Simulating Chord in Repast Simphony

Enrico Marchi - enrico.marchi-2@studenti.unitn.it (211464)
Giacomo Vitali - giacomo.vitali@studenti.unitn.it (266445)

https://github.com/antbucc/DS2-2019-2010/tree/Second-Assignment/Marchi-Vitali

Introduction

This project consists of the implementation of a version of *Chord* [1], a scalable, peer-to-peer distributed lookup protocol for distributed hash tables.
The protocol allows every node of a distributed system to find the location of a key in O(logN) steps, where N is the number of nodes in the system.
The system is completely dynamic in terms of nodes joining or leaving, this mean that the nodes exchanges messages to maintain updated information about the system.
The goal of this work is to show how much the protocol scales in terms of size of the system (we simulated up to 10000 concurrent nodes) and how it is able to handle failures (node crashes).
A recursive version of the protocol was implemented and simulated with the Repast Simphony framework, then, for the performance evaluation, Python scripts were used to analyze the simulation logs.

Implementation

The system is modelled as an agent-based system, where the main agents are instances of `ChordNode`, identified by a unique address. The rest of the entities in the system act as a support for the simulation (e.g. the channel, the traffic generator, the node manager).
The behaviour of the nodes is specified by the `step()` method, scheduled to be executed one time for each step/tick of the simulation, and by the `receiveMessage()` method, which is called asynchronously (at the beginning of each tick) when a node receives a `Message` through the `Channel` object.

At each step, a node executes the following sub-steps:

- It checks if it has already joined the network, if not it starts the procedure to join (`join()` method) by asking a random node for its successor node in the key space.

If it has already joined:

- With a periodicity defined by the parameter `stabilizePeriod`, it executes the stabilize protocol with its successor node.
- With a periodicity defined by the parameter `checkPredecessorPeriod`, it checks if its predecessor is still active (has not crashed).
- With a periodicity defined by the parameter `fixFingerPeriod`, it updates its finger table, calling the method `fixFingers()`. However, if the finger table is not yet completely initialized, the period parameter is ignored and exactly one entry of the table is updated at each step.

- Finally, it updates the timers of the timeouts and executes the actions related to the expired timeouts.

The nodes can exchange the following messages (subclasses of `Message`):

- `SuccessorRequest`: the source of this message asks for the successor of a specified key, the request will be then forwarded around the network until the successor is found.
  The origin of a series of `SuccessorRequest` messages can be a node joining the network, a generic lookup for a key or the procedure for updating the finger table of a network. All the forwarded messages in a series are identified by a `requestId` field.
- `SuccessorResponse`: represents the response of a `SuccessorRequest` containing the address of the node responsible for the requested key (the successor of the key).
  The response may be incorrect if the network has changed recently (a node has joined or has crashed) and the predecessor node of the real successor of the key has not updated its state to reflect the change. Moreover, a request may fail if the corresponding timeout expires before the response is received.
  The response has the same `requestId` of the corresponding `SuccessorRequest`, this allows the receiver to correctly handle the reception of the response and disambiguate the case in which multiple concurrent requests were sent out.
- `Ack`: when a `SuccessorRequest` is received, an ack is sent back to the sender of that message to acknowledge that the receiver is still alive.
- `StabilizeRequest`: this message is periodically sent by each node to its successor as part of the stabilize protocol.
- `StabilizeResponse`: represents the response of a `StabilizeRequest` and contains the address of predecessor of node that responds.
- `PingRequest`: this message is periodically sent by each node to its predecessor to check it is still active.
- `PingResponse`: in response to a `PingRequest`, it signals that the node is still alive.

The `chord` package contains the core elements of the system:

- `ChordNode`: It is the concrete node implementing Chord. Its only public interface consists in the `lookup()` method, which initiates a request to find the node responsible for a specified key.
- `Builder`: It builds the Repast model for the simulation
- `NodeManager`: It handles the creation and removal (crash) of the nodes in the system, controlled by the parameters `maxNodes` and `churnRate`. It assigns to each node a unique address (integer) used to compute the key of the node by hashing.
  At the beginning of the simulation, the system is initialized by creating two nodes for the bootstrap.
- `TrafficGenerator`: It generates lookup request for random keys, by calling the `lookup()` method of an arbitrary node.

The visualization of a simulation shows the nodes in the full key space, as blue dots in the Chord ring topology.
The first node created is selected and the nodes in its finger table are highlighted in yellow, while the lookup requests initiated at the selected node are shown with arrows: the light blue ones represent the `SuccessorRequest` generated from the selected node and forwarded around the network, a green one represents a correct `SuccessorResponse`, while a red arrow represents an incorrect one.

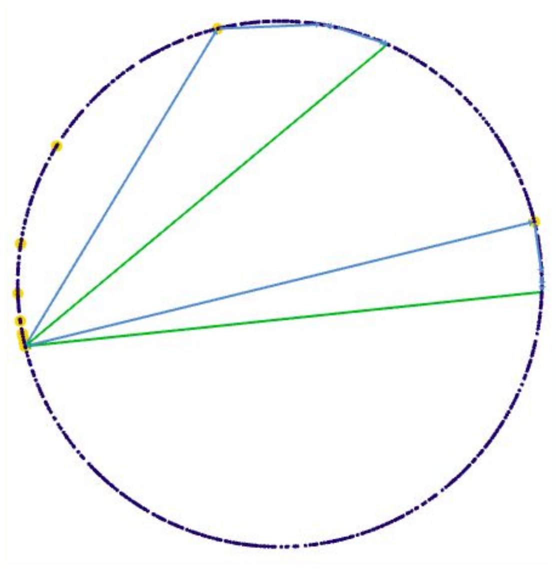
Fig. 1: Snapshot of a simulation
Two successful lookups are visualized

Analysis

The base setting for the simulations is:

- `maxNodes` = 1000
- `churnRate` = 5
- `checkPredecessorPeriod` = 6
- `fixFingerPeriod` = 10
- `stabilizePeriod` = 25

The log of a simulation records the following events *LookupStart*, *LookupOK*, *LookupERROR*, *LookupFAILED*. More in detail, when a SuccessorResponse message is received by the initiator of the lookup request, the logger, based on a global vision of which nodes are currently active, can determine whether the address contained in the response is correct (*LookupOK*) or not (*LookupERROR*).

The goal of the first set of simulations is to analyze how the system scales with increasing numbers of nodes in the network. In particular the simulations are run varying the number of `MaxNodes` (from 100 to 10000) and logging the number of hops taken by the system to complete each lookup.
The graph in *Fig. 2* shows on a logarithmic scale the average number of hops (blu line), the 1-percentile (red line) and the 99-percentile (yellow line). Moreover, the function $\frac{1}{2}$ Log(N) is plotted, representing the theoretical expected value from the original paper [1].
Also meaningful is the fact that the value of the 99-percentile is close to the Log(N), which supports the thesis of this work.

Fig. 3 shows the distribution of the number of hops in a single simulation using the base setting with 1000 nodes. The distribution is approximated by a normal with mean 4.94 and variance 2.40.

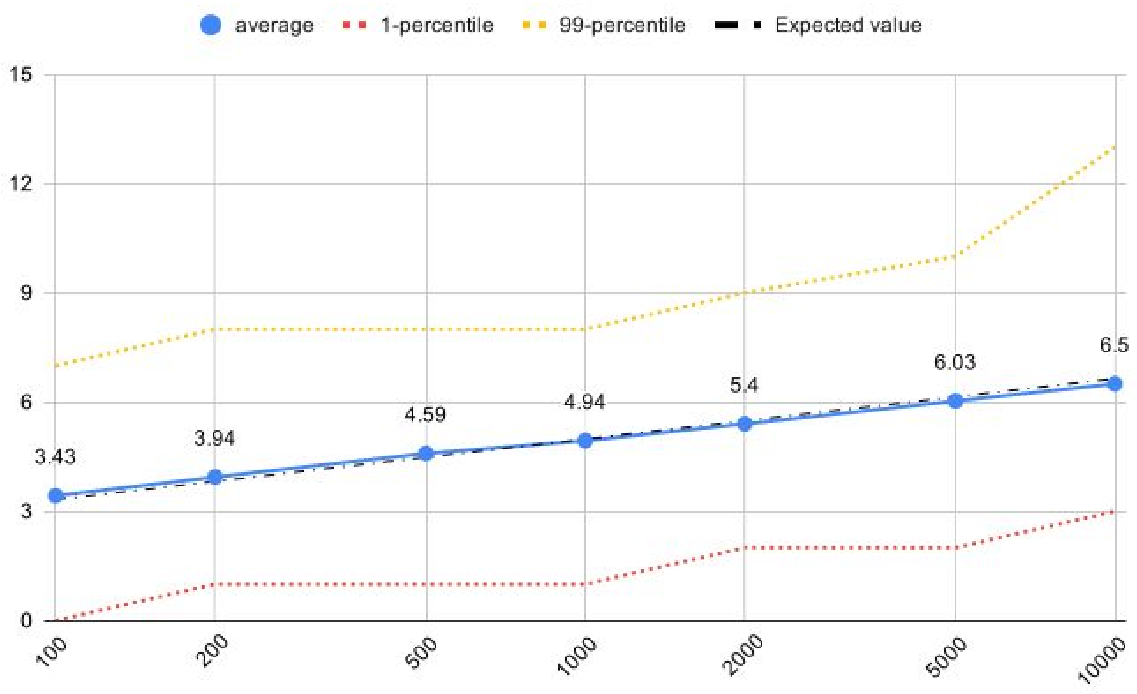

Fig. 2: Average number of hops with different network sizes

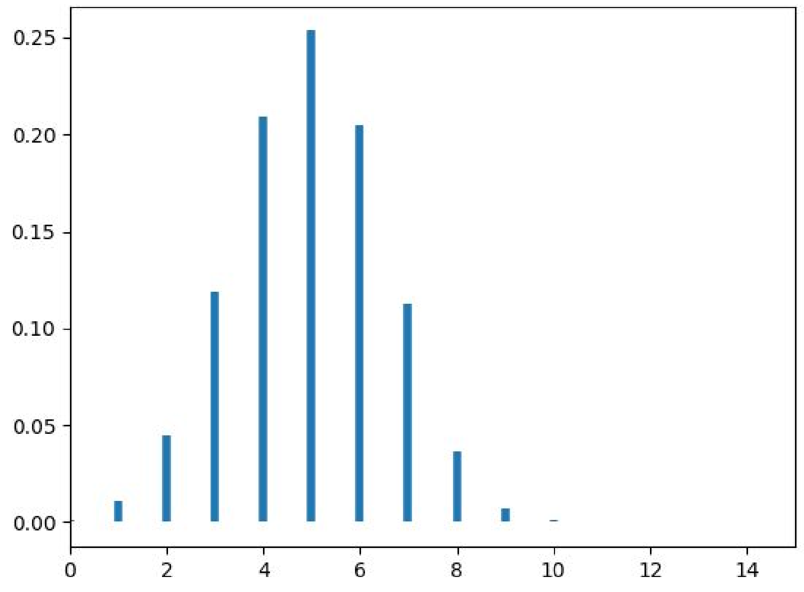

Fig. 3: Distribution of the number of hops with a network of 1000 nodes

For the next set of simulations, the number of nodes is kept constant (1000 nodes) and the parameters `churnRate`, `stabilizePeriod`, `checkPredecessorPeriod` and `fixFingerPeriod` are varied. In particular the `churnRate` ranges between {5, 10, 20} while the other parameters are modified jointly by multiplying their base values for 0.5, 1.0, 2.0.

The goal of these simulations is to extrapolate the fraction of correct lookups, of wrong ones and the failed ones (expired timeouts for the requests).

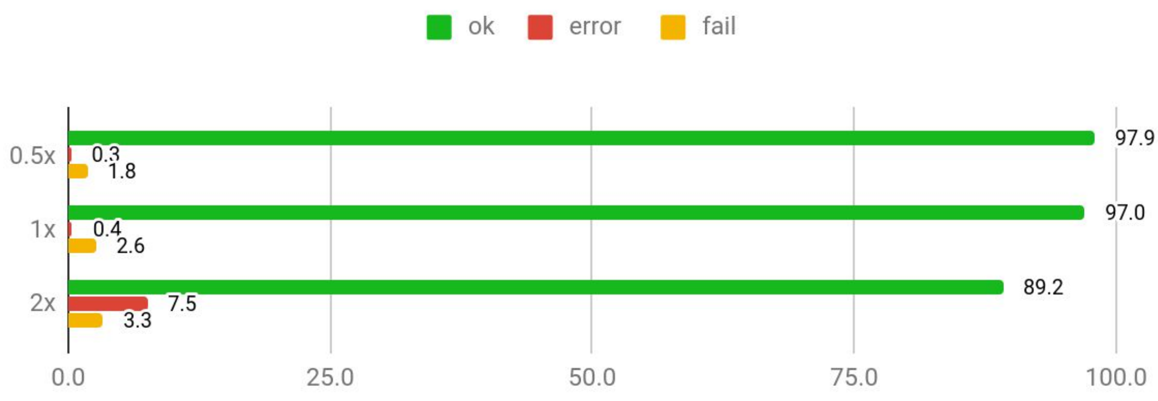

Fig. 4: Lookups results with churn rate 5

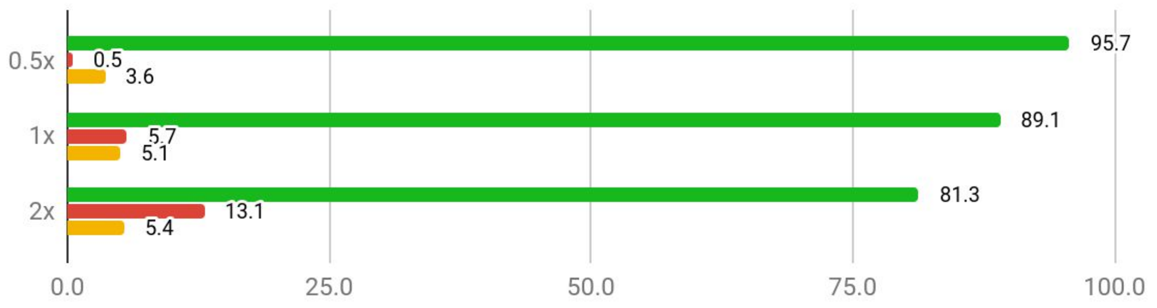

Fig. 5: Lookups with churn rate 10

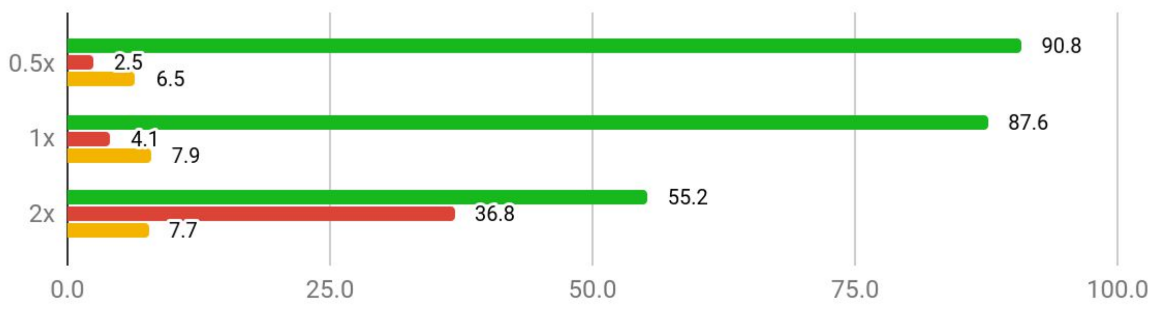

Fig. 6: Lookups with churn rate 20

Fig. 4, *5* and *6* show that with increasing churn rates there is a quick deterioration of the performance, that can be countered by increasing the frequency of the messages used to maintain the topology (stabilize, fixFinger, checkPredecessor).

Conclusions

The results of the simulations confirm both the theoretical expected values and the empirical results obtained in the original paper in terms of scalability and robustness, indeed the fraction of correct lookups in general stays high, beyond 95%, while the rate of wrong responses is below the 1%; the remaining is the fraction of failed requests which can be handled by resending the request after the timeout.

To tolerate a greater number of crashes it's necessary to increase the frequency of messages to fix the network quickly, with the trade-off that the usage of the network bandwidth increases.

References

[1] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. 'Chord: a scalable peer-to-peer lookup protocol for internet applications' *IEEE/ACM Trans. Netw.* 11, 1 (February 2003), 17-32