Chord

Luca Zanella
Student ID 207520
luca.zanella-3@studenti.unitn.it

Giuliano Turri
Student ID 207553
giuliano.turri@studenti.unitn.it

***Abstract*—Today's peer-to-peer applications are widely used to store data in a distributed fashion. A common issue of these sorts of systems is how a particular data item is localized inside the entire network. Chord is a distributed Hash Table protocol designed to efficiently address this problem mapping a set of keys to each node. This paper will analyze our implementation of the protocol. We begin with a description of the architecture that has been realized using the Repast Simphony framework, then we show how the visualization model has been accomplished, finally, we discuss the analyses that have been performed on the implemented model and the achieved results.**

***Index Terms*—Peer-to-Peer, Distributed Hash Table, Repast Simphony, Java**

I. Introduction

A fundamental problem in peer-to-peer (P2P) applications is the efficient location of the node where a specific data item is stored. This problem can be addressed using a distributed hash table (DHT), which is a distributed system where each node participating in the network can discover the value associated with a key in a few number of hops, also in large networks. An example of a DHT-based P2P application is represented by Chord [3], which can retrieve the position where the desired data item is stored in a network with N nodes in $O(logN)$ when the routing information is not out-of-date.

In Chord, each node of the network has a particular identifier which is obtained by hashing some properties of the node. Based on this value, nodes are arranged around a ring, known as *identifier circle*. Data items are distributed among nodes by hashing their value with the same hash function used for nodes. The node responsible for a given data item is the one that has the least identifier that is greater or equal than the identifier of the data item.

In order to scale to large networks, nodes in P2P applications cannot store the entire information of the system because it would require too much space. As a result, each Chord's node has partial knowledge of the system represented by a finger table and a list of immediate successors that are deterministically populated. In a system with N nodes, this finger table contains at most $O(logN)$ different entries and the list of successors is typically twice the size of the finger table. Since nodes are arranged in a ring, every node maintains the information about its predecessor and its successor. In particular, the successor is represented by the first entry in both the finger table and the list of successors. Although the correctness of the protocol is guaranteed as long as each node knows its correct successor, the entries of the finger table and successor list have to be periodically refreshed to accelerate lookups. This refreshing mechanism is realized through ad-hoc procedures that are executed periodically.

The lookup in Chord can be resolved both iteratively or recursively. An iterative lookup involves the same node requesting information contained in the finger tables and the list of successors from other nodes until it reaches the predecessor of the node storing the data item it is searching. On the other hand, a recursive lookup involves the nodes included in the lookup path forwarding a request until the predecessor of the node storing the data item is reached. The classic recursive style is expected to have a low latency when failures do not occur along the lookup path.

The implementation of Chord has been realized using Repast Simphony, an agent-based modeling framework which allows us to create an efficient simulator to test the functioning of the protocol in different environments.

II. Architecture

Rather than describing what has already been discussed in depth in [3], in the following sections we provide a description of the specific features that have been included in our implementation.

A. Identifier Space

In order to create a reliable simulation network, we assign a random IP address and port number to each node inside the network, in such a way that different nodes can not have the same combination. As suggested in [3], to obtain the Chord identifier of a given node, this combination of IP address and port number is provided as input to the SHA-1 hashing function. This consistent hashing generates a 160-bit hash value ranging from 0 to $2^{160} - 1$. As a result, the portion of the key space for which a node is responsible is more balanced. In our implementation, each Chord identifier is stored using the Java `Long` data type, which, starting from Java 8, can be used to represent unsigned 64-bit long values. Additionally, the `Biginteger` class is used to manage some mathematical operations which, due the magnitude of the numbers used, would have been wrongly handled otherwise. To avoid overflow errors, the Chord identifier of a given node is calculated as:

$$id = k \bmod 2^{64}$$

where id, k and 2^{64} denote the Chord identifier assigned to the node, the identifier produced by hashing the node's IP address and port number, and the maximum number of

values that can be represented using the Java `Long` data type respectively. As a result, our network is capable of maintaining 2^{64} nodes, making negligible the probability of two nodes hashing to the same identifier. Therefore, our implementation is potentially able to efficiently simulate a possible real large scale network of approximately $1.84 \cdot 10^{19}$ nodes. However, the Repast framework does not allow us to handle numbers that big, limiting the Chord ring to have a maximum of around $1.87 \cdot 10^{9}$ possible nodes.

B. Routing Information

Each node maintains a small amount of "routing" information, allowing the Chord protocol to scale well to a large number of nodes. In our implementation, this information is stored inside a finger table that contains at most 64 different entries, which is consistent with the $O(\log N)$, where N represents the number of nodes in the system, as described in [3]. Moreover, to increase the robustness of the protocol, each node stores a list of r immediate successors, where r can be decided at run-time by the user.

Due to the circular nature of the Chord ring and the data structures provided by Java, attention should be paid to the way these data structures are accessed. Given two nodes with Chord identifiers n and n' respectively, we want to understand whether a key k is within the range $(n, n']$. Depending on the position of the nodes inside the Chord ring, we need to distinguish between two cases:

- $n < n'$
- $n \geq n'$

In the first case, the node with identifier n has a lower identifier than n'. In this case, to determine whether k is between n and n', we need to solve the following system:

$$k \in (n, n'] = \begin{cases} true & (k > n) \wedge (k \leq n') \\ false & \text{otherwise} \end{cases}$$

For a better understanding, in fig. 1 is shown an example in which k is within the range $(n, n']$ when $n < n'$.

In the second case, the node with identifier n has a greater identifier than n'. In this case, to determine whether k is between n and n', we need to solve the following system:

$$k \in (n, n'] = \begin{cases} true & (k > n) \vee \\ & (k \leq n \wedge k \leq n') \\ false & \text{otherwise} \end{cases}$$

For a better understanding, in fig. 2 are shown two examples in which k is within the range $(n, n']$ when $n \geq n'$.

In our implementation, this method of determining if a given key k is within two nodes in the Chord ring is used several times during the various operations. For example, during a lookup operation, it is used by every node which wants to check whether its successor is the successor of a given key k. Furthermore, it is used to find the closest preceding node to a given key k in both the finger table and the successor list.

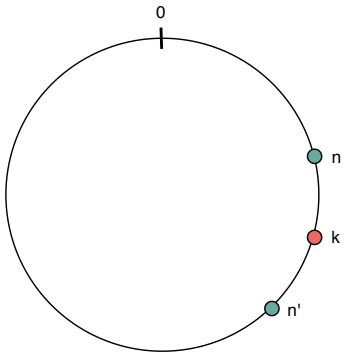


Fig. 1: A Chord ring where the identifier n of a given node is lower than the identifier n' of another node and the key k which n is looking for is between n and n'.

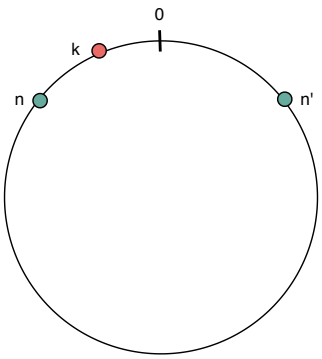
C. Lookup Strategy

The Chord algorithm examined in [3] uses iterative lookup. In this lookup method, the originator, which is a node that is trying to reach the successor of a given key k, uses its routing information to find the closest node preceding the key k and sends a request to it. When this node receives the request, it returns the closest node preceding the key k it is aware of. This information is then used by the originator to contact this new node which is closer to the desired key. This procedure is iterated until the node that precedes the key k is reached. Since the originator handles the complete routing traffic, it can easily keep track of the status of the request. As a result, failures occurred during the lookup can be detected very fast, allowing the originator to continue the request.

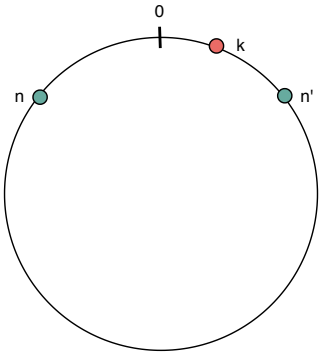
In the recursive lookup, which is mentioned but not examined in [3], the originator, using its routing information, sends a request to the closest node preceding the key k. In turn, each node involved in the lookup path forwards the request to the closest node preceding the key k according to its routing information. This procedure is continued until the node preceding the key k is reached. At this point, this node sends the successor of the key k back to the originator. While recursive routing as described in [1] is expected to perform better than iterative routing in absence of failures, its performance, in terms of network overload, should deteriorate in environments where nodes join and leave frequently since the routing information may contain nodes which are no longer participating in the protocol. In our implementation, to improve the recursive routing, every time a node receives a lookup request, it sends an acknowledgment back to the sender node as explained in [2]. This mechanism increases the number of messages circulating on the network, but we consider it a good trade-off to improve the performance of the protocol in the presence of not up-to-date routing information.

D. Failure Detection

In order to detect failures, each node maintains several tables where specific messages that have been sent are associated



(a) The key k which n is looking for is greater than both n and n'.



(b) The key k which n is looking for is lower than both n and n'.

Fig. 2: A Chord ring where the identifier n of a given node is greater than the identifier n' of another node and the key k which n is looking for is between n and n'.

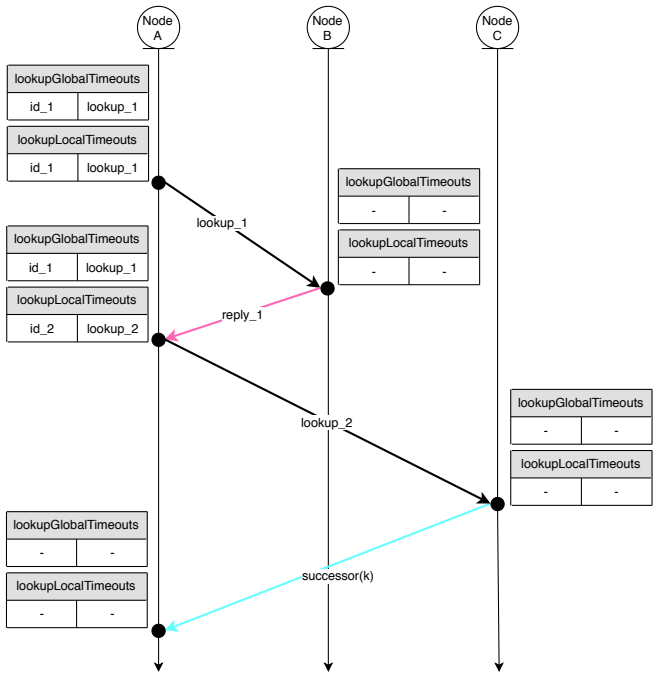


Fig. 3: A representation of how the timeout tables containing the lookup requests are handled during a single lookup performed using the iterative version of the protocol. The colors of the edges reflect what is described in sec. III. Node A wants to find the successor of a given key k, so it sends the request *lookup_1* to node B. After sending this message, node A, which is the originator of the lookup, adds this message in both its `lookupLocalTimeouts` and `lookupGlobalTimeouts` tables. When node B receives this request, it sends a reply message carrying information about node C to node A. Upon receipt of this reply, node A removes the *lookup_1* message from its `lookupLocalTimeouts` table, sends the request *lookup_2* to node C and adds this message in its `lookupLocalTimeouts` table. When node C, which is the predecessor of key k, receives the request from node A, it sends a message carrying its successor to node A. Finally, when node A receives the reply from C, it removes *lookup_1* message from its `lookupGlobalTimeouts` table and *lookup_2* message from its `lookupLocalTimeouts` table.

with the maximum application round they can be received without being considered expired. Depending on the type of message, each node stores:

- A table, referred to as `lookupLocalTimeouts` in our code, containing the lookup requests that this node sends as it is part of a lookup path.
- A table, referred to as `lookupGlobalTimeouts` in our code, containing the lookup requests that originate from this node.
- A table, referred to as `stabilizeTimeouts` in our code, containing the stabilization requests that this node sends to its successor to update both its successor list and its successor.
- A table, referred to as `predecessorTimeouts` in our code, containing the requests that this node sends to its predecessor to check whether it has failed or not.

There are some differences in the way the tables containing the lookup requests are handled between the iterative and recursive versions of the protocol.

As depicted in fig. 3, when the protocol runs in iterative style, the originator of a given lookup is the only node that forwards requests for this lookup. As a consequence, every time the originator sends a request to the closest node preceding the key k it is looking for, it adds the message to the `lookupLocalTimeouts` table. Only the first time it sends this lookup, it adds the message also to the `lookupGlobalTimeouts` table. Then, every time it receives a reply from an intermediate node, it removes the request this reply refers to from the `lookupLocalTimeouts` table. Finally, it removes the initial lookup contained in the `lookupGlobalTimeouts` when it receives the reply containing the successor of k.

As depicted in fig. 4, when the protocol runs in recursive style, the originator of a given lookup is no more the only node which sends requests for this lookup. Each node involved in the lookup path adds the message it forwards to the `lookupLocalTimeouts` table. Similar

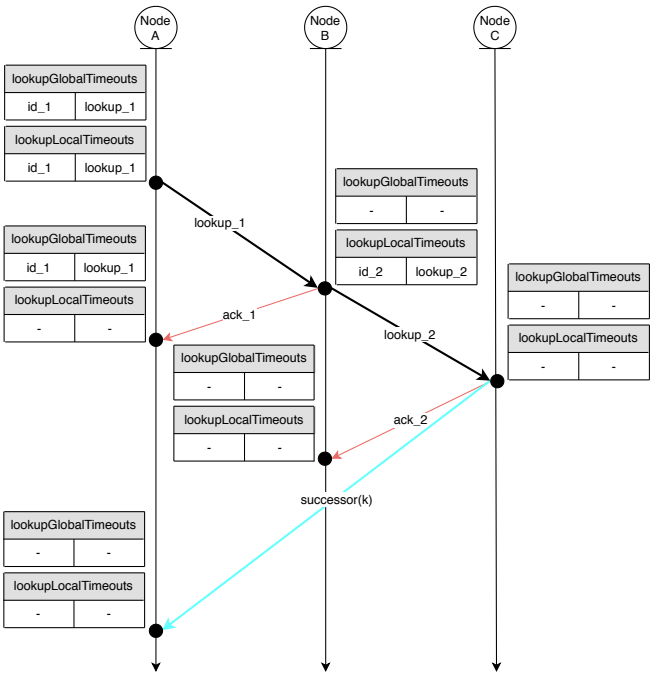


Fig. 4: A representation of how the timeout tables containing the lookup requests are handled during a single lookup performed using the recursive version of the protocol. The colors of the edges reflect what is described in sec. III. Node A wants to find the successor of a given key k, so it sends the request *lookup_1* to node B. After sending this message, node A, which is the originator of the lookup, adds this message in both its `lookupLocalTimeouts` and `lookupGlobalTimeouts` tables. When node B receives this request, it sends an acknowledgment back to node A, then forwards the request *lookup_2* to node C and adds *lookup_2* to its `lookupLocalTimeouts` table. Upon receipt of this ack, node A removes the *lookup_1* message from its `lookupLocalTimeouts` table. When node C, which is the predecessor of key k, receives the request from node B, it sends an ack back to node B and then sends a message carrying its successor to node A. Upon receipt of this ack, node B removes *lookup_2* message from its `lookupLocalTimeouts` table. Finally, when node A receives the reply from C, it removes the *lookup_1* message from its `lookupGlobalTimeouts` table.

to the iterative style, only the originator adds the initial lookup also to the `lookupGlobalTimeouts` table. Then, every time a node in the lookup path receives an acknowledgment, it removes the request this acknowledgment refers to from the `lookupLocalTimeouts` table. Finally, the originator removes the initial lookup contained in the `lookupGlobalTimeouts` when it receives the reply containing the successor of k.

At each time step, every process checks whether the entries inside these tables are expired or not. If an entry is expired, the destination of this message is considered as failed and the failure is handled accordingly to what is described in [3].

E. Node Join Optimization

An enhancement can slightly improve Chord performance when a node joins the network. While in [3], a joining

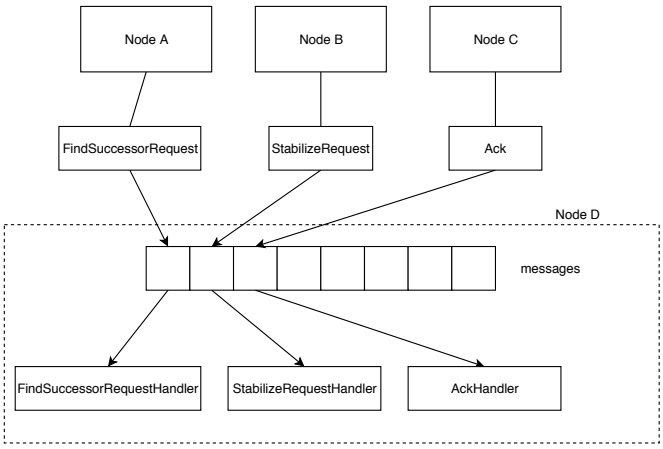


Fig. 5: A representation of the dispatching of messages to the corresponding handlers.

node n which has just found its successor s can notify s about its existence only during the stabilize procedure, in our implementation n notifies s immediately after discovering it. In this way, s can replace earlier its predecessor with n, allowing its previous predecessor p to adjust p's successor to n as soon as it either receives the `NotifyPredecessor` message if the corresponding optimization is enabled, or it runs the stabilize procedure.

F. Message Management

Since Repast Simphony is a framework designed to model general agent-based systems, it does not provide a mechanism to realize the communication between agents through messages. To address this issue, we introduce a layer for handling the exchange of messages between nodes independently from the protocol logic. As shown in fig. 5, each node maintains a buffer, referred to as `messages` in our implementation, which stores all the incoming messages that the node has not already handled. There are eleven types of messages:

- `FindSuccessorRequest`: a message sent by either an originator or an intermediate node to request the successor of a key. This message contains an attribute called `reqType` that represents the nature of the request. It can be `JOIN`, `FIXFINGER` or `LOOKUP`: the first means that the request refers to a lookup needed to find the successor of a node which is joining the network, the second is a request initiated by a node which is refreshing its finger table, while the last means that the request comes from the application layer of some node.
- `FindSuccessorReply`: a message representing the reply to a `FindSuccessorRequest`. This message has a field that indicates whether the reply contains the successor of the key the originator is looking for (`SUCCESSOR`) or an intermediate node to which the request has to be forwarded (`INTERMEDIATE`).
- `JoinRequest`: a message sent by a node, which is trying to join the network, to a node which belongs to the Chord ring in charge of finding its successor.

- `joinReply`: a message representing a reply to a `JoinRequest`. It contains the successor of the joining node.
- `StabilizeRequest`: a message sent to the successor to obtain the information about the successor's predecessor.
- `StabilizeReply`: a message representing the reply to a `StabilizeRequest` message. It contains the information about the successor's predecessor.
- `NotifySuccessor`: a message sent to the successor to notify it about the existence of the sender.
- `NotifyPredecessor`: a message sent by a node to notify its old predecessor that the sender's predecessor has changed.
- `CheckPredecessor`: a message sent by a node to its predecessor to verify whether it has failed or not.
- `VoluntaryLeave`: a message sent by a node to notify its successor and predecessor that it is voluntarily leaving the network.
- `Ack`: a message sent by a node to acknowledge the receipt of another message. For example, it is used during the recursive version of the protocol to notify the sender of a `FindSuccessorRequest` about the receipt of this message.

Similar to a remote procedure call, the sending of a message is simulated by calling the method `receive` of the destination node. This node is then responsible for the dispatching of the message to the appropriate handler. Every message includes a field, named `tick`, which denotes the tick during which the message is allowed to be processed by the receiving node. By varying this tick, we can simulate network delays.

G. Parameters

The main purpose of our implementation is to analyse how the Chord protocol behaves under certain conditions. For this reason, the graphical user interface (GUI) of the simulator provides the possibility to change some parameters which determine the behaviour of the protocol, such parameters are described below. Notice that the key space parameter is considered as a constant and it is not included in this list.

- *Default Random Seed*: default parameter of Repast. It describes the random behaviours, by using the same seed we can reproduce a single specific run of the model.
- *Initial Nodes Number*: the initial number of nodes in the network. The number of nodes may change later because of joining or leaving nodes.
- *Network Delay*: the λ parameter of the *Exponential* distribution ruling the delays in the network.
- *Protocol Mode*: denotes if the protocol has to be executed in iterative or recursive mode.
- *Successor List Size*: the maximum number of elements that can be contained in the successor list.
- *Lookup Generation Probability*: the probability that the application layer of each node in the network issues a lookup during every simulation round.
- *Failure Frequency Mean*: the mean of the *Poisson* distribution ruling the number of nodes that may fail at every simulation round.
- *Voluntary Leave Frequency Mean*: the mean of the *Poisson* distribution ruling the number of nodes that voluntary leave the network at every execution tick.
- *Joining Frequency Mean*: the mean of the *Poisson* distribution ruling the number of new nodes that try to join the network at every execution tick.
- *Stabilize interval*: the time interval, expressed in ticks, between the sending of two different `StabilizeRequest` messages.
- *Check Predecessor interval*: the time interval, expressed in ticks, between the sending of two different `CheckPredecessor` messages.
- *Fix Finger interval*: the time interval, expressed in ticks, between two different execution of `fixFingers` procedure.
- *Timeout Local Lookup*: the amount of time, expressed in ticks, after which a node considers as failed a node which has not replied to a previously sent `FindSuccessorRequest` message.
- *Timeout Global Lookup*: the amount of time, expressed in ticks, after which a node considers as failed a lookup.
- *Timeout Stabilize*: the amount of time, expressed in ticks, after which a node considers as failed a node which has not replied to a previously sent `StabilizeRequest` message.
- *Timeout Check Predecessor*: the amount of time, expressed in ticks, after which a node considers as failed a node which has not replied to a previously sent `CheckPredecessor` message.
- *Optimization: Notify Predecessor*: it enables the optimization of the protocol which consists in signaling the node's old predecessor that the node's predecessor has changed through the sending of a `NotifyPredecessor` message.
- *Dataset Type*: it denotes which data source has to be collected during the simulation. The data is stored in a csv file, which is located inside the main directory of the application.

III. Visualization

The Repast framework provides the possibility to visualize the interactions between agents to better understand how the protocol behaves using different parameters' configurations. In order to better analyze how a lookup request, coming from the application layer of a given node, is satisfied by the system, we display a network containing the nodes that participate in the protocol and the connections among them. As shown in fig. 6, this network is represented by a graph where nodes are arranged around a circle, in a similar way to typical Chord representations. The meaning of the various edges and nodes is explained as follows:

- *Small gray nodes*: nodes which belong to the Chord ring.

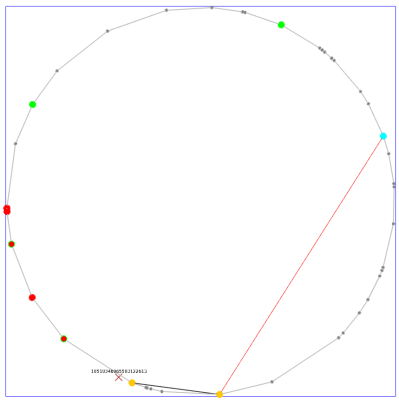


Fig. 6: A screenshot representing the appearance of the model during a lookup performed using the recursive version of the protocol.

- *Yellow nodes with black border*: nodes which are trying to join the network.
- *Big cyan node*: the originator of the considered lookup.
- *Big red nodes*: nodes which belong to the successor list of the node which has to find the closest node preceding a particular key.
- *Big green nodes*: nodes which belong to the finger table of the node which has to find the closest node preceding a particular key.
- *Big orange nodes*: the intermediate nodes which are involved in the current lookup.
- *Thin light gray edges*: the link between a node and its immediate successor
- *Black edges*: they represent the sending of a `FindSuccessorRequest` message.
- *Magenta edges*: they represent the sending of a `FindSuccessorReply` message, which is done by an intermediate node (only in iterative mode).
- *Cyan edges*: they represent the sending of a `FindSuccessorReply` message to the originator of the lookup, containing the successor of the key it is looking for.
- *Thin red edges*: they represent the sending of an `Ack` message.
- *Red X*: the position on the ring of the key the originator is looking for.

The visualization shows only one lookup at a time, randomly chosen among the active ones. As a result, even in the presence of many simultaneous lookups, only one of them is displayed. In this way, it is possible to better understand how the algorithm evolves also in networks with many nodes and a high lookup frequency.

The collection of the data to visualize is performed by a particular agent called `Visualizer`. When a node performs an action which has to be visualized on the display, it sends to the `Visualizer` all the necessary information.

IV. Experiments

The data collection process is performed using an agent of the network, referred to as `Collector` in our implementation. This agent is responsible for both the gathering of the data from the nodes of the network and the storing of this information in the appropriate data source. From the Repast GUI, we provide the possibility to select the type of event to analyse. By default, no event is selected. However, the following events can be monitored:

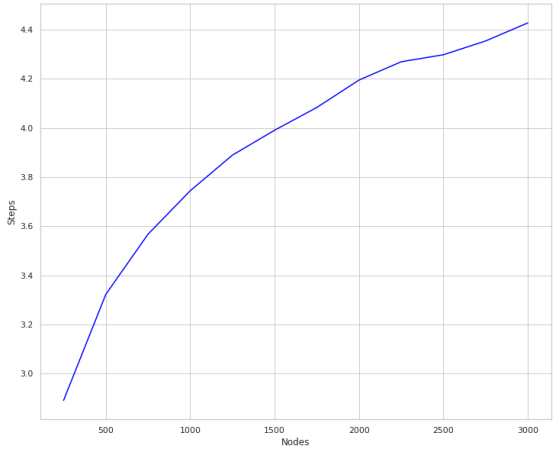


Fig. 7: A representation of the average number of steps needed to reach the predecessor of a key varying the number of nodes in the network. The graph shows that the path length is logarithmic with respect to the number of nodes.

- *Lookups-Time-Stable*: the time needed to complete each lookup operation by varying the number of nodes in the network.
- *Lookups-Time-Unstable*: the time needed to complete each lookup operation by varying the leave and join rate.
- *Lookup-Steps*: the path length needed to find the successor of a key.
- *Expired-Requests*: the number of requests which are received from nodes that other nodes thought were crashed by varying the local timeouts.
- *Stabilization-Period*: the number of nodes which have a wrong successor by varying the rate R at which nodes join and leave the network.
- *Stabilization-Interval*: the number of nodes which have a wrong successor by varying the stabilize interval.

When a specific event happens in a node, such as the receipt of a lookup reply, this node sends the required information to the `Collector` agent. In the next sections, we discuss the results observed from these analyses.

A. Average Lookup Path Length

In this experiment, we analyze the performance of the protocol in terms of lookup path length. A lookup path is defined as the sequence of nodes that have to be contacted to find the successor of a given key. Its length is given by the number of requests that have to be forwarded to find this key. Since this length can be used to determine how fast a lookup is

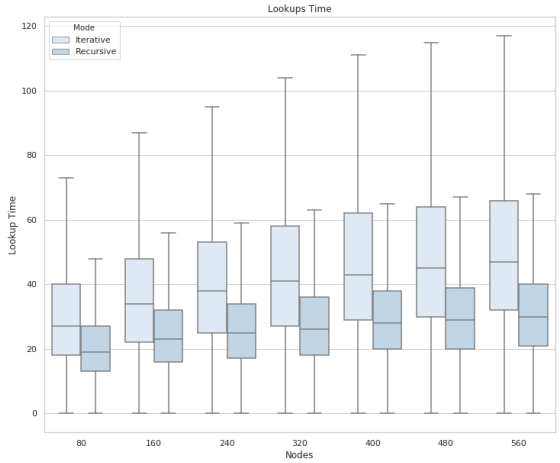


Fig. 8: A representation of the average time, expressed in simulation rounds, required to find the successor of a key. It is possible to see how the recursive version of the protocol has a lower latency than the iterative one.

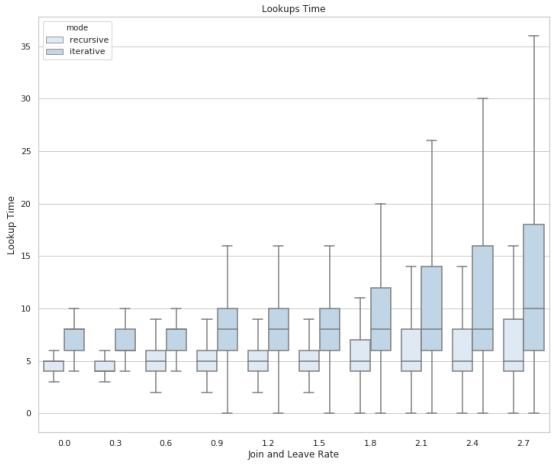


Fig. 9: A representation of the average lookup time varying the rate at which nodes can join and leave. The graph shows how the lookup time increases both in the iterative and recursive version of the protocol by increasing the instability of the network.

performed, we ran multiple simulations with different number of nodes and, for each simulation, we compute the average path length. Because we are interested in the analysis of the normal behaviour of the protocol, in this experiment, neither the leave and join of nodes nor the delay of the network are considered. The results obtained are shown in fig. 7 where it can be seen that the number of hops before reaching the predecessor of a key is logarithmic with respect to the number of nodes in the network, thus confirming what is claimed in [3]. The data collected to perform this analysis refers only to the iterative version of Chord, however, the lookup path length does not depend on the protocol mode because the number of requests forwarded in iterative and recursive are the same.

B. Average Lookup Time

In sec. II-C, we described the difference between the recursive and the iterative version of the protocol. In the iterative mode, the originator is responsible for forwarding all requests to every intermediary node, meaning that, before sending the next request, it has to wait for the reply of the previously contacted node. On the other hand, in the recursive version, the originator sends only one request and when the message reaches the destination it is immediately forwarded to another node. For this reason, on average, the time needed to find the successor of a given key is lower in the recursive version.

In order to verify this hypothesis, we performed some experiments monitoring the time elapsed between the sending of the first request of each lookup and the delivery of the message containing the successor. This experiment is performed using both the recursive and the iterative version of the protocol for different network sizes. In this analysis, we do not consider joining or leaving nodes since we are interested in the analysis of the lookup time without taking into account potential delays that these events may introduce. However, to obtain more realistic data, we model the communication delay between nodes using an *Exponential* random variable X with $\lambda = 0.2$. As a result, the expected value $\mathbb{E}(X)$ of the network delay is $\frac{1}{\lambda} = 5$. As expected, in fig. 8 is shown that the recursive mode has a lower latency than the iterative one. In this picture, it can be seen that the distribution of the average lookup time is spread on a larger time interval in the iterative version than the recursive one. Intuitively, the higher the number of consecutive messages that have to be exchanged before reaching the destination, the larger is the range of time the lookup may take. This intuition can be formalized considering the variance of the sum of multiple *Exponential* distributions.

Proof. Suppose that k is the number of messages that have to be exchanged before finding the successor of a given key. The time needed to exchange k messages is the sum of k independent *Exponential* random variables with the same mean $\frac{1}{\lambda}$. The sum of these random variables corresponds to an *Erlang* random variable with variance $\frac{k}{\lambda^2}$. Notice that the higher the number of messages that are forwarded (the larger is the value of k), the larger is the variance. □

C. Lookup Time in Unstable Networks

In the previous section, we analyzed the time needed to complete a lookup in a stable network. In an unstable network instead, the time required to find the successor of a key increases because it is possible that a node which belongs to the lookup path fails or leave the network. In this case, the failure is detected when the corresponding timeout expires and the node tries to re-send the lookup request to another one.

In fig. 9, it is possible to see how the lookup time increases with different join and leave rates. In the performed

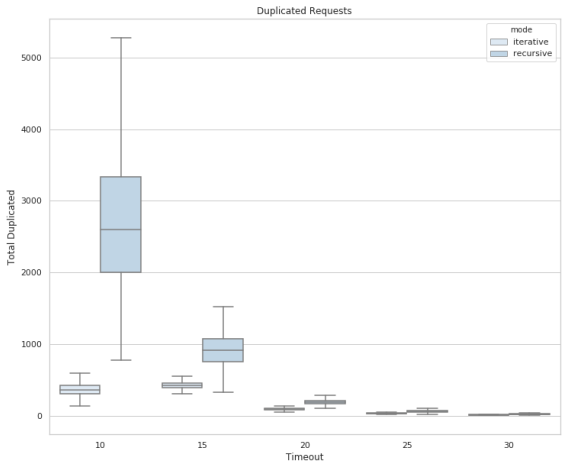


Fig. 10: A representation of the distribution of the requests received from nodes which were wrongly considered as failed, as a function of the timeout, for both the iterative and the recursive versions of the protocol. The timeout is defined as the maximum number of rounds a node waits before considering as failed a node to which it previously sent a message.

experiments, the network is initialized with 500 nodes and the mean of the random variables modeling the join, the leave and the failure events are increased keeping the same proportions: respectively $3/6$, $2/6$, $1/6$. Notice that the sum of the means of the leave and failure events is equal to the mean of the join event since, in this way, the number of nodes in the network remains stable during all the simulation and the lookup time is not affected by the change in the number of nodes. The resulting graph shows how the lookup time varies as the probability of join and leave increases and confirms what we have previously discussed: the higher the instability of the network, the longer the time a lookup requires to be accomplished. As previously explained in sec. IV-B, also in the case of unstable networks, the recursive version of the protocol is much more efficient than the iterative one.

D. Duplicated Requests

In sec. II-D, we explained how node failures are detected in our implementation. In short, at each step, each node iterates through the entries of some tables to determine whether the reply of a request has not been received within a given period. If the request has not been received within this time limit, the node assumes that the recipient of the request has failed. This timeout has to be carefully chosen since we want to minimize the number of nodes that are considered as failed but are not actually failed. A naive choice would be to choose a large timeout. However, while we want to minimize the number of false positive detections, we do not want to wait for too long to discover nodes that are failed. In order to find a good trade-off, we performed an analysis of the number of "duplicated requests", that is the number of requests received from nodes which were considered as failed, by varying the timeout.

Since we are interested in determining the number of requests coming from nodes considered as failed but are not failed, we consider a network with 500 nodes where nodes cannot join, leave or fail. We run the protocol for 500 application rounds using both the iterative and recursive versions for increasing values of the timeout. Fig. 10 shows the distribution of "duplicated requests" as a function of the timeout. This data is highly dependent on the network delay, which is modeled using an *Exponential* random variable X with $\lambda = 0.2$. As a result, the expected value of X is $\mathbb{E}(X) = \frac{1}{\lambda} = 5$, meaning that a message requires 10 application rounds on average to go to the recipient and come back. As expected, increasing the value of the timeout results in a smaller number of "duplicated requests". In fact, using a timeout lower than twice the round trip time results in a high number of false positive failure detections. In particular, it can be noticed how the recursive version of the protocol experiences a large number of "duplicated requests" because every time a node in the lookup path detects a failure, it tries to accomplish the lookup by sending the request to the closest preceding node, resulting in many nodes solving the same lookup. For this reason, the recursive version of the protocol generally experiences more "duplicated requests" than the iterative one.

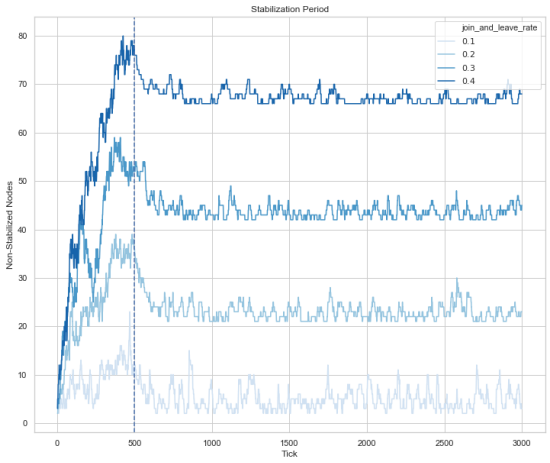
Given the results of this experiment, we suggest using a timeout parameter that is a little larger than twice the expected round trip time.

E. Stabilization Period

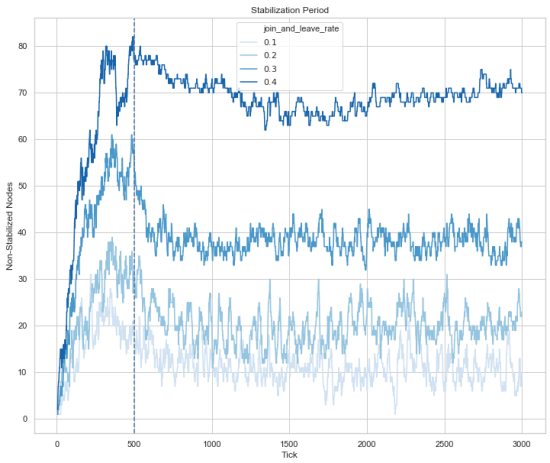
This experiment aims at showing the period required by the stabilization procedure to stabilize the network. Here, a node is intended to be stabilized when it has the correct information about its successor.

We consider a network with 100 nodes, where each node can join and leave the network with a rate R ranging from 0.1 to 0.4. As we are interested in determining the stabilization period using the same rate for leave and join, we do not consider nodes which can fail since, otherwise, the network would lose many nodes during the simulation. In particular, nodes are allowed to join and leave the network only during the first 500 application rounds. This experiment is performed for both the iterative and the recursive version of the protocol.

Fig. 11 shows the number of nodes which have the wrong successor information as the simulation proceeds. As expected, increasing the rate R results in a network where more nodes have the wrong information about its successor. In addition, it shows how much the network requires before reaching a point in which the number of nodes that have not the correct successor does not decrease any more. This number does not reduce because of the clusters that are formed in the network. As previously mentioned, nodes in this network do not fail, meaning that the fluctuations that can be observed are due to nodes which erroneously believe that some other nodes are failed. This happens because of the delay in the communication between nodes and the timeout tables



(a) Iterative version of the protocol.



(b) Recursive version of the protocol.

Fig. 11: A representation of the number of nodes that have the wrong information about their successor in a network with a rate R at which nodes join and leave ranging from 0.1 to 0.4, for both the iterative and recursive version of the protocol.

described in II-D. Also, it can be noticed that there is not much difference between the two versions of the protocol because the stabilization procedure only involves the exchange of messages between a node and its successor.

F. Stabilization Interval

In order to find a good setting of the stabilization interval, we performed an analysis of the number of nodes with the wrong successor information varying the interval of the stabilization procedure. The choice of this interval is a trade-off between the number of messages that are exchanged in the network and the freshness of the successor information.

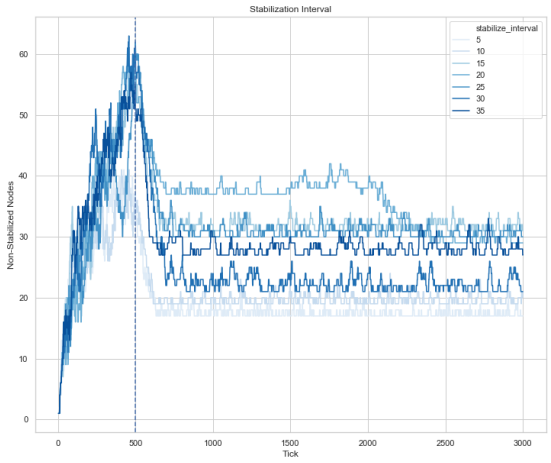


Fig. 12: A representation of the number of nodes that have the wrong information about their successor for different interval values of the stabilization mechanism.

We consider a network with 100 nodes, where each node runs the stabilize procedure according to a fixed interval which can take one of the following values: $5, 10, 15, 20, 25, 30, 35$. The join, leave and failure events of a node are modeled using three *Poisson* random variables: $X_1 \sim Pois(0.3), X_2 \sim Pois(0.2), X_3 \sim Pois(0.1)$ respectively. Since the sum of three *Poisson* random variables is a *Poisson* random variable $Y \sim Pois(\lambda)$ with mean λ given by the sum of the means, we expect a variation in the structure of the network with a rate of 0.6. We consider this value a good rate to test the effectiveness of the stabilization interval. In particular, these variations are done only during the first 500 application rounds. This experiment is performed only for the iterative version of the protocol since, as shown in fig. 11, the version of the protocol does not affect significantly the stabilization. Finally, given the results of fig. 10, we use a timeout of 30 application rounds.

Fig. 12 shows the number of nodes that have the wrong successor information as the simulation proceeds. As expected, the lowest values of the stabilization interval allow the nodes to have the most up-to-date successor information. However, in an unexpected way, a stabilization interval performed every 30 or 35 application rounds allows nodes to have more fresh information than stabilization intervals of 15, 20, or 25 application rounds. This suggests a dependency between the stabilization interval and the timeout, which has been examined in sec. IV-D and rules the detection of failed nodes. In particular, in our implementation, when a node detects a failed successor, it automatically adjusts the successor information. Stabilize messages that are sent to a failed successor, before the failure is detected, will be lost and will be sent again a long time after the failure is detected, resulting in possibly wrong successor information for more application rounds. On the other hand,

stabilize messages that are sent after the failure is detected, will be sent to a node which is more likely participating in the network. It should be noticed that in this experiment nodes are not allowed to join, leave or fail after the first 500 rounds. However, since the delay of the network is modeled with a random variable, it might be possible that nodes erroneously consider as failed some other nodes.

Given the results of this experiment, we suggest using a stabilization interval parameter that is a little larger than the timeout used to detect failed nodes. We do not suggest to use too low intervals since this would result in a large number of exchanged messages, overloading the network.

V. Conclusion

The Chord protocol has been designed to localize where a specific data item is stored. It seems particularly useful for P2P networks where each node shares data and wants to retrieve information from other nodes. In this work, we have implemented Chord using Repast Simphony, an agent-based modeling framework. We described the main features that we have introduced in our implementation to bridge the gap between the theoretical aspects of the protocol and its practical application. In particular, we described how we achieved to deal with a large number of nodes in the network; we discussed the method used to determine whether a given key is between two nodes in a ring without using circular data structures; in addition to the iterative version, we implemented the recursive one, improving its functioning through the use of acknowledgments; we also realized a mechanism to detect node failures, which was only briefly described in [3]; finally, we introduced a mechanism to allow the communication between nodes.

From a qualitative analysis of the simulations, our implementation works very well even with a large number of nodes by satisfactorily completing the lookups. The hard part consists in the determination of the parameters that allow the protocol to work in the best way. This is the reason why we focused some of our quantitative analyses on the learning of the timeout and stabilization interval values. With the right setting for these parameters, it is possible to observe how the network is able to adjust its structures after being altered by a high number of joins, leaves, and failures. We also observed that the recursive version of the protocol requires a lower latency than the iterative one to reach the successor of the desired key when the routing information is up-to-date since nodes in the lookup path do not need to wait for a reply before forwarding the request. However, with a high rate at which nodes join, leave, and fail, the recursive version could cause many redundant messages circulating in the network. In fact, we observed a high amount of duplicated requests when nodes erroneously consider other nodes as failed.

In order to overcome the issues faced by the recursive version, we might introduce in our implementation an additional mechanism that stops nodes in the lookup path forwarding a request that has been marked as failed by its originator.

References

[1] Eng Keong Lua et al. “A survey and comparison of peer-to-peer overlay network schemes.” In: *IEEE Communications Surveys and tutorials* 7.1-4 (2005), pp. 72–93.

[2] Alberto Montresor. *Distributed Algorithms: Peer-to-Peer Systems*. University Lecture. Oct. 2018. URL: http://disi.unitn.it/~montreso/ds/handouts17/07-p2p.pdf.

[3] Ion Stoica et al. “Chord: a scalable peer-to-peer lookup protocol for internet applications”. In: *IEEE/ACM Transactions on Networking (TON)* 11.1 (2003), pp. 17–32.

APPENDIX A
HOW TO INSTALL

Repast Simphony is an agent-based modeling framework that allows users to develop their own models. Repast Simphony applications can be developed in Java. It provides a graphical user interface (GUI) to display the state of the model simulation.

In this appendix, we provide installation instructions to simulate "chord", a distributed hash table protocol implementation developed using Repast Simphony.

You must have installed the Java Runtime Environment (JRE) to install the model simulator. If the JRE is not on your computer, see Install-Overview for a full explanation on how to install it. Ensure to install Java 11 or any later version, otherwise, the model will not be executed properly.

Once you have installed the JRE, download the model from Chord-Model. Go to the folder where the `chord_model.jar` has been downloaded and double-click on it. An installation wizard will be displayed on the screen. Select the language of the installation in the "Language Selection" dialog box and select "OK" to go to the "Installation of chord" screen. Select "Next" to read the information about the model. Then, select "Next" again to open the license agreement. Review the license agreement before proceeding. If you agree to the license terms, select the "I accept the terms of this license agreement." button, then select "Next" to choose the installation folder. By default, the software will be installed in `C:\ProgramFiles\chord`. You can choose a different location by clicking on the "Browse..." button, navigating to the desired folder and selecting it. Ensure the selected folder is empty, otherwise its content will be overwritten. Select "Next" to decide which packs to install. Leave the default setting and select "Next" to start the software installation.

Once the installation is finished, select "Next" to open the "Setup Shortcuts" screen. If you want to create a shortcut in the Start-Menu leave the default setting, otherwise remove the flag from the "Create shortcuts in the Start-Menu" checkbox. Select "Next" to go to the final installation screen. Unless you need to install the software in other computers, you do not need to select "Generate an automatic installation script". Finally, select "Done" to terminate the installation procedure.

If you created a shortcut in the previous steps, you can run the model by searching for the shortcut on the Start-Menu and single-clicking on it. Otherwise, go to the folder where you installed the model and double-click `start_model.bat`. The Repast Simphony GUI will be displayed. You are now ready to run the model.