Chord

Zanetti Marco

Pirocca Simone

Chord is a peer to peer algorithm based on distributed hash tables, it is completely decentralized with the feature to be particular efficient in operation like insertion and removal of nodes of the network and it has a logarithmic temporal complexity respect to the number of nodes.

The implementation and simulation of the Chord algorithm has been done using the Repast Symphony framework.
Repast Symphony is an agent-based modelling and simulation platform for workstation and small cluster, it is mainly used for biological modelling and simulation but its application is possible also on distributed systems protocols.

1. Model Implementation

The algorithm has been implemented with the use of some Java classes, two of them are the most important Node and SuperAgent.

The Node class represents the structure of a node of the algorithm, one instance of this class is created for each node of the network.

The SuperAgent class has been created mainly for the simulation phase, in fact it operates as an "over the other" agent which is used to introduce chaotic events, in order to simulate the failure of some nodes in the network and understand how it works in stress conditions. It is also used to periodically add keys (data) to the network and ask them later to allow analysis of key distribution and steps required to retrieve them.

Some utility classes are present in the source code of the model and mainly provide common operation used to manipulate hashes or provide configuration structures.

Other important classes are the HashHandler one and the NetworkConfiguration one: the former provides static methods to perform hash functions and return an m-bit identifier, the latter is the software abstraction of some parameters of the network. Given that this two classes are just containers or perform simple operations, their description will not go in detail, we will instead focus on the description of Node and SuperAgent classes and on the explanation of the BroadcastContextBuilder that is the object which takes in charge the duty of instantiating the simulation.

2. Superagent.java

The SuperAgent is the object delegated to manage the “chaotic” behaviour of a network, simulating some unpredictable failure due to nodes crash. When a node crashes it is not able to process and manage keys, meaning that the robustness of the whole newtork is reduced.

As an supernatural agent in the model the SuperAgent knows any node in the simulation either it is active or crashed, in the source code this condition is implemented with the use of a collection of all nodes instantiated. In a real case scenario the whole class won’t have any utility but here it is absolutely needed to run the simulation.

The SuperAgent takes some configuration parameters from the ContextBuilder object and complete the setup of nodes, giving them an identifier which is just an m-bit number derived from the SHA1 hash of node’s IP, the reduction is performed taking the mod operation with 2^m. Always because of the simulation needs the SuperAgent gives a random node, joined in the network, when asked; this is to allow newly nodes to join the network.

The SuperAgent also runs some threads: the “ChaosThread”, a secondary thread which at specific time interval pick a random number of nodes and change their “crashed” field, changing it from FALSE to TRUE and vice versa to simulate a random failure in a node and it possible recover after some times, Two remaining threads are dedicated to key insertion and retrieving, once the network in up the first one start to create keys and to add them to a random node, the second one ask the network for a given key and measures the path length to it. Technically all these processes are started by the ContextBuilder once the environment has been instantiated.

3. Node.java

The Node class is the core of the whole algorithm implementation because here are managed all the operation to join or leave the network and to handle keys.

Many private methods in this class are used to perform state changing ad to manage sub/unsub procedures, maintain the finger table and others propagate and split keys to other nodes in the network.
Some methods have a recursive implementation because many tasks require a search action to retrieve keys or find a node responsible for a specific id.

It is in this class that the “chaotic” influence of the SuperAgent through a simple control on the CRASHED boolean variable which, if TRUE, prevents the execution of normal function such as notify, fixFingers, stabilize, join and disjoin; in this way we have been able to simulate at any time an

error in the network without adopting a precise network topology and thus generalizing the simulation as much as possible.

The behaviour of the Node is dependent on Repast's rounds using the @ScheduledMethod attribute on the "main" method **checkClock()** which checks if we are at the start of a new round and runs some functions, otherwise it changes the state of the node. All private methods and functions are called from this scheduled method.

Some methods such as setId are just used at startup, in order to initialize the system in a consistent state and give it means to enter and operate in the network, they are usually invoked by the SuperAgent.

4. BroadcastContextBuilder.java

The BroadcastContextBuilder is a class needed by the Repast environment in order to initialize the system and run the simulation for the portion which cannot be set from configuration wizards available once the model is launched.

This builder inherits from the Repast ContextBuilder and in its body firstly retrieve parameters which have to be set in the simulation menu and before the initialization phase which is launched with the reserved button. The list of this parameters can also be set in the parameters.xml file inside project's source tree but is is way easier to set it in the simulation window; like in many other java project parameters usually need an explicit cast to their primitive type.

With these parameters one utility object is instantiated, this is mainly for a sake of clarity and source code readability, in fact this object is used as a common configuration for every node and to pass to the SuperAgent parameters it needs.

At the end of the **build()** method of the ContextBuilder there is the creation of SuperAgent and of all nodes; nodes are immediately added to the context of the simulation and then the list passed to the SuperAgent which concludes their setup as we have seen above.

Once also the SuperAgent is added to the context it is possible to start its inner threads and then finish the initialization of the simulation.

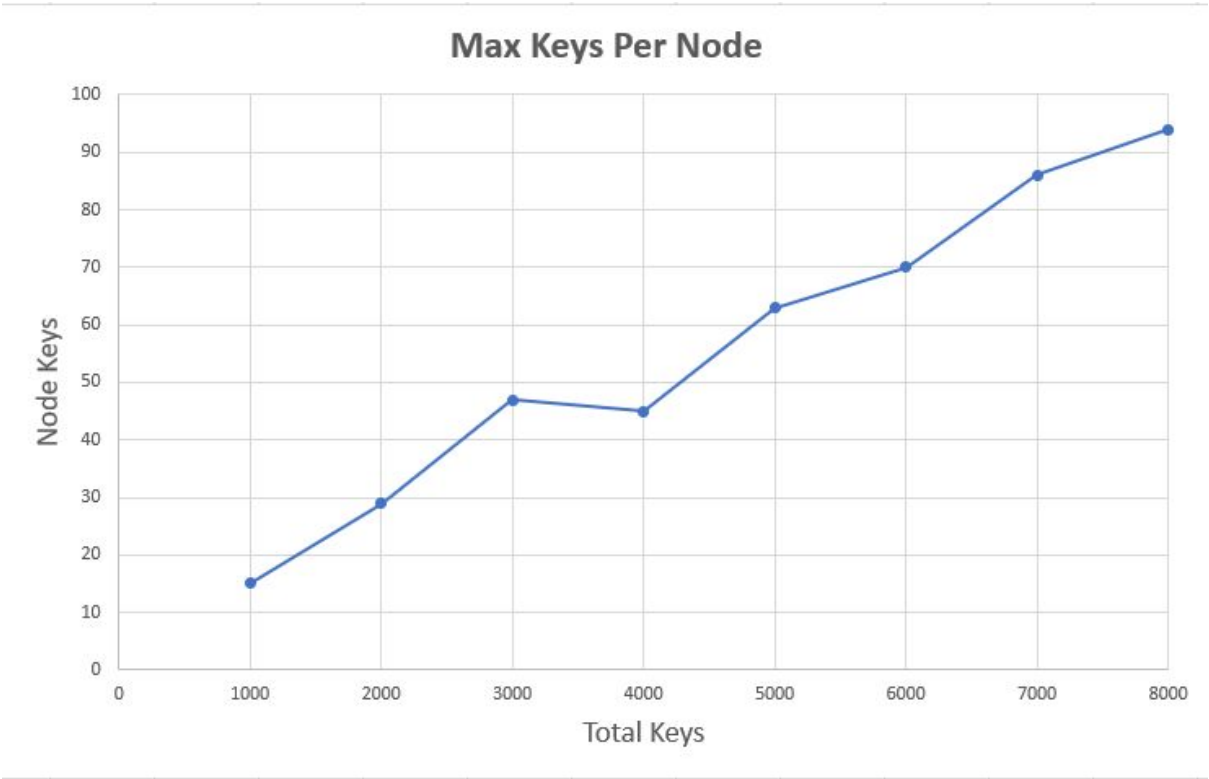
Analysis

The purpose of this section is to replicate some analytical or experimental evaluations of the paper regarding this protocol, after implemented it using Repast framework. To create the ideal scenario for a particular analysis, sometimes we had to adapt the code itself. To obtain our results, we used ad-hoc methods called by particular datasets, used to create graphs and files containing the outputs of the simulation run in certain conditions. All of this is offered directly by Repast.

We decided to focus on the two most important factors of the suggested protocol: the maximum number of keys per node and the maximum path length of a query.

KEYS PER NODE

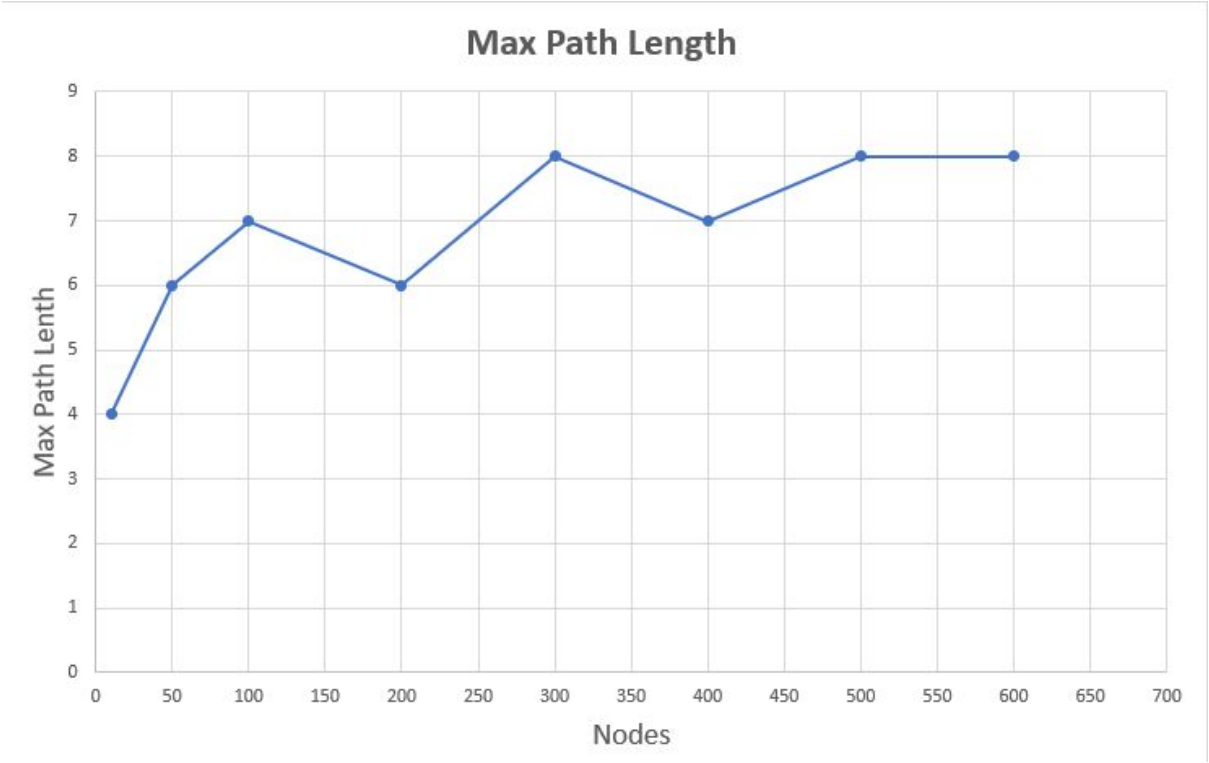
The first analysis takes in consideration how keys are stored into the joined nodes; in fact, chord is mostly used to distribute among the participants the various pairs key-value. One of the main characteristics of the protocol is the homogeneous distribution of those keys within the network: in particular, considering ‘n’ nodes and ‘k’ keys, every node should have k/n keys on average. Since this property is very important in chord, we conducted different simulations in order to identify how in a concrete system random generated keys are distributed.
For time constraints, we were not able to use the same numbers of the paper. Instead, we considered as number of created keys a minimum of 1000 and a maximum of 8000, in eight different simulations, keeping a fixed number of nodes, which were 500. In particular, for this particular analysis we first of all waited for every node was joined, and after that we started to create and spread the keys among the network, with a speed of 10 keys per second.



Instead of considering the number of keys on average per node, we focused on the maximum number. This by the fact that there is no sense in calculating the average of such a number, that would be in any case k/n. This way, instead, we can observe the upper bound, being sure that all other nodes held a number of keys lower than or equal to what shown in the graph.
Talking about the results, we can notice a quite linear andament of the number of keys per node with respect to the total created keys. This confirms that there is not a preference in storing keys into a particular set of nodes, since the more keys are generated the more keys a generic node is going to hold. About the numbers, though, we can notice that they are higher than that we expect theoretically. In fact, considering 500 nodes and 1000 keys (the first simulation), the maximum number of nodes should be 2, while the physical result is 15. The same is for every simulation, like more than 60 keys per node instead of 10, with 5000 total keys, or 90 keys per node instead of 16 for the last simulation. We remind that the number shown is the maximum value, so we can't know how is the situation on average, neither we can imagine that all nodes were storing the same amount of keys.

PATH LENGTH

In chord you can reach any node in a very small amount of time, without storing the list of all the joined nodes. Basically, the query of a certain key or node is forwarded from a node to another once the right one receives the query, so it will answer to the original sender.
Having such a short answer of a query within a scalable protocol is the most important aspect of chord. For this reason, we decided to conduct the second analysis on the maximum path length of a query before getting an answer. The length will start from 0 and will be incremented by one every time a node receives and forwards the query.
In particular, the simulation consists in asking every second (once every node has been joined the network) the value of a particular key, already created and stored by the liable node. Then, every node checks if the key is into its list; if so, it will answer with the value, together will the total length, otherwise it will forward the query to its closest preceding node using the 'findsuccessor()' function recursively, increasing the length of the path.



Then, a variable in the super agent (that simulates a real application) stores the maximum path length, checking for every answered query if the length is higher than this variable or not. We decided to keep track of the maximum value rather than the average andament since for us knowing the worst scenario were more realistic and significant.
Also here, for computation constraints we couldn't use thousands of nodes for better understand the andament like the paper describes. Therefore, we conducted eight different simulations, starting from only 10 nodes up to 600 nodes for the last one. All this keeping a fixed number of total created keys, which was 500.
Since the difference regarding the number of nodes were not so big like in the theoretical graph of the paper, also the results are not so meaningful. Starting from 4 forwards of the query as maximum value for 10 nodes, we had a pique of 6 forwards for 50 nodes and another increasement with 100 nodes. But after that, the maximum path length remained to 7 or 8 for every other simulation. Moreover, the result numbers are quite bigger than was ipotized theoretically (7 concrete forwards with 100 nodes compared to 3 forwards considered in the paper).
Anyway, we have to keep in mind that every query even with hundreds of nodes receive the response after passing through a maximum number of 8 nodes.