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 graphical simulator 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 indicated in the previous section the `TopologyBuilder` has different duties, which can be divided into two macro-phase: the initialization and the scheduling. During the first phase the node and the data are initialized, instead in the second are scheduled both the join/leave process for nodes and the lookups.

2.1.1 Initialization

First of all the `TopologyBuilder` initialise the ring with a number of nodes indicated by the parameter *init_num_nodes*, the nodes can be initialise in two different ways[3]. In the first mode, each node is inserted one at time using as successor a random node already present in the network and leaving between two insertion a time indicated by parameter *inseriton_delay*, which is greater than the maximum time needed for a stabilization, i.e. there is always a stabilization round between two insertions. Note that, the first node inserted does not make a *(*join()) but call the method *create()*, as highlight in Section 2.2.1. The other procedure proceed in a different way, indeed each node is inserted with his correct successor without delay between the insertion and no *create()* method is called, this in order to simulate a ring which as already reached the initial structure. The second thing that the node have to initialise are the data, after a stabilisation round from the last join the `TopologyBuilder` create the data according to 3 parameter: *data_size*, *key_size* and *total_number_data* which indicates the size of the random string that symbolize the data, the size of the sub-string used as key, and the total number of data present in the ring, respectively[4]. After that all the data are generated, the `TopologyBuilder` assign each key to first node active which has a id greater than the key[5].

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

2.1.2 Scheduling

As introduced the function which are scheduled by the `TopologyBuilder` are of two different type, the first manage the leave/join process, the other manages the creations of the lookup.
The first leaving is scheduled after the last node inserted have done a stabilisation plus a *leave_interval* and is repeated each *leave_interval*. During a *leaving* a set of active nodes[6] is selected and is scheduled their exit at distance of one tick from each other, this is done, in order to have the time to advice the neighbours and transfer the data. After that the last node left the ring, a number of tick equals to *join_interval* ticks are waited before start the joining procedure. During the *joining* a set of nodes[7] enter in the ring, in contrast to the initialization phase, there is any delay between two insertions, so in order to maintain the

[4]Note that the size indicates the number of characters used to create the string. Moreover is important to remember that there can not be more than $2^m - 1$ data.

[5]Clearly all the keys greater to the last active node, are given to the smallest node in order to respect the modular structure of the ring.

[6]The number of nodes leaving the ring can be setted trough two parameters: *min_number_leaving* and *leaving_amplitude*.

[7]As for the leaving the number of nodes entering can be decided trough two parameters.

consistency each joining node cannot choose a node which has just joined as his successor. Is important to note that this is not the only way in which a node can leave, indeed, as described in **??**, if a node is forced to leaving it is removed from the active nodes, and a node in more is scheduled to join at the next join procedure.
As regards to the lookups the procedure is similar to what previously described, *lookup_interval* ticks after the data generation, the `TopologyBuilder` schedules the creation of *number_lookup* query each *lookup_interval*[8]. There can be two different behaviour used to create the lookups: one where each query search the same key and one where each query search for a different key.

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[9] 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 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.

[8]In case that the number of nodes initialized and not crashed is lesser than the *number_lookup* all the alive nodes create a query, as a node can do only a lookup at once.

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

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) [10]. 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. 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,

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

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

4 Analyses and Results

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.