MidTerm Peer2Peer Systems and blockchains

Analysing the Kademlia DHT

Lorenzo Bellomo, 531423

April 23, 2019

Project Usage and parameters

The project can be executed both using command line arguments, or through the interactive shell. The three parameters to be passed for the execution are m, n and k, which are respectively the system wide parameters modelling the number of bits of the identifier space, the number of joining nodes in the network and the size of the routing table buckets.

The only (fixed) system wide parameter is *alpha*, which is fixed as 3 and is used in the Kademlia lookup procedure.

Class overview

The project comes organized in 6 classes

- *Coordinator*: This class models the *Coordinator* entity (which has a global view of the network). It keeps a collection of *NodeDescriptor* instances (one for each node joining the network), together with some statistics to be displayed at the end of the execution.

  The Coordinator is the only entity which is spawned by the Main class, and it provides as only public method the create network. This method, which is the one invoked by the main, calls two private methods: *initialize* (which resembles the *2a* phase in the MidTerm text) and *generateNewNode* (which instead models the points *2b* and *2b*).

  The two other main functions of the Coordinator are:

  - At the end of the creation of the network some *statistics* are computed and displayed through the command line
  - It acts as a *RPC registry* in the Kademlia network. In this way nodes that want to execute a *remote procedure call* on another node, have to ask the instance to the coordinator.

- *Main*: This class is the one with the main function, which simply asks for the three system wide parameters and spawns the Coordinator life cycle.

- *Node*: This class models the node entity, as a triple of *IP address* (modelled as a *String* in order to avoid exceptions relative to non-existing IP addresses), *identifier* (a *BigInteger* instance) and *UDP port* (which is not used in this simulation).

- *RoutingTable*: This class models a routing table of the Kademlia network. It consists of a kind-of Map structure, where each node keeps m bucket, each one of size k. This class provides methods to update the routing table, finding the best entries currently present in it. The procedure adopted for routing table update is the one proposed by Kademlia, in which the least recently seen node is pinged. Nevertheless, the network to be created is a *churn-free* one, which means that the ping method is a fake one, that always acts as if the nodes were alive.

- *NodeDescriptor*: This is the class that models the view that the Coordinator has of each node. This class basically models an actual node in the network, as a couple <Node, RoutingTable>. This class is the one over which the *Remote Procedure Calls* are instantiated. The main methods that are provided by this class are:
  - *findNode*: This method is the one exposed in the Kademlia original paper. The only main difference resides in the fact that the parameter passed to this method, in addition to the id, is the list of travelled nodes (needed in order to allow a new node to become known in the network). It was chosen to pass this list as a parameter in order to avoid the burden of modelling RPC messages in this simulation.
  - *nodeLookup*: This is the Kademlia lookup procedure. It finds in the network the closest k nodes to the target id passed as parameter. It doesn't explicitly exploit the bootstrap node (the one the Coordinator generates for him), but it just uses the best known nodes to get to fill the routing table k-buckets.
- *Utils*: This class simply provides some utility function (mainly to be used by the class *NodeDescriptor*) in the form of public static methods. This is the class that provides a function to generate a new node, and which is exploited by the *Coordinator* in order to populate the network.

Data Structures

The main data structure to be presented in this report is the one used in the routing tables. It has to model 160 buckets, each one of size at most k. The final choice was to use an ArrayList of 160 positions, and for each entry, using a Queue (a LinkedList) of Node classes. In order to access the right bucket out of the 160, whenever an update to the routing table has to be issued, then a *findBucketIndex(id)*. This method computes the bucket index as $log_2 d$, where d is computed as $myId \oplus targetId$.

The only other data structure worth mentioning is the HashMap kept by the coordinator and containing all the NodeDescriptor instances.

Stats Display

At the end of the creation of the whole network, some statistics are displayed through the command line. Their purpose is mainly to show what happens with different network wide parameters, and also compute an expected value for the number of edges.

The expected value of edges is computed as follows:

- A number of random identifiers is generated (i.e. 20), and for each of them, the number of nodes "falling" in every *k-bucket* is computed.

- At the end of this process, an array of m position is filled, where in position i the average number of nodes falling in this k-bucket for a generic node is stored. This way I can estimate the number of edges for each bucket i in this way:
  - *if the number is greater than k* then this bucket is probably getting filled, so the process adds k to the total number of edges
  - *otherwise* this bucket has a lot less edges to offer to the whole network, so the process adds the value computer averaging on the sample generated in advance.
- Throughout this process, by summing the temporary values, it is possible to compute the average number of outgoing edges from a generic routing table.
- The total number of estimated outgoing edges is computed by simply multiplying this value with the total number of nodes.

This value should be considered valid with reasonable certainty because the Kademlia process is, by construction, expected to converge with a few number of steps, and get to the *best solution* in a few steps. In order to validate this hypothesis, another stats comes helping: the *recursive lookup depth*. For each lookup call, the recursive depth of given call is computed by counting the times that the loop in the lookup process is executed. At the end of the construction of the network, for each recursive depth, the number of times it was reached is displayed. This value, in order to respect the *fast convergence hypothesis* of the Kademlia protocol, should be not only low on average, but with also relatively low variance. Other stats that are displayed at the end of the network creations are:

- The *time* needed to generate the network.
- The *expected number of nodes* for each distance (bucket) for a generic node.
- The number of registered *collisions* for each hash function.
- The value *maxTheoretical*, which is the number of edges a theoretical network with every k-bucket of every routing table being filled would have. This value has to be considered an upper bound to the number of total edges in the network, as not enough nodes for each bucket can be present in the network, and it is only displayed for comparison purposes. This value is computed by summing up k for each bucket, except for the first ones (for example bucket 1 accounts for 1, bucket 2 accounts for 2...).

Node Join

A node join is organized as follows:

1. the Coordinator entity creates a new node with a random IP address. The identifier is then built by using the SHA1 function on given IP address, and computing this value modulo 2^m. The identifier value is stored using the Java class *BigInteger*, which belongs to the *java.crypto* package. This class provides a number of methods for special operations of common use in cryptography, like XOR($\oplus$) or modulo(%), which are useful also for the Kademlia protocol. The BigInteger class also provides also allows the use of long identifiers, without bounds to the length (for example 2^{64} overflows for longs in Java). Nevertheless generating a new node does not prevent collisions like SHA1 does, since the identifier space is shrank to $[0, 2^m)$. The Coordinator checks if the newly generated id is already in the network, and in case of a collision, it simply increments the collision counter and generates a new node

2. the Coordinator finds a bootstrap node for the newly generated node, so that the new node has an outgoing edge towards the network, useful for populating its routing table. The new node then issues one lookup procedure for its own identifier,

3. after the first lookup, the new node issues m lookups, one for each *k-bucket*.

4. at this point the node join procedure is finally complete. Every node which received a findNode call originated in the new node, had a chance to know its new identifier, and, if possible adds this new contact to its routing table. This only happens "if possible" since the algorithm applied to update the routing table resembles the original one, which by default keeps the "stablest" nodes, so the ones added to the routing table in the farthest moment in time. The last nodes have a considerably lower chance of being added, in this *churn free* environment, to any routing table. In conclusion, a higher *in-degree* should be expected for the first nodes that were generated, with respect to an actual working Kademlia network.

Topology Analysis

The following section addresses the network topology analysis part of the assignment. It will be organised in two parts:

- *Program analysis*: a first, more lightweight part, in which only the program generated parameters are analysed (so without the use of external tools for graph analysis). This part is useful in order to ensure the correctness of the procedure and to see some small details.

- *Tool analysis*: a second part, where a small number of networks are analysed in depth.

Program analysis

The main point of interest for this part is the correctness of the implemented Kademlia procedure. Regarding that, two important focus point have to be analysed: *number of edges* and *recursive depth*. The first one accounts for the correctness of the joining procedure, together with the fact that "old" nodes get a chance to know new nodes. The second point instead focuses on the *fast convergence* of the algorithm. Of course those values have to be coherent in different networks (i.e. filled ones, and almost empty ones).

All the following analysis is the result of the average of 5 runs for each network. The single values of those runs, and their averages, are present in the excel file *programAnalysis*, added as an attachment to the project.

There follows a naive analysis of some networks:

1. $< m, n, k >=< 10, 1000, 10 >$: This network is characterized by an almost full identifier space (1000 nodes in a 1024 range identifier space). Something to expect from such a filled network is a high number of collisions in the node generation phase, and a really high number of edges, according to the fact that a lot of buckets (also those ones relative to close neighbours) should be filled for the majority of nodes.

   The average values are:

   - The upper bound for the number of edges is 75000.
   - the number of edges is 74650, while the expected theoretical value is 75670. The percentage difference (so the variation) between those values is around $1,50\%$.
   - The average time to generate the network is 6,58 sec.
   - The average recursive lookup depth is 1,70 (so most of the times, the loop in the lookup is executed 1 or 2 times), while the maximum depth reached is 5.
   - The number of collisions with hash functions is 2835.

   The two highlights from this process are the values of average recursive depth, which is very low (so the procedure actually has a fast convergence, when not considering churn), and the number of edges, which comes really close to the expected one. The filling factor of this network (where $fillingFactor = numEdges/upperBound$) has been computed to be 99,53%. An important thing about the filling factor has to be clarified: this value is not the network filling factor, with respect to $numberOfNodes/numberOfIdentifiers$, but it is the filling factor of the routing table, with respect to the upper bound computed.

   This shows the fact that the simulation works well in a filled environment.

2. $< m, n, k >=< 20, 500, 5 >$: This network differs from the precedent because it has a significantly lower filling factor, as the range of identifier

is $[0, 2^{20})$ with only 500 nodes inside. This network should have a significantly lower number of edges with respect to the upper bound.

The average values are:

- The upper bound for the number of edges is 46000.
- the number of edges is 18646, while the expected theoretical value is 18925. The percentage difference (so the variation) between those values is around $1,41\%$.
- The average time to generate the network is 2,04 sec.
- The average recursive lookup depth is 2,11, while the maximum depth is 5.
- The number of collisions with hash functions is 0.

As expected, the number of collisions is dramatically lower (actually, 0 collisions over 5 runs). The recursive depth is a little bit increased, but the real important piece of information relies around the number of edges. The upper bound is far from reached, (filling factor is 40,45%), but the theoretical estimate is matched within 1,41% accuracy.

3. $<m, n, k>=<64, 10000, 20>$: This one is an example of a huge network. This is used to assert the goodness of the implemented procedure even in big networks.

   The average values are:

   - The upper bound for the number of edges is .
   - the number of edges is , while the expected theoretical value is . The percentage difference (so the variation) between those values is around %.
   - The average time to generate the network is min.
   - The average recursive lookup depth is , while the maximum depth is .
   - The number of collisions with hash functions is .

   The purpose of this simulation is to show that, even for a huge network, the recursive depth reached by the algorithm is very low.

Tool analysis

The tool used for graph analysis is *Gephi*.