Chord

Distributed Systems 2 – 2nd assignment

Nicolò Pomini, 203319
nicolo.pomini@studenti.unitn.it

December 21, 2019

1 Introduction

This report describes the implementation and the evaluation of the second assignment of the *Distributed Systems 2* course, which is based on a publication written by Stoica, Ion, et al. that describes *Chord*, a scalable distributed hash table protocol [1]. The assignment is developed using Repast Simphony[1], an open source agent-based modeling and simulation platform for Java and Groovy languages.

I have faced several issues with Repast Simphony, and they are explained in Section 4.

The report has the following structure: Section 2 gives an overview of Chord, Section 3 states the assumptions and the architectural choices, as said Section 4 illustrates the issues with Repast, and finally Section 5 describes the experiments and the results of the evaluation of the protocol.

2 Protocol overview

Chord is a Peer-to-Peer distributed hash table protocol, that is able to scale with respect to the number of total nodes in terms of messages exchanged and number of items handled by each member. It is fully decentralized, meaning that nodes discover themselves changes in the system, and that they retrieve information they do not own autonomously.

Nodes are arranged in an overlay network that is circular, called the *chord ring*, and each of them has a successor and a predecessor. The node and the key spaces are thus circular – usually expressed as a power of 2 – and every node is responsible for a set of keys, that goes from the node's predecessor (excluded) until itself (included):

$$keys(n) = (n_{pred}, n]$$

In case a node has to retrieve an item that it does not own, it asks to other nodes who is the owner of such an item. This process is handled in a scalable

[1] https://repast.github.io/

way, using a number of messages that is bounder by $O(\log n)$, supported by data structure called *finger table*, that allows a node to achieve such an upper bound. Inside the finger table there are references to nodes that are exponentially far by the node itself in the circular space.

Of course, Chord is robust to nodes that fail or join the network during the execution of the protocol.

3 Assumptions and architecture

In the simulator implemented for this assignment I assumed that every node is identified by an integer number – called *Id* – and that the number of available keys is a power of 2. In this way no hash functions and no IP addresses must be used to identify a node, and also having the $totalKeys = 2^M$ allows to have the size of the finger tables equal to M. The total number of nodes, instead, can be any integer number $n \,|\, n <= 2^M$. In all the examples and the simulations performed, n is significantly smaller that the total number of keys.

The simulator has several parameters to set, and it can be run in two operating modes. They are briefly introduced and explained in Section 3.1.

The simulator is developed in three classes: the most important is the `Node` class, where the behaviour of each node is implemented; the other two classes – `Builder` and `NodeManager` – are used to create and manage the simulations. Section 3.2 and Section 3.3 give details about said classes.

3.1 Simulator parameters

The parameter `M` is the exponent of 2 that defines the space of the keys. As said, $totalKeys = 2^M$. The parameter `NODES` is used to set the number of nodes in the network. Each node keeps a list of successors, in order to support node failures and departures, and the size of such list is set with the parameter `SUCCESSOR_LENGTH`. Then, the running modality is specified: it can be either *standard* or *disaster*.

The standard execution is used to simulate the protocol under normal conditions. In this case, another parameter is needed, called `LEAVE/JOIN`, that tells the simulator how many nodes leave and join the system per round. In this way, a number equal to `LEAVE/JOIN` nodes is randomly chosen to leave the system, while the same number of new nodes enters the system with a random position and knowing a random node in the system itself. The new nodes will use the known node to discover their position in the ring, their successor and predecessor and their keys.

The disaster execution is used to simulate extreme events, such as a massive number of nodes that crash simultaneously. This modality needs a parameter called `FAIL_PROB` that indicates the fraction of nodes that crash at the same time. It is a real number between 0 and 1. For example, a value of 0.5 would cause half of the total nodes to crash simultaneously.

3.2 Node

The node class is the core of the project. Every node is identified by an integer *Id*. It has two lists of nodes, the finger table and the successor list, plus some other lists used to evaluate the protocol, such as a list of integers containing the numbers of keys over time. Each node also has a counter called *failures* to track how many requests failed during a *disaster execution*, and a counter called *timeouts* to count how many unreachable nodes were met during the *disaster mode*.

The lifetime of a node is the following, as described by Stoica, Ion, et al. in their publication [1]: at every iteration it controls that its finger table is up to date, checking whether the contained nodes are still available. This operation is called *fixFinger*. Also, all the nodes check periodically whether their successor has changed, using a function called *stabilize*, and also they check if their predecessor is still alive, with the function *checkPredecessor*. Finally, nodes performs lookups, searching for keys and potentially sending messages to other nodes to satisfy their searches. Every nodes extracts a random number in the key space and starts looking for the key identified by the latter number.

Searching for a key means finding the node that is responsible of such key. The function *findSuccessor* is used in this case. The result of *findSuccessor* is modified from the original version in [1], returning a pair $(node, mess)$, containing the searched node and the number of messages needed to respond. The latter information is used in Section 5 to evaluate the protocol.

In case all the successors of a node are failed, the protocol stops since the latter node has remained isolated. In case the *disaster execution* mode is simulated, it is possible that the simulation is aborted for this reason.

Finally, the function *join*, used by a new node to join the chord ring, is slightly modified with respect to the version showed in [1], making it return a boolean value: it returns true in case everything goes fine; otherwise, in case the node contacted is not able to provide the new node's successor, *join* returns false, indicating that the procedure to insert a new node in the system has failed.

3.3 Builder and Node Manager

The other two classes, `Builder` and `NodeManager` are used to initialize and manage the simulations. The protocol is simulated at steady state, so the builder prepares the *chord ring* with all the nodes pointing to their successors and predecessors, and with all the finger tables already filled.

The node manager deals with removing and adding nodes to the system. In case of the standard execution, at every iteration a given number of nodes is randomly chosen and removed from the network, and as many as the latter are added in random positions. In case of a disaster execution, the manager removes a portion of the nodes – as specified by the parameter `FAIL_PROB` – at the very beginning of the simulation, in order to see how the protocol behaves immediately after such an extreme event.

4 Problems with Repast Simphony

Unfortunately, I have faced several problems with Repast Simphony, that have remained unsolved.

The first one – and the less serious one – is about the display of nodes arranged in a ring and connected with arrows. The arrows should have pointed from a node to its successor, but they have not been included into the project because they caused the program to crash suddenly: after a random number of ticks, a runtime NullPointerException was thrown, unrelated to any component of my project. I asked on StackOverflow[2] and I was answered by one of the developers of Repast, but as today (december the 21st) I still have no answer on how to solve the problem.

Some more serious problems happens when I tried to simulate the protocol with a very large number of nodes – like 1 million or more. In these cases, Repast was not able to load in memory all the nodes, causing either a NullPointerException when a node was trying to contact a node not loaded in memory, or a stack overflow caused by an endless recursion of the method *findSuccessor*, in case a node was not able to find any other node nearer to the searched key, and consequently getting itself as nearest node, and calling continuously *findSuccessor* on itself. For this reason all the simulations made in Section 5 involve a smaller number of nodes with respect to the simulations made in the original paper [1].

5 Evaluation

The evaluation is performed in a similar way of Stoica, Ion, et al. in [1]. Basically, the two main metrics are the load balance and the path length, meaning the number of nodes managed by each node and the number of nodes visited to resolve a lookup operation respectively. At each round of the simulation, 3 nodes are randomly chosen and removed from the protocol – simulating a volunteer leave – and 3 other new nodes are created, added in a random position and put in communication with a random node belonging to the system. In this way the system is never stable, and it has to run the stabilization protocol at every round to fix node's successors.

In addition to that, some extreme situations are simulated, making a huge percentage of the node fail at the same time, using the simulator in *disaster execution* mode.

Let us see the details of the evaluation experiments.

5.1 Load balance

To evaluate the load balance of chord, several simulation were run, with the following setup: the number of nodes is equal to 1000, the total number of keys is

[2] https://stackoverflow.com/questions/59161210/repast-simphony-runtime-nullpointerexception-caused-by-the-network-projection

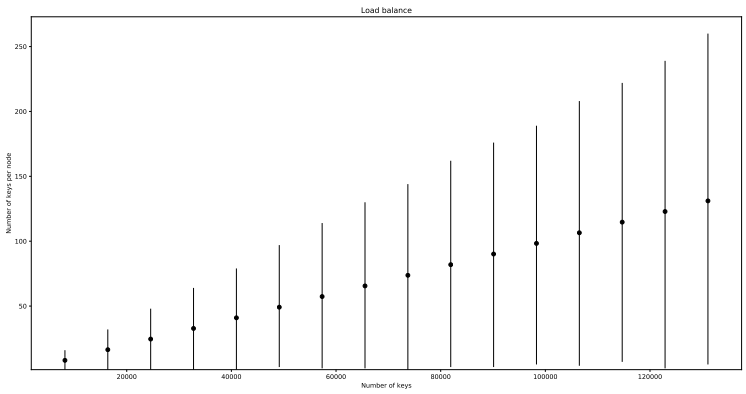

Figure 1: The number of keys handled by each node. Each vertical bar represents an experiment: the lowest point is the minimum value recorded, the highest is the maximum, and the dots are the average.

2^x – with x spacing from 13 to 17, from 8192 keys to 131072 keys – incrementing at each simulation the number of keys by 2^{13}; finally, the simulator is always run in the *standard* way, with 3 nodes leaving and other 3 joining per round.

To measure the number of keys managed by each node, it is recorded the distance between the node itself and its predecessor, every time the latter changes. Since nodes leave and join continuously during the simulation, the method *notify* always changes some predecessor in some node, every round. Once a new predecessor is set, the difference between the *Id* of the current node with the *Id* of the new predecessor is computed (in the clock space of the chord ring) and added to a list called *numberOfKeys*, available in each of the peers.

For each experiment, the minimum of all the minimum number of handled keys is plotted, together with the maximum of all the maximum and the average of all the averages. Such values are bounded, with a maximum value around 250 in the worst case. Due to the random positions of the nodes, some of them are more distant to their predecessors with respect to the average distance between nodes, and so they handle a larger number of keys. Since at every iteration of the simulation new nodes join the system, it happens that some of them joining at the very last round have no time to stabilize, and so their number of handled keys is 0.

Figure 1 shows how the average number of keys handled by each node scales as the ratio between the number of keys and the number of nodes, when the number of keys increases linearly.

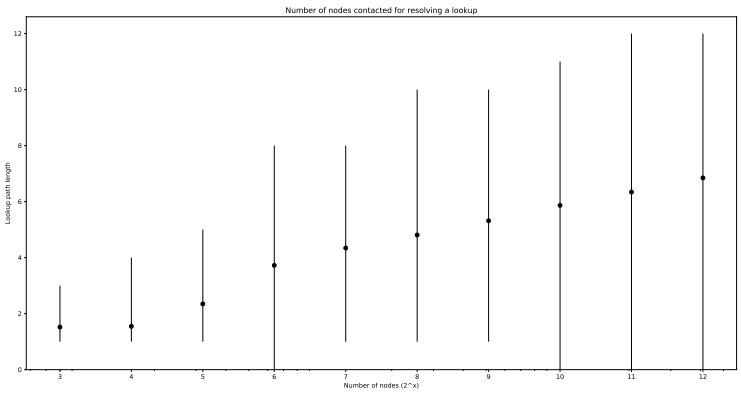

Figure 2: The number of nodes contacted to resolve a search. Each vertical bar represents an experiment: the lowest point is the minimum value recorded, the highest is the maximum, and the dots are the average.

5.2 Path length

The path length measures how many nodes are contacted to resolve a key search. Since the number of keys stored by each node is limited and it is much smaller than the total number of keys, it is likely that nodes must contact other nodes in order to satisfy a search.

We measured the number of nodes contacted in each query, that is bounded by $O(\log n)$ thanks to the finger tables. The experiments involved a number of nodes equal to 2^x, a total number of keys is 2^{x+7} – in order to have a number of keys that is around 100 times the number of nodes – and the simulator was run in the *standard* way. The value of x spaces from 3 to 12.

The way the path length was measured is the following: the function *findSuccessor* is modified, making it return a tuple $(node, |mess|)$ – where $node$ is a pointer to the successor, and $|mess|$ is an integer number counting how many nodes had been contacted to obtain the result. All the $|mess|$ values are stored by each nodes in a list called *pathLengths*, and the data plotted in Figure 2 are – for each experiment – the minimum of all the minimal *pathLengths* recorded, the maximum of the all maximal *pathLengths*, the average of all the average *pathLengths* recorded by each node.

As can be seen in Figure 2, the number of nodes contacted to satisfy a search scales logarithmically with the number of nodes – in fact the scale on the X axis if logarithmically, and both the average number of nodes contacted and the 99th percentile grows linearly – or they stay constant – in the plot, and so overall the path length is bounded by $\log n$. Also, in the x-axis is reported the exponent used to compute the number of nodes, and as can be seen the average values on the y-axis are always lower than the x ones. Also, the majority of maximums

are also bounded by $\log n$.

5.3 Simultaneous node failures

In these experiments a significant portion of nodes is being made fail simultaneously, to simulate some extreme phenomena of bad luck. The settings of the simulator are the following: there are 1000 nodes, 2^{14} keys, and a percentage $p\%$ of nodes fails at the beginning of the simulations. Then the protocol is run for 20 rounds, to see how it behaves in these extreme situations. The value of p goes from 10% to 50%, increasing it by 10% for each experiment.

Three metrics are recorded:

1. The average path length, in the same way it is done in Section 5.2.
2. The min, max and average number of timeouts. A timeout occurs when a node tries to contact its successor (or a member of its successor list), but the latter has failed. Each nodes counts how may timeouts it faces, and then the results are aggregated.
3. The number of lookups that fail. A failed lookup is a lookup where the function *findSuccessor* is not able to return a node. This happens when *closest_preceding_node* returns the same node that is executing the function, and this behaviour would cause an infinite recursion in *findSuccessor*, since a node n would call *closest_preceding_node*, obtaining itself as result and calling again *findSuccessor* on itself, forever.

The results are expressed in the following table:

Failed nodes (%)	Avg path length	Total failures	Avg timeouts (min and max)
10	5.78	112	0.11 (0, 2)
20	5.67	623	0.32 (0, 1)
30	5.57	674	0.45 (0, 4)
40	5.46	700	0.67 (0, 7)
50	5.32	770	0.8 (0, 8)

As can be seen, the number of timeouts is really limited: this means that the protocol is able to restore the pointers between nodes, and only the rounds immediately following the huge number of fails face timeout problems.

The number of failures, instead, is much higher, but if we think that the total number of lookups is #survived nodes $*$ #rounds, the number of failures is still limited. For example, the lookup performed in the first case is equal to $900 * 20 = 18000$, and only 112 of them fails, while on the lst case we have $500 * 20 = 10000$ lookups, with 770 of them that fails. It is clear that the protocol is very robust with a limited amount of nodes failing simultaneously – like the 10% of them – but it is still quite robust even with huger accidents.

The average path length is in line with the experiment in Section 2 with the same amount of nodes, and does not suffer from a huge percentage of crashes all of the sudden.

6 Conclusions

In this report is shown how the implementation of Chord allows the system to scale with respect to the number of nodes. In fact, the number of messages required to search for a key is bounded by $O(\log n)$, while the number of keys handled by each node grows as K/N.

References

[1] Ion Stoica, Robert 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. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.