Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications

Alessandro Cacco
mat. 203345
alessandro.cacco@studenti.unitn.it

Andrea Ferigo
mat. 207486
andrea.ferigo@studenti.unitn.it

Enrico Zardini
mat. 207465
enrico.zardini@studenti.unitn.it

1 Introduction

This work aims at illustrating an implementation of Chord, a scalable distributed lookup protocol described in [1]. Basically, Chord provides a primitive, i.e. *lookup*, that allows to determine the responsible for a key in an efficient way. Hence, it represents a great solution to the data location problem: each data item needs just to be associated with a key and stored in the node to which the key is mapped.
In practice, the nodes are logically arranged in a ring topology and each of them is responsible for the ids belonging to the interval $(predecessorId, nodeId]$ [1]. The key-data pairs are assigned to the nodes depending on the hash value of the key and consistent hashing is used in order to keep the load balanced. Moreover, each node is required to maintain information about only a few other nodes: the predecessor, some successors and the elements of the finger table[2]. Therefore, Chord scales well to large numbers of nodes without affecting performance. Actually, it adapts effectively also in dynamic environments with frequent joins and leaves thanks to a simple stabilization algorithm.
Finally, it is worth mentioning that the iterative version of Chord has been implemented. Hence, a node resolving a lookup initiates all the communications needed to reach the target.
Starting from this, Section 2 will describe in detail the implementation, Section 3 will present the view that has been developed to show the protocol's functioning and Section 4 will describe the simulations that have been performed and the results obtained.

[1]The ids are integers in $[0, 2^m)$, where m is the number of bits of the identifiers. All the arithmetic is modulo 2^m.

[2]The finger table is a routing table: the i^{th} entry points at the responsible for the identifier $nodeId + 2^{i-1}$, with $1 \leq i \leq$ m.

2 Implementation

This section presents the methods and the behaviour of the five classes that have been used to implement the protocol. In particular, these classes can be subdivided into three groups:

- the control class (`TopologyBuilder`) that instantiates the nodes, initialises the ring, manages joins/leaves and initiates the lookups;
- the class (`Node`) that defines the behaviour of

the agents in the simulation, i.e. the nodes;

- the support classes (`FingerTable`, `Lookup` and `Utils`).

Before going into details, it is worth mentioning that the correspondence between simulation ticks and seconds has been assumed in both implementation and simulations.

2.1 TopologyBuilder

As introduced in the previous section the `TopologyBuilder` has several duties, which can be grouped into two phases: initialization and scheduling. During the first one it initialises the nodes and generates the data. Instead, in the second one it schedules insertions/leavings and lookups.

2.1.1 Initialization

The first thing the `TopologyBuilder` does is initialise the ring with a certain number of nodes, which is defined by the parameter *init_num_nodes*. Actually, the ring can be initialised in two different ways[3]. As regards the first one, the nodes are inserted one at a time providing them with a reference to a random node already present in the network. In particular, between two consecutive insertions the `TopologyBuilder` waits a certain number of ticks, which is defined by the parameter *insertion_delay* [4]. It is also worth mentioning that the first inserted node does not execute the *join()* method, but the *create()* one, as highlighted in Section 2.2.1. As for the other mode, all *init_num_nodes* nodes are inserted at the same time without any delay. Moreover, each of them is provided with the right immediate successor in order to start the simulation with a ring that has already reached a stable state. Hence, no *create()* or *join()* method is called. Instead, *initSuccessor()* is used.

The second thing that the `TopologyBuilder` does during the initialization is the generation of the key-data pairs, which is performed after a stabilisation round since the last insertions. In particular, the `TopologyBuilder` creates the data according to 3 parameters: *data_size*, *key_size* and *total_number_data* which correspond to the size of the random strings that represent the data, the size of the sub-strings used as keys, and the total number of data stored in the ring, respectively[5]. At the end of the data generation, the `TopologyBuilder` assigns each key to the first active node that has an id greater than or equal to the hash value of the key[6].

2.1.2 Scheduling

At the end of the initialization, the `TopologyBuilder` schedules two types of functions:

- the functions managing the leaving/insertion process (*leaving_nodes()* and *join_new_nodes()*);
- the function initiating the lookups (*lookupSingleKey()* or *lookupMultipleKeys()*)

As regards the former type, the leavings are scheduled every *leave_interval* ticks starting after one stabilization since the last insertion. In particular, a set

[3] It is possible to change the mode using the parameter *one_at_time_init*.

[4] The *insertion_delay* is always greater than the maximum time needed for a stabilization, i.e. there is always a stabilization round between two consecutive insertions

[5] Note that the sizes indicate the number of characters. Moreover, it is important to highlight that there cannot be more than 2^m data.

[6] Clearly, all the keys greater than the id of the last active node are assigned to the node with the smallest id in order to respect the modular structure of the ring.

of active nodes[7] is randomly selected every time and their leavings are scheduled at a distance of one tick from each other. This is done in order to allow the nodes to notify the neighbours and to transfer the data. Then, after *join_interval* ticks since the last leaving, the insertion procedure is started. Analogously, a set of random inactive nodes[8] is selected; however, in this case, unlike the leavings and the initialization phase, the insertions are concurrent with no delay between them.
Actually, it is worth mentioning that this is not the only way a node can leave the ring: as described in Section 2.2.6, a node may be forced to leave due lack of alive successors. In that case, it is removed from the active nodes by the `TopologyBuilder` and an additional join is scheduled at the next insertion round.
As regards the lookups, the procedure is very similar: every *lookup_interval* ticks since the data generation, the `TopologyBuilder` schedules the initiation of *number_lookup* queries[9]. Actually, there are two different lookup modes: all nodes look for the same key (*lookupSingleKey()*); every node looks for a random key (*lookupMultipleKeys()*) [10].

2.2 Node

The `Node` class, which defines the behaviour of the agents in the simulations, consists of several methods that can be grouped into six categories:

- initialization;
- lookup;
- stabilization;
- data management;
- crash and recovery;
- leaving.

Each of them will be examined in a specific Section.
As regards the main fields, every node maintains: a finger table (the implementation will be presented in Section 2.3.1), a list of successors, a reference to the predecessor and a key-value map for data.
Basically, the finger table - as a routing table - allows to perform efficient and scalable lookups. In fact, the finger table is not needed for correctness[11] but avoids lookups characterised by a number of messages linear in the number of nodes.
Instead, the list of successors improves the robustness to failures and departures w.r.t. just a reference to the immediate successor. In fact, all successors would have to fail simultaneously in order to destroy the ring. It is worth mentioning that the first elements of the finger table and the successor list coincide.
As regards the reference to the predecessor, it is exploited by the stabilization algorithm to maintain the consistency of the successors and is used to transfer part of the data to the new responsible in presence of a join.

2.2.1 Initialization

As described in Section 2.1, two ring initialization modalities are supported: the insertion of one node at a time until the desired size has been reached; the creation of a ring of the desired size where each node knows the immediate successor.
The former is supported by the *create()* and *join()* methods: *create()* is called on the first node in order

[7] The number of nodes leaving the ring is given by the combination of two parameters: *min_number_leaving* and *leaving_amplitude*.

[8] The number of nodes joining the ring is given by the combination of *min_number_joins* and *join_amplitude*.

[9] In case the number of alive nodes is less than *number_lookup*, all of them initiate only one query, since a node can perform only one lookup at a time.

[10] The mode is defined by the parameter *one_key_lookup*.

[11] It is sufficient that every node knows its successor to achieve correctness.

to build a new Chord ring, whereas *join()* performs all the operations required to join an existing ring. Hence, *join()* is used also after the initial phase of the simulation to insert new nodes in the ring.
Instead, the latter is supported by the *initSuccessor()* method, which initialises the successor list using the node provided.
As regards joining an existing ring, it implies asking a node already contained in the ring for the successor of the current node. This is done through the *find_successor_step()* method, which is presented in the following Section.

2.2.2 Lookup

As introduced in Section 1, the lookups are performed in an iterative way. Hence, the node on which the *lookup()* method is called initiates all the communications needed to reach the responsible for the given key.
Basically, the node first checks if its successor is the responsible for the key of interest: if it is, the lookup is finished; otherwise it asks the closest preceding node (w.r.t. the given key) among the ones contained in the finger table and in the successor list. Since the iterative version has been implemented, if the contacted node's successor is not the responsible, the contacted node provides the lookup initiator with a reference to the closest preceding node among the ones it knows. The procedure is repeated until the responsible is found.
Since the same steps are performed during the stabilization, the *lookup()* method just calls another function, i.e. *find_successor()*. In practice, *find_successor()* does the first check, whereas the *find_successor_step()* method implements the iterative step. As regards the communications between nodes, they are performed through the methods *processSuccRequest()* and *processSuccResponse()* using an exponentially distributed packet delay with mean of 50 milliseconds (0.05 ticks) as in [1].
Actually, the lookup initiator may not receive an answer due to a failure or an out-of-date reference (the target node has left the ring). In that case, after 500 milliseconds (0.5 ticks) it contacts the node from which it has learnt about that node in order to obtain another reference (*getPrevSuccessor()* method). If that node does not reply either, the lookup initiator iterates the procedure.

2.2.3 Stabilization

The stabilization protocol is run periodically to ensure that the various pointers (finger table, successors, predecessor) are up-to-date. The main method is *stabilization()*, which starts the various operations.
First of all, the node contacts sequentially its successors until it finds one alive (this is done in *stabilization()*). At that point, if the predecessor of the replying node belongs to the interval $(nodeId, successorId)$ the current node updates its immediate successor (*stabilization_step()* method) [12]. The next step consists in asking the immediate successor for its successor list, which is used to update the list of the current node. As regards the communication, it is managed through the methods *processStabRequest()* and *processStabResponse()*. In particular, the former method also verifies if the predecessor of the contacted node needs to be changed (*notifiedPredecessor()* method). In that case, the contacted node transfers the necessary data -if any- to the new predecessor and notifies the old predecessor that its successor should be changed (*setNewSuccessor()* method) accelerating the integration of a new node in the ring.
Once the response has been processed, the node moves on to the stabilization of the other pointers.

[12] This may happen in case a new node has recently joined the ring.

This is done in the *fix_data_structures()* method: basically, the node stabilizes alternatively one entry of the finger table (*fix_fingers()* method) and one entry of the successor list (*fix_successors()* method) in addition to the predecessor. As regards the finger table, the node calls the *find_successor()* method in order to find the responsible for the id the entry points to. Instead, as for the successor list, the node calls the same method on the id of the last stabilized successor plus one. Once the responsible has been found, the corresponding data structure is updated by the method *setResult()*. In particular, since the first entry of the two data structures represent the immediate successor, they are never stabilized during this phase (it has already been done in the previous one). Finally, the node checks if the predecessor is still alive: if it is not, the pointer is set to `null`, which allows the node to accept a new predecessor in the next stabilization.

2.2.4 Data management

The data management category includes two methods: *transferDataUpToKey()* and *newData()*.
Basically, the former extracts the data with a hash value of the key up to a target value from the data owned by the current node, whereas the latter performs the acquisition of new data. In practice, these methods are exploited to transfer data: to the new predecessor; to the successor before leaving the ring (this operation is managed by the `TopologyBuilder`).

2.2.5 Crash and recovery

Nodes failures are simulated as follows: each node periodically crashes with a certain probability (*nodeCrash()* method) and schedules the recovery after a specific interval (*recovery()* method). While failed the node does not process any request or response. Instead, once it is up again, it performs immediately a stabilization using the references it had before crashing.

2.2.6 Leaving

As described in Section 2.1.2, nodes leavings are decided and scheduled by the `TopologyBuilder`. However, all the operations that a node has to perform before leaving the ring are implemented in the *leave()* method of the `Node` class.
Basically, the node contacts first the immediate successor notifying it of the departure and providing it with the new predecessor -if it is not `null`- and the data the current node was holding (*setPredecessor()* and *newData()* methods). Then, it contacts the predecessor providing it with its immediate and last successor, which are used to update the successor list of the predecessor node (*setLastSuccessor()* method). At that point, the node can safely leave the ring and clear all its data structures (*clearAll()* mehtod).
Actually, a node may also be forced to leave the ring. In fact, if during a stabilization or a lookup operation it realizes that all its successors are failed or have left the ring, it is forced to leave the ring. This is managed by the *forcedLeaving()* method, which notifies the predecessor of the departure (*successorLeaving()* method), clears all the data structures and communicates the leaving to the `TopologyBuilder` that will insert an additional node in the next join round.

2.3 Support classes

This section briefly describes the structure of the three classes that support the operations of `TopologyBuilder` and `Node`, i.e.: `FingerTable`, `Lookup` and `Utils`.

2.3.1 FingerTable

The `FingerTable` class defines the structure of the finger table, which is nothing more than a routing table, as described in Footnote 2. In practice, the finger table has been implemented as an hash-map: the key corresponds to the index of the routing table, whereas the value is the reference to the pointed node. Moreover, the `FingerTable` class provides all the necessary methods to interact with the finger table, such as *getEntry()*, *setEntry()*, *getKeys()*, *removeEntry()* etc.

2.3.2 Lookup

The `Lookup` class provides a useful structure for keeping track of the information related to a specific lookup.
In particular, when the `TopologyBuilder` initiates a lookup, it creates a new instance of the `Lookup` class providing information such as the hash value of the target key, the id of the node performing the lookup, the starting tick and the correct result (id).
Once the lookup has been finished, the lookup initiator calls the *setResult()* method on the corresponding `Lookup` instance providing the result information, i.e. the reference to the node responsible for the target key, the path length, the number of timeouts and the number of nodes contacted. At that point, a check to verify if the right node has been found and it has effectively the key is performed.

2.3.3 Utils

The `Utils` class provides three static utility methods: *getHash()*, *getNextDelay()* and *belongsToInterval()*.
The first one computes the hash value of the key provided (SHA-1 is used). The second one returns a random delay taken from an exponential distribution with mean equal to the one supplied. The last one verifies if the value provided belongs to the interval defined by the endpoints supplied in modular arithmetic.

3 Simulator

This section deals with the view that has been developed to show the protocol's functioning.
In order to execute the simulator, the first step consists in importing the project. After that, the *Chord* model has to be run: the default graphic interface of Repast Symphony will be shown. At that point, the simulation parameters can be set in the *Parameters* panel. In particular, the initialisation strategy is determined by the checkbox labelled as *Init - initialise the ring one node at a time*. Instead, the *Scenario Tree* panel allows to select the data to be saved (`Data Sets` item) and the location (`Text Sinks` item). As regards the former, it is possible to choose the methods to call on the agents and the time to do that. As for the latter, the data set fields to be stored in the file can be selected. Actually, the lookup results are saved in a different way: in fact, at the end of the simulation the `TopologyBuilder` writes the collected data to a file through the method *getLookupsResults()* (this behaviour is defined at the end of the *build()* method of the same class). Once all this has been done, it is possible to start the simulation. Moreover, the tick at which to pause or stop it can be set in the `Run Options` panel.

4 Analyses and Results

This section presents the simulation runs that have been executed to test the protocol strength and per-

formance, with their respective analyses and results explanations. The parameters involved in our simulations are explained in Table 1, with the corresponding default parameters. The performed runs are designed with variations on the default parameters so four analyses are possible, namely:

- Crash probability analysis (Section 4.1), in order to understand the impact of having a higher number of nodes crashing during the protocol, causing errors in the other nodes data structures;
- Network size analysis (Section 4.2), with proportioned lookup batch sizes, allows both for scaling testing and for lookup load stress-testing;
- Key number analysis (Section 4.3), testing the scaling performance of the protocol wrt the number of keys in the system

Table 1: Default protocol parameters for the executed experimental runs.

Parameter	Value
Crash probability	0.05
Crash recovery interval	25
Crash scheduling interval	60
Key size	5
Number of keys	1500
Value size	10
Hash size	12
Initial number of nodes	1000
Successors list	20
Join tick period	20
Join amplitude	0
Join min number of nodes	10
Leave tick period	50
Leave amplitude	0
Leave min number of nodes	10
Lookup period	35
Lookups per batch	500
Single key lookup	no
Stabilization amplitude	1
Stabilization period	15

4.1 Impact of crashing nodes

Table 2 lists the runs performed in order to analyse the impact of higher percentages of crashing nodes. First of all, the plot in Figure 1 validate the experimental runs, as we can see how the number of nodes is consistent throughout each run, with a reasonable expected difference between them.

Table 2: Variations of the default parameters for the node crashing impact analysis. Note that `R01` is the default run with no variations

Run	Crash probability
`R01`	0.05
`R02`	0.1
`R03`	0.2
`R04`	0.3

The plots (Figures 2 to 7) clearly shows how the protocol is resistant even to a third of the total nodes periodically experiencing a sudden crash. Particularly, Figure 2 shows how the number of steps needed for a lookup slightly increases in mean and variance with the crashing probability, consequently increasing the average lookup duration which quickly stabilizes after an initial transient state, as can be observed in Figure 3.

Furthermore, Figure 4 confirms that lookups takes longer due to a higher quantity of contacted nodes, not because the lookups find valid fingers, value which also stabilizes after an initial transient state (around ~ 400 ticks) as shown in Figure 5. The crash probability parameter has also an impact on the num-

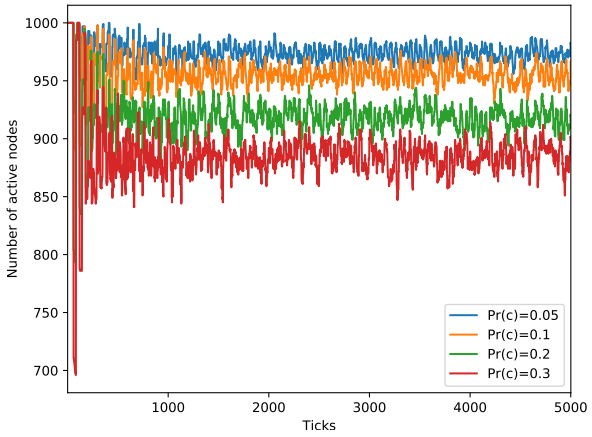

Figure 1: Number of working nodes throughout the runs

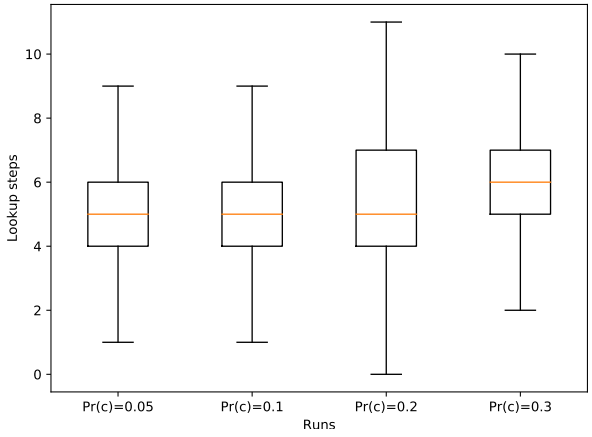

Figure 2: Boxplot showing the number of steps needed to complete a lookup for each run

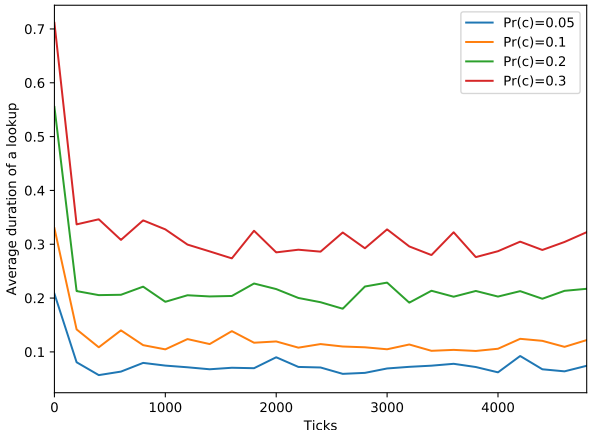

Figure 3: Average duration of lookups throughout each run

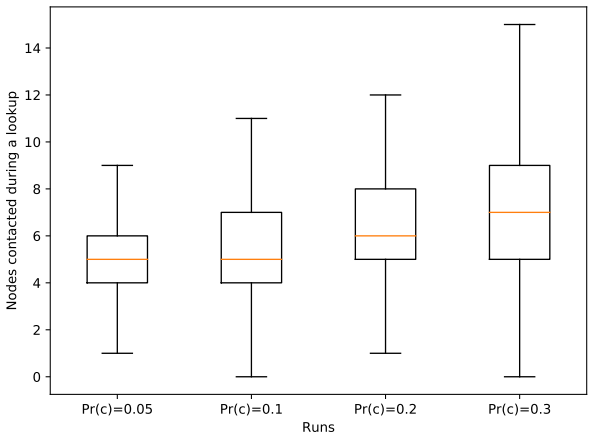

Figure 4: Boxplot of the number of nodes contacted to complete a lookup for each run

ber of timeout which are experienced by a lookup, in particular in the initial phase of the protocol in which the data structures are not yet fully populated (see Figure 6).

Intuitively, a higher crashing probability should cause more lookup failures, however the protocol resistance is again shown in Figure 7, where the number of failed lookups ($\sim 4 \times 10^1$) is negligible wrt the total number of lookups ($\sim 7 \times 10^4$).

An interesting analysis is the one presented in Figure 8, where the number of error present in the nodes data structures can be observed. As expected, a greater quantity of crashing nodes implies more errors, which are still contained and constant throughout the runs.

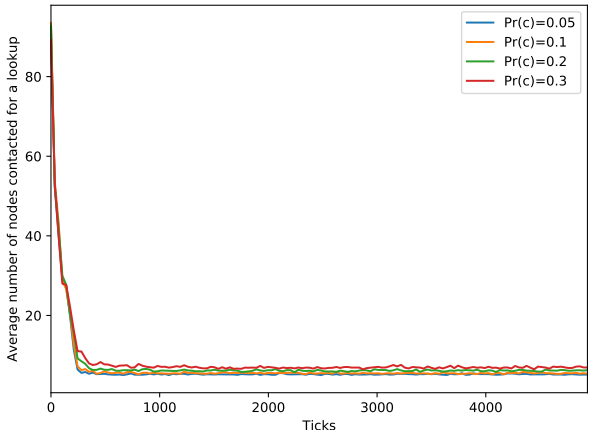

Figure 5: Average number of nodes contacted for a lookup throughout each run

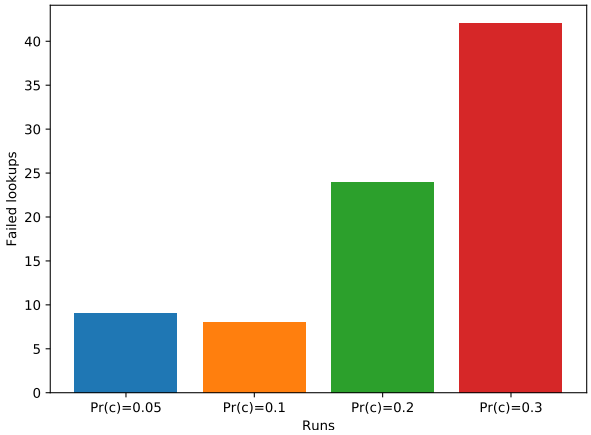

Figure 7: Number of failed lookup. Note that the total number of lookups for these runs is $\sim 7 \times 10^4$

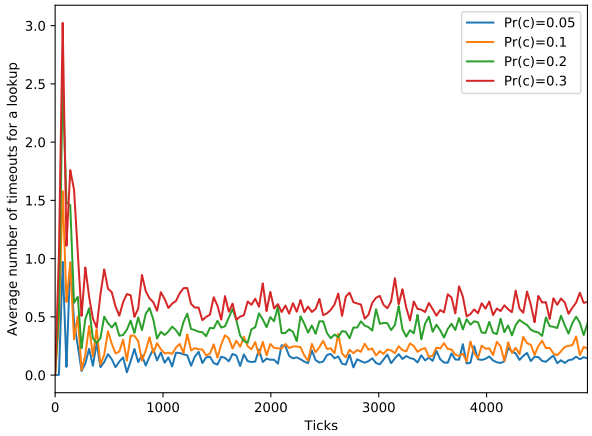

Figure 6: Timeouts experienced by nodes during lookups throughout each run

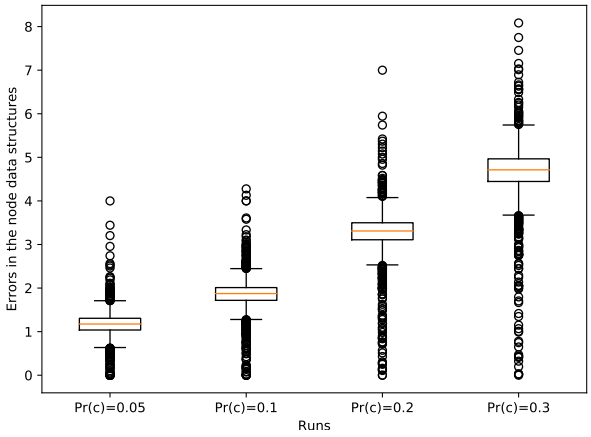

Figure 8: Number of errors in the nodes data structures, i.e. nodes that unsubscribed or crashed in the finger table and/or in the successors list

4.2 Protocol network size scaling

Further tests have been performed in order to verify the scalability of the algorithm, with the runs listed in Table 3. Several parameters have been varied parallelly, so that the values matches meaningful realistic situations.

The plot in Figure 9 shows how the duration of a lookup increases with a bigger network, however the average lookup length is still very low, showing how the

Table 3: Variations of the default parameters for the scaling analysis runs. `R01` is omitted being the default run with no variations.

`R05`	
Number of keys	30
Hash size	5
Initial number of nodes	25
Successors list	8
Join min number of nodes	2
Leave min number of nodes	2
Lookups per batch	10
`R06`	
Number of keys	60
Hash size	6
Initial number of nodes	50
Successors list	10
Join min number of nodes	3
Leave min number of nodes	3
Lookups per batch	25
`R07`	
Number of keys	1000
Hash size	10
Initial number of nodes	500
Successors list	15
Join min number of nodes	5
Leave min number of nodes	5
Lookups per batch	250
`R01`	
...	

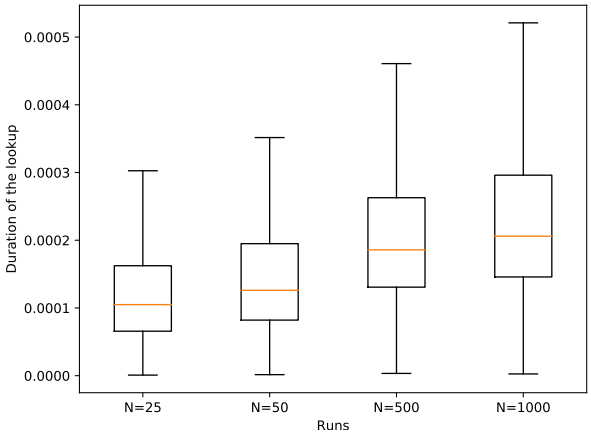

Figure 9: Boxplot of the lookups durations for each run

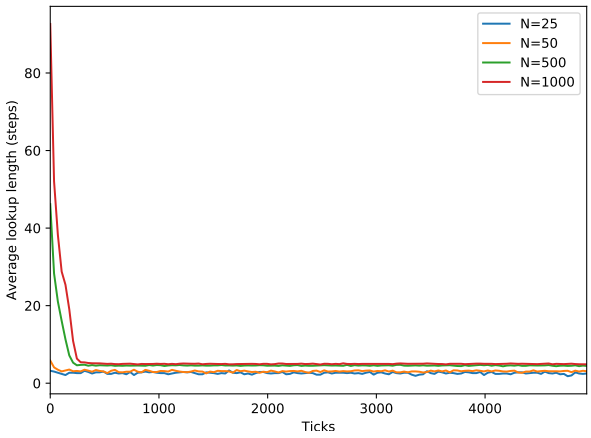

Figure 10: Average duration of lookups throughout each run

4.3 Performance and quantity of keys

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, F. Kaashoek, F. Dabek and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications", *IEEE Transactions on Networking*, vol. 11, Feb. 2003. DOI: 10.1109/TNET.2002.808407.

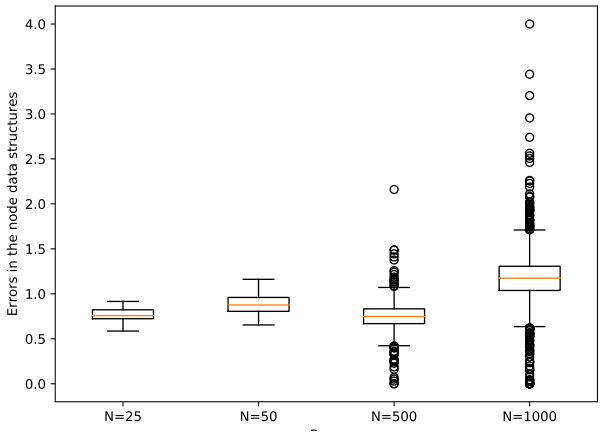

Figure 11: test

Table 4: Variations of the default parameters for the key quantity analysis.

Run	**Number of keys**
R01	1000
R02	2000
R03	3000
R03	4000