Chord: a scalable peer-to-peer lookup protocol for internet applications

Marian Alexandru Diaconu, Valentino Armani

January 2020

The following report explains the implementation of *Chord* algorithm. Moreover, this document does not describe specifications neither contains pseudocode of the theoretical algorithm, hence for further details of the protocol, please refer to [2].

1 Multi-Agent System

The Chord protocol is modelled as a Multi-Agent System (MAS). Each participating peer corresponds to an agent which has partial knowledge of the network and is in charge of responding to keys lookup. Nodes interact via messages in order to move forward local and global state. Messages are transmitted through the network using a router, which allows total encapsulation of a node state with respect to other nodes. On top of this basic simulation environment, we integrate a virtual application layer using a special agent which does not participate to the protocol. This agent is called Chord Node and acts on behalf of all nodes simulating application-dependent behavior such as key lookup generation and statistics computation.

1.1 Actions Scheduling

The Repast Symphony scheduling mechanism allows managing execution of actions according to a simulation clock. The clock, measured in "ticks", is incremented after the execution of all the actions scheduled at that clock. Specifically, all the protocol-related actions of the agents are scheduled according to Table 1.

At tick 0 the environment is created and populated with a customizable number of peers. These nodes are initialized with all the information necessary for the protocol and therefore they initially constitute a stable network. Every 40 ticks a node is randomly selected that it is in charge of a single key lookup. Lookups are performed in an iterative style. In our implementation the system, every 600 ticks, simulates the node join/crash rate; the nodes which crash, they immediately disappear from the network while the nodes that join, they start the joining procedure that requires, since it is based on a lookup, many ticks to be

Action	Starting Tick	Interval
Environment init	0	0
Agents init	0	0
Key lookup	1	40
Node join/crash rate	3	600
stabilize	4	600
fix_fingers	4	600
Message receive	msg creationTick + (1 OR 2)	/

Table 1: Schedule times for protocol actions.

completed. Consistently with chord protocol, each peer periodically performs, every 600 ticks, some predefined functions: stabilize and fix_finger. Basically, with this scheduling mechanism, we ensure that crashes and joins are performed every one stabilization round. Finally, the receive tick of a message depends on the creation tick of that message according to the formula in Table 1. The intervals are defined considering the relations among actions in a real network: a transmission delay of $\sim$ 50ms, a lookup period of $\sim$ 1sec and a stabilization period of $\sim$ 30sec.

2 Node Agent

The core of the protocol is embedded in the behaviour of the node agent. This component is in charge of resolving queries on behalf of the entire system. A query can be expressed as a key lookup request to the system. Nodes interact between them through messages which allow knowledge distribution and therefore protocol progress. The local information available to a node such as *predecessor id*, *finger table* and *successor list* is well defined in [2] and of course such data is embedded in the proposed implementation.

2.1 Random Node Id

The protocol described in [2] relies on *Consistent Hashing*. Basically, the same hash function used to produce node identifiers is used to produce also key identifiers. Consistent hashing solves the problem of having a hash function modulo the number of nodes. Having such hash function would be a problem because it would allow easy assignment of keys to nodes, but when the number of nodes changes, keys migration is required. In the proposed implementation instead of using SHA-1 to produce such identifiers, they are randomly generated. This can be done because our implementation relies on the following assumption: once a node joined the network, that is it knows its successor and predecessor ids and also its successor and predecessor nodes are aware of new node existence, the node keys between the new node and its predecessor are automatically transferred from the new node successor to it. Thanks to this assumption, the problem solved by consistent hashing is not present in our implementation because keys are not stored at all. Hence nodes are only aware of which keys are their responsibility, but they do not store them, which implies that no migration of keys is required. This can be done because once a node is aware of its predecessor and successor, it is in possession of all the information necessary to know that it is responsible for the keys between itself and its predecessor and is able to answer eventual queries for its own keys. Let assume the existence of nodes with ids 40 and 50. When a new node arrives, let assign to it id 45, it will send a JOIN to node 50. When 50 receives JOIN from 45, it will change its predecessor to 45, will inform 45 about its predecessor, which is 40, but also notify 40 about the fact that its successor has changed. After the join, if node 50 receives a query for a key which is now responsibility of node 45, it will search its finger table for the node whose id most immediately precedes the key, which will imply the lookup to traverse all the ring and allow the new node to fully join the network. Therefore with this mechanism, we are able to simulate keys storage without implementing it. Another observation in favour of our choice will be explained in the Analysis section. In that section, we will show how the number of keys stored per node and the probability density function of the number of keys per node behaves like the original article.

2.2 Messages

As previously mentioned node agents interact which other agents through messages. Such messages allows them to perform the actions required for the protocol correctness. Moreover messages allow nodes to transit from *NEW* state to *SUBSCRIBED* state. Nodes which are in the *NEW* state are the ones which have not finished the join procedure. Once this procedure is completed, nodes transit to *SUBSCRIBED* state and become able to store keys and to actively participate in the algorithm. The implemented messages are the following:

- **FIND_SUCC** - Represents the most important message because it allows to start the main procedure of the protocol that is finding the responsible for a key. Once this message is received a node checks if it is the responsible of that key. If it is, it replies with a *FOUND_KEY* message, otherwise it searches its finger table for the node whose id most immediately precedes the key and answers with a *FOUND_SUCC* message. The action implemented by this message is used also by new nodes that try to join in order to find their successor and by subscribed nodes to fix their finger table. Therefore on reception of this message additional actions may be required, hence this message can have three sub types explained later in this sub section.

- **FOUND_SUCC** - This message basically implements the iterative style of the lookup procedure. Each node initially launches *FIND_SUCC* to the node whose id most immediately precedes the key, selecting it from its finger table. The selected node either is the successor of the key or replies with a *FOUND_SUCC*, announcing it knows a node which is closer to the key than him and sending this closer node id to the originator of the lookup. On reception of this reply, the node who initialized the lookup sends another *FIND_SUCC* to such closer node. This procedure is repeated until the correct successor is reached.

- **FOUND_KEY -** This message represents the acknowledge that nodes are searching for. Basically, a lookup ends when on reception of a *FIND_SUCC* message, a node is able to reply with a *FOND_KEY* message because it is the successor of that key. Hence the reception of this message means that the lookup reached the successor of the searched key. Moreover, once this happens, the node interacts with simulation's master node, signaling him the lookup outcome.

- **STABILIZE -** Every 600 Repast running environment ticks, each node generates a *STABILIZE* message which is sent to its successor. With this message, a node asks for successor changes in knowledge. If predecessor of successor has changed, its id is sent back, otherwise the successor list.

- **ACK_STABILIZE -** To accomplish stabilization, the node who receives *STABILIZE*, answers with a *ACK_STABILIZE* message which contains the information mentioned previously.

- **NOTIFY -** During a stabilization round, a node may understand that its successor has changed. If this is the case, it contacts the new successor using this message, telling him to update its predecessor to this node.

- **NOTIFY_SUCC_CHANGE -** When a new node join, once its *FIND_SUCC* request reaches the right successor, the successor of the searched key sends a *NOTIFY_SUCC_CHANGE* message to its own predecessor, in order to inform him about the new node and making him avoid waiting until the next stabilization round to discover the new node.

- **NOTIFY_CRASHED_NODE -** Our implementation allows nodes to detect in a synchronous manner when nodes have crashed (more on this in next subsection). Suppose that, during a lookup, node x has discovered, from node y that node z is close to key k. If x contacts z and discovers that it has crashed, it will immediately sent a *NOTIFY_CRASHED_NODE* message to y, in order to inform it, that z has crashed. Consequently y will mark z as crashed in its own *fix fingers table.*

A key observation for the protocol but also for our implementation is that lookups are used for three reasons: (1) searching the correct successor of a key, (2) searching the correct successor of a new node during a join, (3) fixing entries of the finger table. For this reason, a node must be able to understand which kind of lookup is processing. In order to accomplish this but still having a simple lookup procedure which can be applied transparently in all cases, *FIND_SUCC*, *FOUND_SUCC* and *FOUND_KEY* messages store an additional sub type which helps a node understand in which type of lookup is involved. The sub types we are defining are three and are the following:

- **LOOKUP** - Signals to a node that the lookup does not require any special behaviour because it represents a simple key search.
- **JOIN** - Signals that the lookup is related to a join procedure. If a lookup has this sub type, once the request reaches the correct successor of the searched key, that node has to set its predecessor to the joining node, and signal to the old predecessor that its successor has changed. Moreover, when the last reply reaches the initiator of the request, that node has to set its successor and predecessor.
- **FIX_FINGERS** - This sub type is used only by the initiator of a lookup request. When the *FOUND_KEY* message is received, instead of sending the outcome to the master node, it uses it to correct its finger table.

2.3 Fail Detection

One of the main requirements of the protocol, is ability of nodes to detect failure of other nodes. Basically if a node does not reply after some amount of time, the node is considered to be crashed. When such situation is detected, in function of node state and of lookup sub type, the node has to perform some actions in order to guarantee protocol correctness. In order to accomplish a correct fail check procedure of the protocol, in our implementation each node makes use of a local and dedicated component called *PendingLookupRequests*. Such data structure is based on an extended *HashMap* defined as follows:

$$PendingLookupRequests : ReqId \rightarrow ReqPaths \tag{1}$$

Where *ReqId* is an *Integer* representing an id assigned locally by the node to such request and *ReqPaths* is an *ArrayList<ArrayList<Integer>>* representing a list of all (successful or unsuccessful) paths for that lookup (the first is the current). Obviously, the elements contained by this *HashMap* contains more information, such as the searched key, needed for implementation reasons, but this information is omitted for simplicity in this analysis. This component allows each node to easily keep track of the state of the lookup and understand if a node has failed or not. Nodes use this structure in the following way:

- Generate a locally unique id for each lookup request and add it to *PendingLookupRequests*
- Add the initiator node to request path and any node which is contacted during this lookup try
- For each node contacted schedule fail check in 5 ticks. This is done because a message response-reply lasts at most 4 ticks (2 ticks/message).
- Fail check (after 5 ticks) in the following way:

  – If the current *ReqPath* contains a new node, it means that we were able to move forward the request and hence the node has not failed, otherwise a node has failed and we are able to identify it.
  – If the path encountered a failed node and the request has already been retried for 5 times, stop retrying and signal lookup failure to master node.
  – If the path encountered a failed node and the request has been retried for less than 5 times, create a new path and retry.

3 Router

Messages are transmitted between nodes by simulating a full mesh virtual underlying network. In order to encapsulate the local state of a node, simulate a real communication and avoid passing object references between agents, nodes know only the identifier of other participants. The identifier acts as the node's network address and can be used to route messages. For this reason, the component called Router is implemented which behaves on behalf of the full mesh virtual network. The router is responsible for the following functionalities:

- **Message transmission**: when a node sends a message, the router locates the destination node and schedules the reception of the message.
- **Message propagation delay**: the receive of a message is delayed by a random tick number between one and two, as stated in Section 1.1.

4 Master Node Agent

The system has a single coordinator agent called ChordNode. This agent is in charge of simulating a unified application layer of all nodes. Basically, it communicates with the node agents imposing them to perform actions such as keys lookup, crash and join. Moreover, it initializes the nodes that constitute the initial stable network and it keeps track of all the lookups. When a lookup is completed or failed, the node that starts it signal the outcome to the ChordNode. That way, the supernode is able to get a global vision of the system, useful to compute statistics and to manage some visualization aspects; while a node is performing a key lookup, a line representing the communication between two nodes is displayed.

5 Analysis

The main focus of the analysis was verifying that the communication cost and the state maintained by each node scale logarithmically with the number of nodes. To do this we performed some measurement metrics inspired by the experiments presented in the original paper. The results of our analysis will be presented starting from subsection 5.2 until the end of the section. However, we reserve the next subsection to briefly present how to interact with the simulator and which are the actual steps needed to configure the environment, run the protocol and output results.

5.1 Simulator configuration

Our simulator is based on Repast Symphony, an agent-based modeling toolkit and cross-platform Java-based modeling system [1]. This technology allows to graphically represent a complex system with the usage of efficient visualization features and also plot computed statistics.

5.1.1 Semantics of the Repast Display

Repast allows to graphically represent the system and the visualization is done through the Repast Display. Figure 1 shows the representation of a system during a simulation. As can be seen, nodes are displayed and arranged in a circle depending on their id. Nodes which have completed the joining procedure and that have their successor and predecessor set, that are in the state SUBSCRIBED, are displayed in black. Nodes which are still joining and still do not have collected all the information necessary to participate in the protocol, that are in the state NEW, are displayed in gray. Moreover, the display shows green lines that represents message exchanges during a key lookup. Lookups are performed in an iterative style and once the answer for a message sent for a step of the lookup is received, the line is removed.

5.1.2 Data Collection

To evaluate that in our implementation the communication cost and the state maintained by each node scale logarithmically with the number of nodes, we performed some experiments. Results of the are computed before at the end of a stabilization round. We basically analyze the following 3 probability distributions:

- **Number of Keys per Node** - from which we calculate the mean, 1st and 99th percentiles.
- **Lookup Path Length** - from which we compute mean
- **Number of Timeouts per Lookup** - from which we compute the mean

Such measures are calculated by the master node, which is the only having a global picture of the situation and the printed to the console. If the reader of this report desires to test such computed measures, it has to execute the code in development mode and inspect its IDE's console. An example of the output is the following:

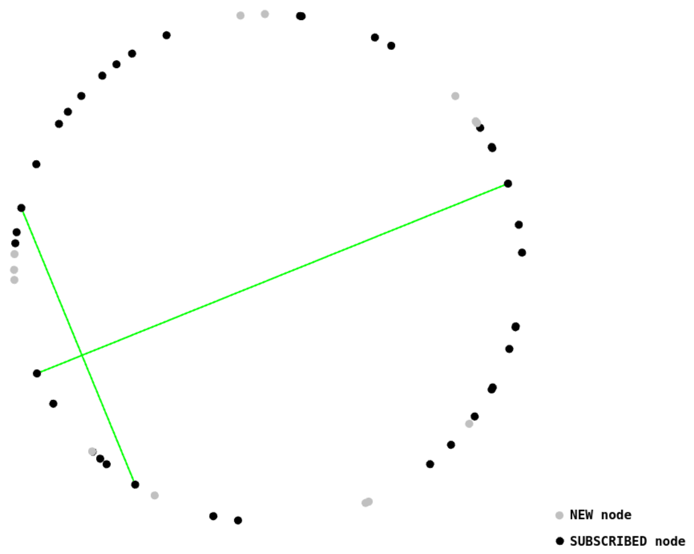

Figure 1: Semantics of the Repast Display

```
########## SIMULATION 1578266879519##########
Nodes: [459282, 1053995, 14729669, 14948457, 16542290, ...]
Space Dimension; Num. of Keys; Mean Num. of Keys; 1st percentile; 99th percentile
2147483647;1000;2147483.647;16272;9780939
Num. of Keys per Node Distribution: [1065;1096;3224;3307;4776;5803;7669;10324;....]
Mean Path Length: 6.7025779917745005
Mean Num. of Timeouts: 0.003912127595546194
```

5.1.3 Configuration Parameters

The simulator exposes two parameters that can be set to recreate specific configurations and influence the algorithm's behavior.

- **Number of nodes** - The number of nodes that are present in the network; this number is kept about constant during the entire simulation. When a number of nodes crashes, the same amount of new nodes is inserted in the network.

- **P single node failure** - The probability that a node crashes. Nodes are randomly select to fail every one stabilization period. Thus, in order to simulate m failure per stabilization period, compute p as follow: $p = m/\#nodes$.

- **Num. Lookups/Stabilization Round** - The number of lookups every stabilization rounr (600 ticks) that are necessary for a experiment can be configured using this parameter.

- **Num. Lookups/40 ticks** For some experiments lookups are needed more often than 600 ticks. For this reason this parameter allow to configure the simulation with a reasonable lookup rate.

5.2 Load balance

We first consider the fairness of our random generator to allocate keys to nodes. In our implementation the node identifiers and the keys are mapped on the same space and, with N nodes and K possible keys, we would like the distribution of keys to nodes to be around N/K. We consider a network consisting of 10^4 nodes and range the possible number of keys from 10^5 to 10^6 by changing the space dimension. A node is responsible for the keys that correspond to the difference between its id and the id of its predecessor. Figure 2 plots the results showing the mean and the 1st and 99th percentiles of the number of keys that a node is responsible for. As can be seen, the number of keys per node exhibits some variations but the results behave like the ones presented in the original article. Moreover, in order to understand better the distribution of the keys we also evaluate the probability density function (PDF) of the number of keys per node that a node is responsible for. Figure 3 shows the result obtained. Also in this case the behaviour is not excellent but it is consistent with the original article.

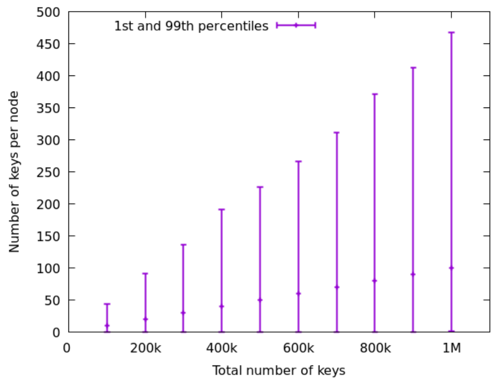

Figure 2: The mean and 1st and 99th percentiles of the number of keys that a node is responsible for in a 10^4 node network.

5.3 Simultaneous Node Failures

As in the original article, we test our implementation robustness when dealing with simultaneous and massive amounts of failures. In a ring of 1000 nodes we make crash a specific fraction of nodes. Such fractions are specified in Table

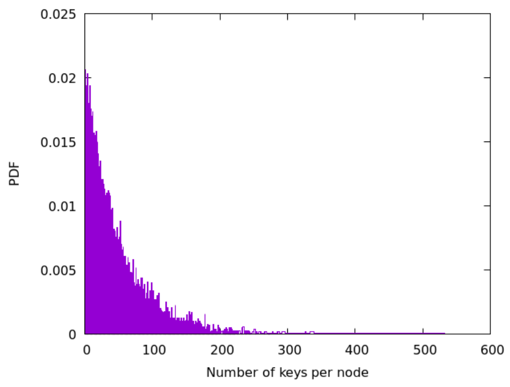

Figure 3: The probability density function (PDF) of the number of keys per node. The total number of keys is $5 * 10^5$

2. Immediately after crashes, 10000 lookup requests are released. After enough time, any request either succeeds or fails. (due to timeout). Finally, we are able to compute the mean path length and the mean number of timeouts of all lookup flows.

Fraction of crashed nodes	Mean path length	Mean num. of timeouts
0.0	6.9	0
0.1	7	0.3
0.2	7.3	0.8
0.3	7.8	1
0.4	7.8	1.3
0.5	8	1.8

Table 2: The path length and the number of timeouts experienced by a lookup as function of the fraction of nodes that fail simultaneously.

As can be seen in Table 2 when the number of crashed nodes increases also the path length and the number of timeouts increases but the mean path length remains logarithmic.

5.4 Lookups During Stabilization

In this experiment, we evaluate our implementation of Chord performing lookups when nodes are continuously joining and crashing. When a node crashes it immediately disappear from the network and it does not communicate any information to other participants. Key lookups, joins and failures are generated according to Table 1 and also the two procedures for stabilization follow the same scheduling. This implies that when join/crash rate is simulated, joins and crashes happen every one stabilization round. Table 3 shows the mean path length and the number of timeouts experienced by 1000 lookups.

# Nodes	Node join/crash rate (prob./per stab. period)	Mean path length	Mean num. (of timeouts)
125	0/0	4.824	0
125	0.008/1	6.582	0.244
250	0/0	5.707	0
250	0.004/1	6.812	0.209
500	0/0	7.621	0
500	0.004/2	7.677	0.317
750	0/0	8.175	0
750	0.0039/3	7.846	0.236
1000	0/0	8.943	0
1000	0.002/2	8.324	0.17
1000	0.004/4	7.963	0.381
1000	0.006/6	8.799	0.506
1000	0.008/8	8.96	0.629
1000	0.01/10	9.701	0.806

Table 3: The path length and the number of timeouts experienced by 1000 lookups as function of node join and crash rates.

As can be seen, we tested many networks with different number of nodes and different probability of node failure. In all the experiments that were performed, can be noticed that, the path length scale logarithmically with the number of nodes and that the number of timeouts is relatively small. This suggests that our implementation behaves as expected.

6 Conclusions

In this report we describe our implementation of chord protocol as described in [2] using Repast Simphony [1] and Java. The main modules of the system are defined and described along with their contribution to the simulation. The main procedures of the protocol are embedded in agents of type *Node*, which move the global state forward while interacting with other nodes of the chord ring. We defined a message-based oriented protocol in order to implement chord specifications. Such protocol is well described in section 2. We explained why in our simulation environment, consistent hashing, does not bring any improvement because keys do not exist at all. In sub section 2.3 we explain our architectural mechanisms for failure detection. Moreover, we tested the system and focused on checking if the computation time was $log(\#nodes)/2$ like in the original paper. As well described in section 5, we found that our system behaves logarithmically as expected.

References

[1] Michael North, Nicholson Collier, J. Ozik, Eric Tatara, C. Macal, M. Bragen, and Pamela Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1, 10 2013.

[2] Ion Stoica, Robert Tappan Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEE/ACM Transactions on Computer Systems*, pages 17–32, 2003.