Implementing Distributed Hash Table (CHORD)

Guillaume Ardaud <gardaud@cct.lsu.edu> - CSC 7103

This report presents the results obtained with my personal implementation of the CHORD algorithm with the Python language. The resulting implementation supports all the basic features of CHORD, including the stabilization protocol; we will see in the report how well it performs.

Overview of the code

The code is written in Python, using standard libraries, and should therefore run on any machine with a 2.x installation of the Python interpreter.

The code is split in 4 files:

chord-main.py is the main file, the one that should be launched from Python. All it does is create a Network object, and goes in an infinite loop calling the tick() method from the Network object every loop.

Network.py is a class that simulates the network onto which the nodes are connected. It stores an array containing references to all the nodes, as well as general methods for the CHORD implementation (for example for message passing, or getting a reference to a node with a given ID). It also has a tick() method, which is executed repeatedly; the calls issuing key lookups and such should be placed in this method.

Node.py is the class that implements a single node. It contains an array of references to other nodes, which constitutes the finger table, as well as methods to retrieve a key, add a key to its own collection, process messages, etc. Note that in this implementation, we do not use SHA1 to hash node IDs and key IDs- we juste use unique integers to identify them. In the end, it is the same; hashing IDs using SHA1 is only necessary in a context of true P2P, where we have no control over the proper names of the nodes or keys shared on the network.

Message.py is a class that represents a message that is exchanged between two nodes. It stores the type of the message (either a key request or a response), information on the node that originally sent the message, the key that is requested, the time at which the message was first issued, the number of times the message was relayed (hops) and finally, an empty ‘owner’ field that is filled by the node possessing the key when it issues the message as a response.

Part A

In this part, we implemented the basics of the CHORD protocol to allow any number of nodes to store any amount of data (in all the tests, the setup involved 1000 nodes and 50 000 keys).
The network then issued 10 consecutive requests from a random node looking for a random key (using the method randomRequest from the Network class), and measured the average time it took to retrieve the key (the full log can be found in the file “partA-table1.log”).

#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Avg.
61	51	61	42	61	31	21	41	51	41	**46.1**

fig 1: Time taken for 10 requests (milliseconds)

Then, for the second portion of part A, the protocol was extended to support simultaneous requests. Figure 2 presents the results obtained for a varying quantity of requests (the corresponding logs are named “partA-Xsimultaneous.log”).

2 simultaneous requests	5 simultaneous requests	10 simultaneous requests
51	38	57
41	50	47
	50	48
	70	38
	40	48
		28
		48
		58
		58
		69
46	44	52

fig 2: Time taken for various number of simultaneous requests (milliseconds; average is in green)

Comparing with the results first obtained, we observe that there is little difference in the results- this is due to the distributed way in which the nodes process the messages.

Part B

In part B, we implemented the CHORD stabilization protocol and introduced a random a number of node failures in the network. This is interesting to observe, because in real life situations a client can potentially disconnect at anytime; the protocol has to handle this efficiently.

The table below shows the result for 5 random queries for a different number of failures (the full log files are named “partB-Xfailure.log”).

1 failure	2 failures	5 failures	10 failures
2.67	2.70	2.64	2.64
2.68	2.71	2.65	2.64
2.66	2.67	2.65	2.64
8	2.70	5.35	2.64
timeout	2.67	5.34	timeout

fig 3: Result for queries after a number of node failures were introduced

We observe that there is a very little failures overall; furthermore, inspecting the logs shows that obtaining a key is done in average roughly in just as many hops. The only difference we can observe is in the total time taken; however in this context, this metric is irrelevant because it is hugely affected by the fact that our simulator has to compute the finger table for each node (1000 in our simulation) each time the stabilization protocol is ran. In a real life context, each node would be a separate computer, and the computational load would be much lighter, thus making this issue completely disappear.

These observations show how well CHORD adapts itself to situation with a heavy percentage of node failures, and how the structure of the lookup table allows to maintain a very efficient network integrity regardless of the failures, making CHORD a flexible and powerful protocol.

Part C

Finally, in part C, we run key searches using a naive approach; that is, the node looking for a key asks its successor for it; if it has it, it returns it otherwise it asks its own successor and so on.

The corresponding method is located in the Network.py file, and is called naiveSearch().

Figure 3 shows the results obtained for 10 requests (see the detailed results in the file "partC-10naiverequests.log").

#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Avg.
9.66	1.49	2.18	9.8	4.03	5.36	9.09	5.79	6.4	4.27	**5.807**

fig 4: Time taken for 10 requests with the naive approach (seconds)

It is quite easy to see where the average time of roughly 5 seconds comes from; since our network has 1000 nodes, if we take two random nodes, there will on average be 500 nodes between them. If we consider the computation time to be negligible and only consider the network delay, then the total time to retrieve a key will be 500*network delay= 500*0.01 seconds in our implementation = 5 seconds on average.

If we compare this average of 5 seconds to the average of 50 miliseconds using the finger table. we can see the benefits of the finger table very clearly, as the finger table performs roughly 100 times better in average.