Implementation of the Chord algorithm using RePast Symphony

Michele Armellini
203828
michele.armellini@studenti.unitn.it

Marco Merlin
205263
marco.merlin@studenti.unitn.it

1. Introduction

In this report are described the implementation of the Chord protocol for lookup operations in distributed environments and the comparison of the results obtained with the results shown in the original paper. This work is part of the laboratory assignment of the distributed system 2 course offered by the University of Trento.

The Chord algorithm is a distributed protocol for peer to peer distributed systems which allows to distribute an hash table among its members and perform lookup operations at any time while maintaining consistency of the hash table and the protocol. To implement the algorithm and simulate the distributed environment the Repast Symphony[1] framework was used, allowing us to simulate the protocol participants, the messages exchanged between them and all the possible events which could happen in a real setting such as joining, leaving or crashing nodes and hash table lookups.

This document first briefly discusses how the algorithm is supposed to work according to the original paper [2], it then describes in greater detail how the protocol was reproduced in Repast and how each routine of the algorithm was implemented in our environment. Finally the document describes the results obtained with our implementation and compares them to the results shown in the original paper [2]. The installer is found at `https://drive.google.com/open?id=1ZRDxogQsoBkdjIKEaN9vmeciCVkqV74_`

2. Implementation

This section briefly describes the Chord protocol and the routines employed by each node to keep the hash table consistent. It then describes the implementation of the protocol in Repast[1] together with some critical decisions made to make the simulation easier to implement.

2.1. Algorithm description

The original paper describes the Chord protocol as a consistent hashing algorithm which compared to other similar algorithms does not need each node to know most of the other nodes in the system at any time. As a consistent hashing algorithm, Chord allows the application level to perform a lookup operation which given an hash value returns the node which currently owns that hash value so that the actual value associated with such key can be retrieved at a later time. The structure of a Chord environment is a ring composed by the participants of the protocol where the ring represents all the possible hash values which a key can assume from the smallest to the biggest. Each node also has a hash value which represents their position on the Chord ring. The ownership of a key is defined by the position of a node on the ring where each node owns the keys in between its hash value and the hash value of the preceding node in the ring. As an example, on a ring going from value 0 to 16 having two nodes with hash 2 and 14, node 2 would own key 15, 16, 0, 1 and 2 while node 14 would own all hashes from 3 to 14 included.

The lookup process can be initiated by the application level on any node on the ring. The most basic lookup process consists of the node initiating the lookup checking whether the requested hash key is in between its own hash and its successor's hash. If it is, the lookup is complete, otherwise the lookup is carried out recursively to the closest node clockwise on the ring, also called the successor, which should repeat the same steps and return a result to the original node. This basic process allows the lookup function to work while each node only stores information about a single other node being its successor. A more efficient approach proposed in the original paper [2] requires each node to also store a finger table containing ownership information of hash keys at a logarithmically increasing distance. In the previous example, node 2 would store the ownership information of keys 3, 4, 6 and 10. The finger table can help to greatly speed up the lookup process so that instead of asking for a key to its successor, a node would ask for a key to the owner of the finger entry preceding the queried key value. The size of the finger table can be adjusted in order to prioritize the amount of information stored per node and the reduction of overall lookup time, however the finger table obviously has an upper size limit imposed by the amount of hashes values present in the Chord ring. A ring with 256

values should not have finger tables with more than 8 entries. The performance gain with finger tables depends on the number of fingers each table contains. A finger table of $\log_2 M$ entries in a 2^M valued ring would lead to a reduction in the maximum nodes queried per lookup from N to $\log_2 N$ where N is the total number of nodes in the system.

Together with the lookup function, the original paper also describes several sub routines which every node run periodically to maintain the hashing consistent even in the presence of events such as nodes joining, leaving or crashing. The main routines are stabilization, notification, finger fix and predecessor check. The stabilization routine is run periodically and consists into checking whether a node's successor is still its successor or if another node joined in between them. After running this routine, the notify routine is called on the successor to let him know about its predecessor. Predecessors are required so that the stabilization procedure can be performed correctly. The fix finger routine is also run periodically by each node to update the finger table so that joining and leaving nodes do not disrupt the lookup operations. Finally the check predecessor routine is used to just detect whether the current predecessor has crashed. The original paper also describes other routines such as the join and leave function which are called by a node whenever it either joins or leaves the system. Finally the Chord protocol also requires each node to store at least log N successors in order to make the whole system resilient to random crashes. This is required since the Chord protocol requires each node to always have a non crashed successor node. By storing the next log N nodes a system guarantees minimal chances for a node to have all of its successors crashed at once. All the routines and functions described in this subsection are discussed in greater depth in subsection 2.3.

2.2. Graphic Environment

To make the simulation more visually pleasing and introducing delays in the delivery of messages, we decided to visually represent all the nodes on a circle as blue dots, where the node's position on the circle corresponds to its assigned hash value. When a node performs a lookup operation, the query to other nodes is visually represented as an orange square moving from the sending node to the receiving node. The message propagation time serves as a delay simulation in the communication between nodes.

2.3. Algorithm implementation

This subsection discusses in detail our algorithm implementation in Repast and several choices we took in order to make the simulation possible. One critical decision taken to make the simulation easier was the way we treated communications between nodes. The original paper describes all the functions and communications between nodes as remote procedure calls. In reality these procedure calls would still behave as messages having a propagation time and thus being asynchronous with respect to all the routines run on each node. To simplify this, we implemented all remote procedure calls as blocking for the whole simulation and thus having no propagation time and consequently making the simulation easier to implement. The only exception is made for the lookup remote procedure call which instead behaves as it would in a real environment. Introducing asynchronous behaviour for all remote procedure calls would not disrupt the functioning of the Chord algorithm as it is built to withstand node crashes, furthermore, the original paper puts a bigger focus on the lookup operation compared to all other routines auxiliary to the correct functioning of the protocol.

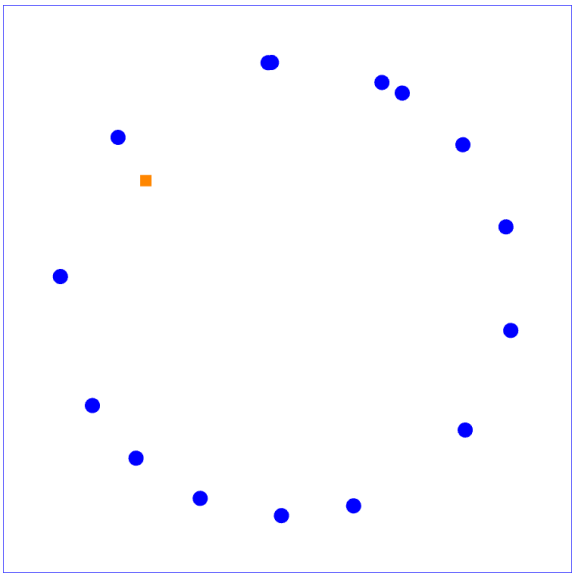

Figure 1. Screenshot taken from the simulated environment showing one of each entity described in Section 2.2

A single node stores only fundamental information such as a list of its closest successor, a predecessor node reference and its own hash. With each hash value owned by a certain node, there may also be an associated value. To simulate this, our nodes also keep a list of random generated values associated with a hash they own. The lookup procedure is not supposed to perform the retrieval of such values, however we choose to implement them anyways because these values are also involved in ring events such as leaving nodes, which are supposed to pass all of their values to the successor node before leaving the ring. Finally, each node also stores an array of outgoing lookup requests with an associated timer. This is needed to keep track of the time taken for a queried node to reply so that, in the event of crashes, requests can be purged if too much time has passed.

As described in subsection 2.1, the lookup procedure carries out recursively since each node passes the query to

nodes in its fingers until some node finds the owner being its own successor. Similarly to DNS queries, the paper also describes an iterative method which sees the node starting the query for a key iteratively asking other nodes, based on its fingers, who the owner is. The main difference between the two methods is that the second is more resilient to failures as with the recursive method even one node failure along the lookup chain would compromise the query operation. Furthermore the recursive method would require each node to keep track of all queries forwarded to other nodes while the iterative method only requires the starting node to keep track of the forwarded requests. For these reasons we choose an iterative approach for our implementation. Our implementation of the lookup procedure starts from the querying node which as a first step checks whether the queried key belongs to itself or to its successor. If it doesn't, the node sends a request message to the next node by consulting its finger table. After a short delay, dictated by the propagation time, the receiving node reads the request and checks whether the required key belongs to itself or its successor, if it doesn't it consults its own finger table and returns the closest node to the sender. Once the starting node receives the reply, it sends out another request to the suggested node and the cycle repeats until some node determines that itself, or its successor, is the key owner. As stated by the original paper, the lookup procedure will work even with incorrect fingers as the request will just be propagated around the circle until nodes with correct finger tables can correctly suggest the key owner. The lookup procedure can be disrupted in the case where the successor pointers are not updated as a result of a joining node. This case is not handled by our simulation since successors and predecessors are always up to date because of blocking routine functions which ensure that when a node joins, all predecessors and successors are updated instantly. In a real scenario the lookup query would simply time out and the application layer could just retry the operation after some time.

The ring stabilization routines also have a big part in the algorithm since they are required to make the look-ups consistent. The most important function is the stabilization routine which serves the purpose of updating the successor list of each node in the event of members joining, leaving or crashing. Since the whole correctness of the Chord protocol lays in having updated successors, this function is greatly relevant. Our implementation of the stabilization function starts by removing any of the successors in the successor list which have crashed since the last stabilization. The function then retrieves its saved successor's predecessor and checks whether it still is itself. If the successor's predecessor is not itself and its clockwise hash distance is less than the distance of the successor, then the node sets the successor's predecessor as its new successor, otherwise it just keeps its current successor. After setting its new successor, the stabilize function copies the successors list of its successor into its own exception made for the last element which is discarded. Finally the stabilize function calls the notify function on its successor setting a reference to itself as parameter. The notify function is used to correctly set a node's predecessor and is only called by a node's most recent predecessor which just completed its stabilization function. Our implementation of the notify function firstly checks whether the current node's predecessor is null or has a greater clockwise distance from the notified node compared to the node passed as parameter and if any of these cases are true the node sets its predecessor to be equal to the passed node. When a new predecessor is set the notified node also passes some of its elements to the new predecessor based on its hash and which elements it should own. After a node has completed its stabilization function and its successor has been notified, these two nodes will be consistent with respect to each other. Another short but necessary routine is the predecessor checking function which simply checks whether the current predecessor has crashed and if that is the case, set it to null.

The fix fingers procedure is also very important and allows each node to update its finger table. This procedure iterates over all finger values and updates their values based on the same procedure used for look-ups where the updating node asks non crashed nodes in its current fingers for the owner of a certain hash. The main difference is that since this procedure is blocking it carries out recursively, meaning that when a node is updating its fingers it starts a recursive query to other nodes which retrieve the owner on his behalf. Updating the finger table using finger tables may seem counter intuitive, however since the successors are guaranteed to always be correct, the recursive search will eventually end correctly even if all finger tables of all nodes are wrong. Because fingers always update correctly, the fix finger procedure guarantees correct finger tables once its complete. It should be noted that our implementation allows for only one node to exist and work correctly in the chord ring. To allow this, some routines had to be slightly adjusted: the stabilization function, after removing all crashed nodes, checks if there is any successor left and if there are no successors, meaning that there is only one node left in the ring, it sets the successor to itself. The notify function before terminating needs to check whether the current successor is the node itself meaning that a second node joined the ring and just called the notify function on the only other node in the ring. If this is the case the notified node needs to set the second node as its successor which is only done in this edge case.

The join, create and leave routines are used to manage nodes performing such actions. The join function takes a node already present in the ring as a parameter to which the joining node can ask who its successor is and once the

successor is set the node can run the stabilize and fix fingers routines to initialize itself. Create is a similar function which is called on the first node of the ring and only differs from the join in setting the node's successor to itself and not calling stabilize since there are no other nodes. The leave routine, differently from a node failure, allows a leaving node to transfer all its elements to its successor before leaving. Lastly the update pending routine is used to update timers on requests towards other nodes. This routine keeps track of how much time has passed since each request was sent and it eliminates pending requests which take too long to return. This is needed since all requests sent towards crashed nodes may never have an answer and need to be purged.

Finally, to orchestrate the whole simulation, we used a global manager approach where a single object determines every event in the system. Initially, the manager creates the chord ring by inserting nodes one at a time. The way nodes are inserted is by making them join the Chord ring one by one, exception made for the first node on which the create function is called instead. Once the ring is formed, the manager starts its main loop where different events are started at regular intervals. Repast applications are regulated by ticks, and each application cycle the tick count is increased by one. In the main loop, each tick the manager has a very small chance to crash a random node, add a new node or remove an existing node from the ring. Each 50 ticks the manager calls on each node the predecessor check routine, the stabilize routine and the fix fingers routine in this order. The manager also calls the update pending routine at each tick to update requests timers. Lastly, each 200 ticks the lookup procedure for a random element at a random node is called, but first the manager checks whether the ring is in a stable condition, meaning that all nodes should have a successor and a predecessor set. This last check is not really needed in our simulation since the stabilization and check predecessor procedures are run every 50 ticks, meaning that they are always run right before a lookup making the ring always stable. However, if the lookup operation were run each 120 ticks for example, this function would prevent the lookup in the event that a node crashed and the ring has not yet stabilized.

3. Implementation Analysis

In this section this implementation is compared to the original one, highlighting the differences and making some theories of what causes different behaviours. Mainly two different aspects will be evaluated: the load balance of nodes and the path length of look-ups.

3.1. Load Balance

In the original paper[2], the ability to consistently allocate the keys to nodes in an even way is tested. The ideal distribution in a network with *N* nodes and *K* keys would be *N/K*, giving to each node the same number of keys of every other node. The way this is done in the original paper is by creating a network of 10^4 nodes and by varying the total number of keys in the network from 10^5 to 10^6 with increments of 10^5. For each increment the experiment was repeated 20 times with different random number generator seeds. Each experiment, the number of keys assigned to each node is counted.

In our case this numbers were not possible to test due to the ways we decided to implement the network initialization and to the fact that everything was run on the same thread, while in a real life scenario each node would be a separate machine. For this reason we decided to change the numbers by scaling down everything by 10 in order to maintain proportions between numbers. Our experiment consisted in a network of 10^3 nodes and the total number of keys went from 10^4 to 10^5 in increments of 10^4. In both plots the mean value is represented by the small black dot, together with the 1^{st} and the 99^{th} percentile represented by the vertical bars.

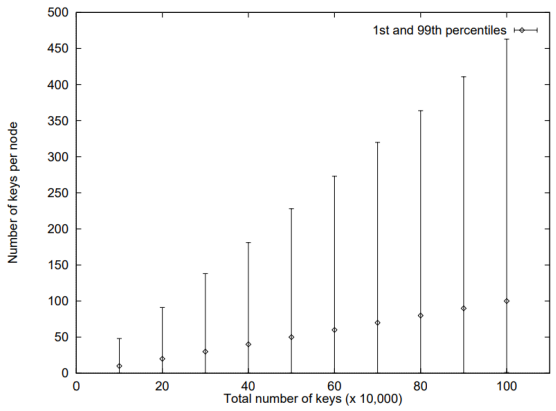

Figure 2. Mean, 1st and 99^{th} percentile of the number of keys stored per node in a 10^4 node network in the original paper

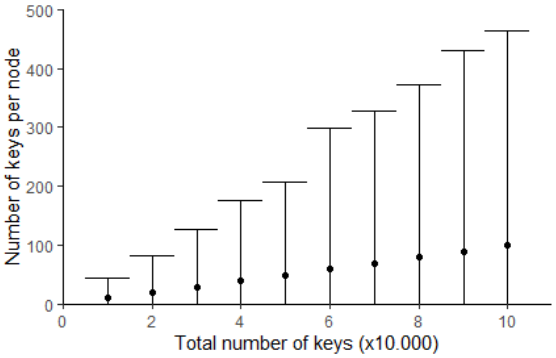

Figure 3. Mean, 1^{st} and 99^{th} percentile of the number of keys stored per node in a 10^3 node network in the implementation discussed in this document.

By comparing the two graphs we can see the values are similar to each other, except for the mean value which is slightly lower in our case. This, however, is not a real indicator of the real performance of the algorithm since there is a high variance in how the nodes are distributed. For example two nodes could be initiated with an id next to each other, making it hard for them to have the same number of keys.

Another analysis was made on load balance by computing the PDF[1] of the number of keys per node by running the experiment with $5 * 10^5$ nodes.

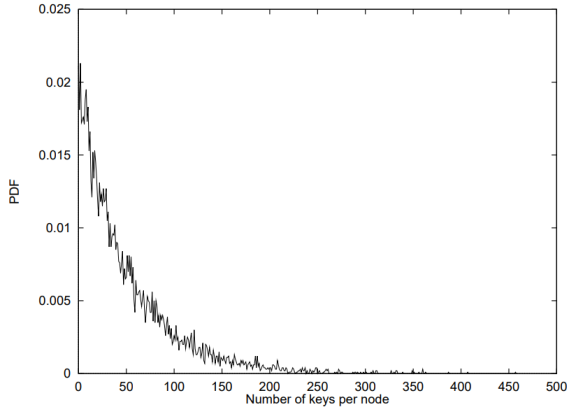

Figure 4. The probability density function(PDF) of the number of keys per node in the original paper

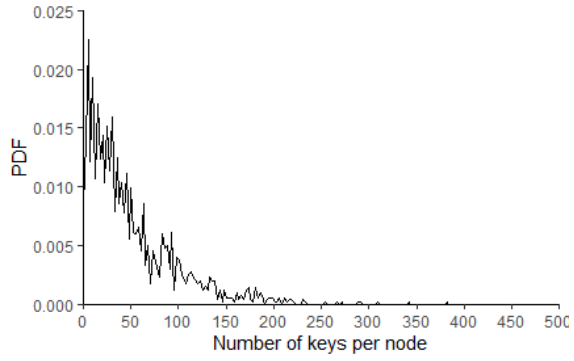

Figure 5. The probability density function(PDF) of the number of keys per node in the implementation discussed in this document.

By comparing the two graphs, there is a slight difference in how the curve decreases. In the original implementation the curve is more regular but decreases faster, as can be seen by the probability a node has to have 50 keys. While the irregularity of the implementation discussed here means it is more probable for different nodes to have a different value, it is also true that it is more probable to have more keys in each node. In the original paper the maximum number of keys stored in a node was 457 (9.1 times mean value), while in our implementation this number was 382 (7.64 times mean value). 99^{th} percentile in the original paper was 4.6 times the mean value while in our case it is 4.14. This means that in our case we were less likely to have a node with a really high number of keys compared to the mean value and, while it could be caused by the randomness of the data and the more data gathered for the original work, it could mean our approach distributes keys in a slightly better way.

[1]Probability Density Function

3.2. Path Length

The second value analyzed in the original implementation is the path length. The path length represents the number of nodes that are visited to resolve a query. This number is proved in the original paper to be O($\log N$), where N is the total number of nodes in the network. In order to better see the actual results of the algorithm, a network of 2^k nodes storing $100 * 2^k$ keys is simulated with k ranging from 3 to 14 for the original paper and from 3 to 12 for this implementation. The difference in the cases tested is because, given how we implemented the algorithm, especially the initialization, it is impractical for us to simulate a higher number of nodes as it would take a whole day just to finish initiating all the nodes. A random set of queries is picked from the system and for each of them the path length is measured. In both plots the mean value is represented by the small black dot, together with the 1^{st} and the 99^{th} percentile represented by the vertical bars.

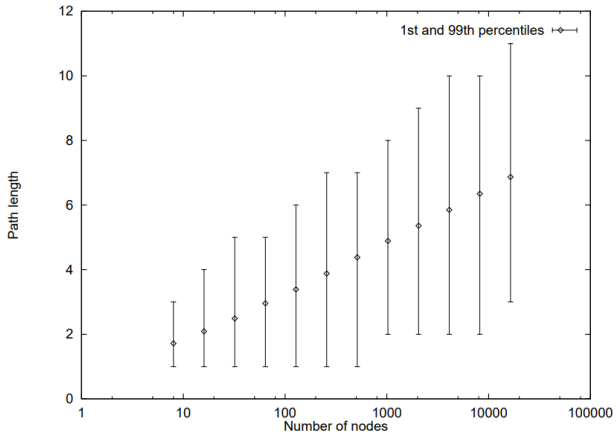

Figure 6. The path length as a function of network size in original paper

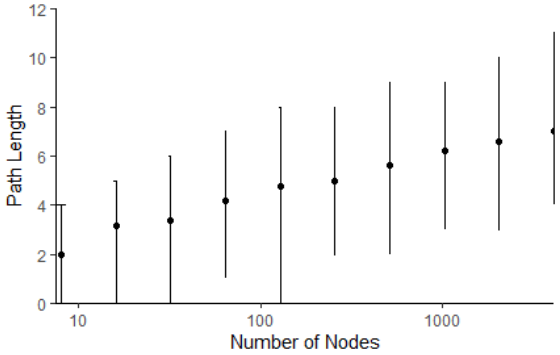

Figure 7. The path length as a function of network size in this implementation

By confronting the two graphs, we can see that the results are similar for both implementation. The main differences are, except from the fact that the last two series of data are missing from our results for the aforementioned reasons, that the values obtained by the original paper have a slightly lower mean value in general, while having a larger range of values in the cases with more nodes.

As with the analysis of load balance, the PDF is also analyzed. This is done by looking at the data with $k = 12$.

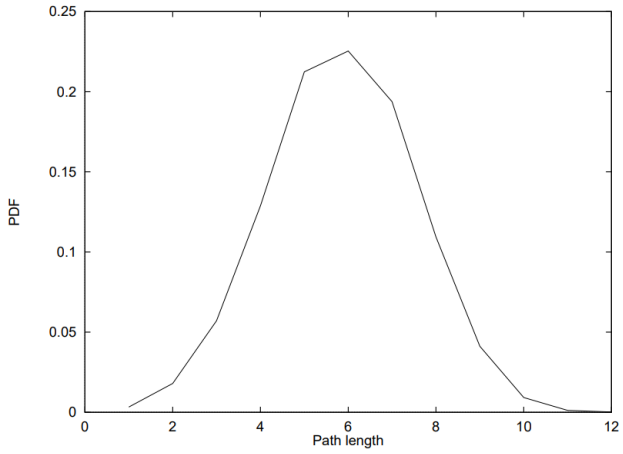

Figure 8. The PDF of the path length in the case of a 2^{12} node network in the original paper

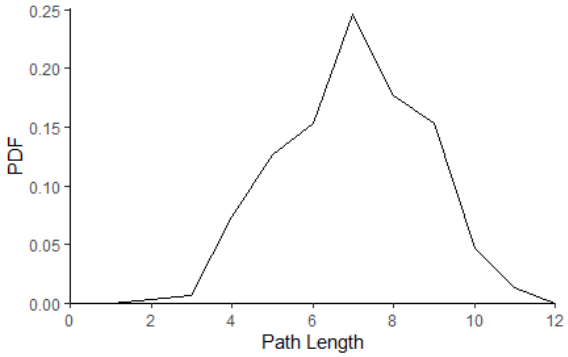

Figure 9. The PDF of the path length in the case of a 2^{12} node network in this implementation

While the results shown in the original paper represent a nice looking distribution which resembles a Gaussian, it is somewhat comparable to the results obtained in this implementation since the mean value is around 6 in the original implementation around 7 for our implementation. The variance is also similar since the curve is distributed between 1 and 12. The weird shape of the curve obtained from our implementation is probably due to a lack of data. This is due to the fact that the simulation is really slow with that number of nodes and makes the gathering of data impractical.

4. Conclusions

To conclude in this paper we presented the implementation of the Chord[2], a scalable peer-to-peer lookup protocol and an analysis of the results obtained in a simulated environment.

In Section 2 we discussed the general mechanics of the algorithm and how they were implemented in the Repast framework with a focus on explaining the changes we thought necessary in order to perform the simulation. The results presented in Section 3 show results compatible to the ones obtained in the paper, meaning that we were successful in recreating it correctly.

References

[1] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1):3, Mar 2013.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.