes in its finger table. To check that, we created a network with $n = 1000$ nodes and $m = 20$ and we counted the number of distinct IDs stored inside the finger table of each node. The results are presented in figure 5. As expected, the number of unique fingers is log-linear w.r.t. the number of nodes. From our evaluation, 404 nodes have 10 different IDs in the finger table, while only 9 nodes have 13 different IDs. This property implies that nodes know only know a small section of the network and can still perform the lookup properly.

A parameter related to the previous evaluation is how many incoming connections each node has. There may be in the system nodes that are more important than others: for example, a node that is present in several finger tables may be contacted more often. Moreover, a node with several incoming links is in a section of the ID space where few nodes are present and, therefore, they all have to rely on that single node. To evaluate this parameter we computed the minimum, maximum and average number of incoming links for systems with different numbers of nodes. The results are presented in figure 6. There is a big difference between the maximum and the average values. For example, for a network of $n = 1000$ nodes, we have a maximum of 58, while the average number of incoming links is 9. This means that, as said, there are few really important nodes, which are used as fingers by several nodes but, on

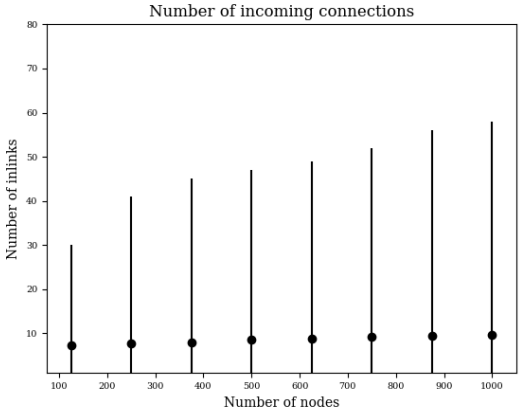

Figure 6. Number of incoming links connection of each node

average, a node can be contacted only by a small subset of nodes. As the number of nodes in the system progresses so does the average number of incoming links: the increase is logarithmic, since for $n = 125$ the average is 7.264, while for $n = 1000$ the average is 9.615.

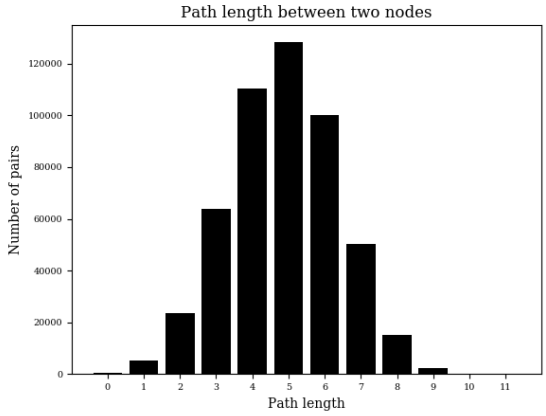

Figure 7. Path distance between nodes in the system

One crucial metric in the evaluation of the protocol is the path length. The path length tells the distance between any two nodes in the system. The smaller the path length, the fastest the protocol is, since the result can be found with fewer exchanged messages. To evaluate this parameter we defined a system with $n = 1000$ nodes and we measured the path distance between all nodes. The results are shown in figure 7. In the histogram, the pairs with path length 0 identify the distance between the nodes and themselves. The majority of the pairs have to perform 5 hops to communicate. This is something that we should expect from the correct behavior of the algorithm: at each hop the protocol removes half of the nodes, so in $O(\log n)$ it can connect the two nodes. Due to bad positioning, some pairs require a much longer time to communicate: 14 pairs require 13 hops to reach one another. This is anyway a really small number of combinations, therefore they do not affect the general performance of the algorithm.

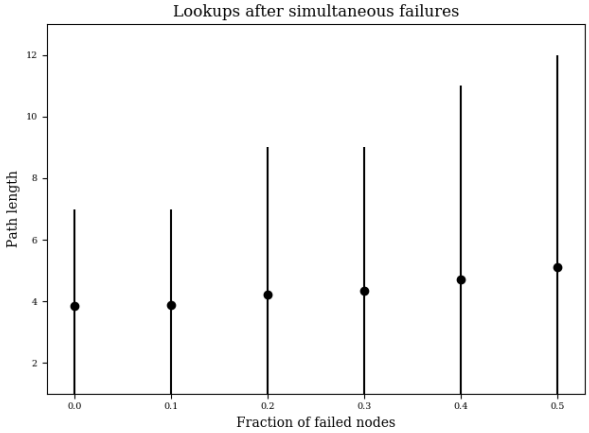

Figure 8. Path distance after simultaneous failures

We have also evaluated how the system performs in case of simultaneous failures. We considered a system that was initially stable, with $n = 1000$ nodes and a successor list size of $r = 20$: then, we simulated the failure of a fraction of nodes p at the same time and, without stabilizing, we performed 10000 lookups. In particular, we measured the path lengths of lookups with respect to p: the result is presented in figure 8. The average path length is very similar to the results of the paper and shows that, despite the massive departure of nodes and no stabilization, the protocol can still perform efficiently. Thanks to the successor list, all the lookups returned the correct node responsible for a key.

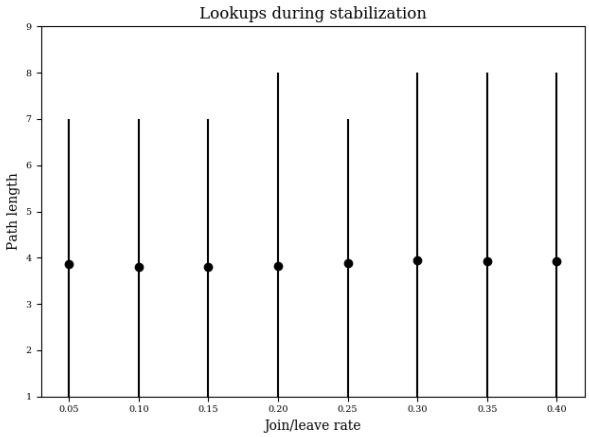

Figure 9. Path distance during stabilization

Finally, we have seen how Chord behaves when nodes continuously join and leave at a certain rate: the periodic stabilization of successors and fingers is fundamental to maintain the pointers sufficiently updated and allow efficient (and correct) lookups. Again, the system was initially

composed by $n = 1000$ nodes that had successor lists of size $r = 20$; we have considered 10000 lookups (one per round) and the stabilization was performed every 30 rounds. We experimented with different join/leave rates that represent how frequently nodes arrive or depart voluntarily: in this sense, we have applied some optimizations that let nodes update their successor and predecessor after such events. The results of figure 9, obtained after 10000 lookups, show that the average path length is very stable even in a more dynamic environment and therefore the performance of Chord is not affected. We have observed that, with higher join/leave rates, there might be some failed lookups: they are however very few with respect to the total number. In these rare cases, the lookup can be reattempted after a short time, waiting for the system to stabilize.

4. Conclusion

Chord is a simple, but very effective protocol for the lookup of resources in a peer-to-peer system. Its main strengths are scalability, load balance and robustness: we have confirmed these results in our simulation. In particular, we have seen that each lookup requires a logarithmic number of messages and each node needs to maintain only few pointers; the protocol can also withstand massive failures, still providing the correct results with a high probability.

Since Chord can be applied very easily on top of existing applications, it should be considered to provide lookup and replication functionalities in an efficient and flexible way.

References

[1] Aragonne National Laboratory. The repast suite. https://repast.github.io/.

[2] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.