Formal specification of the Kademlia Routing table and the Kad Routing table in Maude

Isabel Pita*[1] and Maria Inés Fernández Camacho*[1]

[1]Dept. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain, ipandreu@sip.ucm.es, minesfc@sip.ucm.es

1. Introduction

Distributed hash tables (DHTs) are designed to locate objects in distributed environments, like peer-to-peer (P2P) systems, without the need for a centralized server. There are a large number of existing DHTs proposals; the best known are: Chord [11], Pastry [10], CAN [9], and Kademlia [4]. However, only Kademlia has been implemented in P2P networks through the eMule[1] and aMule[2] clients. Kad is the name given to the implementation of Kademlia incorporated to eMule and aMule, and it shows important differences from the original. There are also two BitTorrent overlays that use Kademlia: one by Azureus clients[3] and one by many other clients including Mainline[4] and BitComet[5].

Although the different DHTs have been extensively studied through theoretical simulations and analysis, there is a lack of formal specifications for all of them. Bakhshi and Gurov give in [1] a formal verification of Chord's stabilization algorithm using the π-calculus. Lately Lu, Merz, and Weidenbach [3] have modeled Pastry's core routing algorithms in the specification language TLA$^+$. They consider the complete P2P network and focused their study on the lookup correctness property: *the lookup message for a particular key is answered by at most one 'ready' node covering the key.* The paper provides a detailed model of the network, the routing table and other Pastry structures, while abstract from an explicit notion of time. Periodic actions are performed non-deterministically. The TLA$^+$ model checker is used to check the model and illustrate some open issues related with the algorithm. Finally the TLA$^+$ theorem prover is been used to verify the correctness property.

There is a preliminary study of the Kademlia searching process protocol by the first author in [10], and a distributed specification of the protocol in [11]. Based on them, we have identified the need of a detailed study of the routing tables, where each peer stores contact information about others. This information is used in the network look-up process.

The original version of Kademlia differs from the real implementation made in Kad. A Kademlia routing table is a proper binary tree whose leaves are lists of at most k contacts, called k-buckets. A

*Research supported by MEC Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009-TIC-1465) .

[1]http://www.emule-project.net
[2]http://www.amule.org
[3]http://azureus.sourceforge.net
[4]http://www.bittorrent.com
[5]http://www.bitcomet.com

bucket is kept sorted by the time contacts were last seen, and is identified by the common prefix of the IDs it contains. Each bucket is responsible for a range of the node ID space. In fact, Kademlia routing table is a list of buckets. The bucket in the first level contains contacts whose first bit is the opposite to the first bit of the owner of the routing table, while he ID of the owner is in the range of the last bucket. Kademlia uses, when add the contact to the routing table, a tighter splitting rule where it is only allowed to split the bucket with common prefix equal to the one of the routing table owner. A KAD routing table is a left-balanced binary tree, and may be a complete tree up to level 4. The tree nodes are called routing zones, and the buckets, called now routing bins, are placed in the leaf nodes. Kad divides the table structure in levels and index (the horizontal distance in number of routing zones from the leftmost one on the same level) and uses the bitwise exclusive-or of the n-bit quantities (XOR) distance to the owner ID of the routing table, to store contacts in routing bins. Kad has a looser splitting rule that Kademlia, it allows to split a bin if its level is smaller than 4 or its index is smaller than 5, which results in more possible contacts. Kademlia has no specific actions to update its routing table periodically, checking for off line contacts only when inserting into full buckets, while KAD has actual maintenance tasks that run at set periods. Each peer has a creation and expiration time, and also a type has scale from 0 to 4. Type 4 means that the contact is rarely connected and will probably be eliminated at the next occasion.

In this paper we present our specification of the routing table in the formal language Maude, as well as some preliminary results on its properties. The Maude algebraic specification language, is based on rewriting logic [2]. It supports both equational and rewriting logic computations and offers simple and elegant time simulation resources. Since the specifications are directly executable, Maude can be used to prototype the systems as well as to prove properties of them.

Our specification includes features not previously modeled as sending messages to update the Kademlia routing table or raising events for populate and removing off line nodes in the Kad routing table. We have focused our work on the routing tables and abstract from the rest of the network. In order to model maintenance cycles, and eventually other effects of time on the system, we turn to the real time extension of Maude [6]. Our model should allow us to test some correctness properties such as: *If there is an alive node in the network and there is space in the routing table, the node will be included in a certain period of time*. We are also interested in checking the interleaving of actions that take place at different points in time, and prove consistency properties like: *if two bins are consolidated, they do not split again until a certain period of time*.

As far as we know there is no other formal specification of the Kademlia and Kad routing tables. Right now the best sources to understand both protocol details are the original paper on the Kademlia DHT and the source code of the Kad implementation. Thus our first contribution is the benefits of having a formal specification of a system that is being consulted by many developers. In addition, the formalization of the routing tables allows us to compare the original version of Kademlia with the real implementation made in Kad.

However, our main contribution is the integration of the dynamic aspects of the routing table in the full protocol specification. The specification includes the ability of the routing tables to send and receive messages autonomously from the node by using different levels of configurations. Detection of non-answered messages is done by assigning a timeout when sending the message and triggering the appropriate action when the time expires. The actions that are performed automatically in Kad periodically, like populate almost empty buckets or remove off line contacts from the buckets, are triggered when their time expires. These actions require having a notion of time defined in different parts of the routing table and allow us to study the interleaving of actions that take place at different points in time.

2. Kademlia node IDs and contacts

2.1. Kademlia node IDs.

The Kademlia DHT assigns bit string quantities both to the node IDs and to the file keys. This simplifies the assignment of key values to nodes.

Kademlia node IDs are 160-bit quantities obtained for example from the SHA-1 hash algorithm.

First we define the sort `Bit` with two constant values `0` and `1`.

```
sort Bit .
op 0 : -> Bit [ctor] .
op 1 : -> Bit [ctor] .
```

We have two operations for the `Bit` sort:

- `equal : Bit Bit -> Bool .`, checks if two bits are the same,
- `XORd : Bit Bit -> Bit .`, computes the XOR operand of two bits.

A bit string is defined as a sequence of bits. It cannot be empty.

```
sort BitString .
subsort Bit < BitString .
op _;_ : BitString Bit -> BitString [ctor] .
```

We specify node IDs as subsorts of arbitrary bit strings with the required length. Namely, the 160-bit ID is of sort BitString160.

```
sort BitString160 .
subsort Bit <  BitString160 < BitString .

var S : BitString .
cmb S : BitString160 if length(S) == 160 .
```

We define the following operations over the bit strings:

- `op first : BitString -> Bit .`, returns the left most bit of the bit string.
- `op NBit : BitString NzNat -> Bit .`, returns the n-th bit counted from the left of the string. It starts counting 1. It is not defined for values of n greater than the length of the string.
- `op getPrefix : BitString NzNat -> BitString .`, returns the first n bits from the left hand side of the string. If the bit string has less than n bits the complete string is returned.
- `op fixNBits : BitString NzNat BitString -> Bool .`, checks if the left most $n-1$ bits of both bit strings coincide. If any of the bit strings has less than $n-1$ bits, it returns false.
- `op addFirst : Bit BitString -> BitString .`, adds the bit at the left hand side of the bit string.
- `op remFirst : BitString -> BitString .`, removes the first left hand side bit of a string. It is not defined on strings of length less than 1.

- `op distance : BitString BitString -> Nat .`, returns the distance between two bit strings. In particular, Kademlia defines the distance between two IDs as the bitwise exclusive (XOR) of the n-bit quantities interpreted as an integer.
- `op length : BitString -> Nat .`, returns the number of bits of a bit string.
- `op equal : BitString BitString -> Bool .`, checks if the two bit strings are the same.
- `op get-ID : BitString -> BitString .`, returns the bit string.
- `op compN : BitString NzNat -> BitString .`, changes the nth bit of the bit string into its opposite. It is not defined if the length of the bit string is less than n.
- `op numBitStr : NzNat -> NzNat .`, numBitStr(Nz1) returns the number of binary words (bit strings) of length Nz1.
- `op suc : BitString -> BitString .`, returns the next binary word (bit string) in lexicographical order.
- `op bWord : NzNat NzNat -> BitString .`, bWord(Nz1,Nz2) returns the Nz2-th binary word of length Nz1 in lexicographical order.

2.2. Kademlia contacts.

Kademlia contacts keep information of the IP address, the UDP port and de node ID, and are defined as triples of $<$ IP address, UDP port, Node ID $>$. We abstract from the concrete Kademlia contacts and allow simpler representations, via a parametric specification, although they should have a similar bit string representation for the node ID and the same notion of distance (XOR). In particular the bit string representation should not be empty, and should have a fixed number of bits so that all the contacts have the same length.

We represent the contacts by the sort `X$Contact`, given by the parameter definition. The defined operations over them are:

- `distance : X$Contact X$Contact -> Nat .`, which returns the XOR distance between two contacts,
- `NBit : X$Contact NzNat -> Bit .`, that returns the nth bit of the contact ID,
- `fix-contact? : X$Contact NzNat BitString -> Bool .`, checks if the first $n-1$ bits of the contact ID coincide with the first $n-1$ bits of the bit string. If the contact ID or the bit string have less than $n-1$ bits, it returns false,
- `getPrefix : X$Contact NzNat -> BitString .`, returns the first n bits of the contact ID. If the contact ID has less than n bits, it returns the complete contact ID,
- `get-ID : X$Contact -> BitString .`, returns the contact ID,
- `length : X$Contact -> NzNat .`, returns the length of the bi tstring that represents the contact ID. It cannot be empty,
- `equal : X$Contact X$Contact -> Bool .`, checks if the two contacts are the same.

3. Kademlia routing table

Each node stores contact information about others in what is called its *routing table*. In Kademlia, every node keeps a list of: IP address, UDP port, and node ID, for nodes of distance between 2^i and 2^{i+1} from itself, for $i = 0, \ldots, n$ and n the ID length. Note that each list contains IDs that differ in the *ith* bit. These lists, called *k-buckets*, have at most k elements, where k is chosen such that any given k nodes are very unlikely to leave the network within an hour of each other.

A Kademlia basic routing table is a proper binary tree whose leaves are k-buckets. But it is not a complete binary tree, where all leaves have the same depth. Each k-bucket is identified by the common prefix of the IDs it contains, it is responsible for a range of the node ID space and together the k-buckets cover the entire ID space with no overlap.

In fact, Kademlia routing table is a list of k- buckets. The bucket in the first level contains contacts whose first bit is the opposite to the first bit of the owner of the routing table, while the ID of the owner is in the range of the last bucket. Kademlia uses, when add a contact to the routing table, a tighter splitting rule where it is only allowed to split the bucket with common prefix equal to the one of the routing table owner. To insert a new contact in the appropriate k-bucket, the search runs along the routing table according to the common prefix between the contact ID and the owner ID. When the appropriate k-bucket is reached, there are the following possibilities:

- If the contact already exists in the leaf, it is moved to the tail of the k-bucket.
- If the contact is not already in the leaft and there is free space, it is inserted at the tail of the k-bucket.
- If the contact is not already in the leaf and the bucket has not free space, the node at the head of the list is contacted and there are three possibilities:
  - If it fails to respond it is removed from the list and the new contact is added at the tail.
  - In the case the node at the head of the list responds, if the bucket can be divided, it is split in two new buckets, the old contents divided between the two, and the isertion attempt repeated.
  - In the case the node at the head of the list responds, if the bucket cannot be divided the first contact is moved to the tail, and the new node is discarded.

    The only bucket that can be split is the only one whose range includes the owner ID of the routing table. So, the binary tree in the basic version of Kademlia is in fact a list of buckets. Figure 1 shows a routing table for node 01010000. In fact it could be the routing table of any peer with prefix `0101`, since the last bucket is not divided any more.

Kademlia does not have explicit operations to maintain the information of the routing table, checking for off line contacts only when inserting into full buckets. When a node receives any message (request or reply) from another node, it updates the appropriate bucket for the sender's node ID.

3.1. k-bucket.

The k-bucket is specified as a special sort of FIFO queue of non-repeated contacts. They may be empty. The order of the contacts in the bucket is very important. It is sorted by time last-seen-least-recently-seen node at the head, most-recently seen at the tail. A k-bucket has at most k elements, where k is chosen such that any given k nodes are very unlikely to leave the network within an hour of each other. Kademlia recommends K = 20 as suggested in the original version.

Buckets have the following policy to receive contacts:

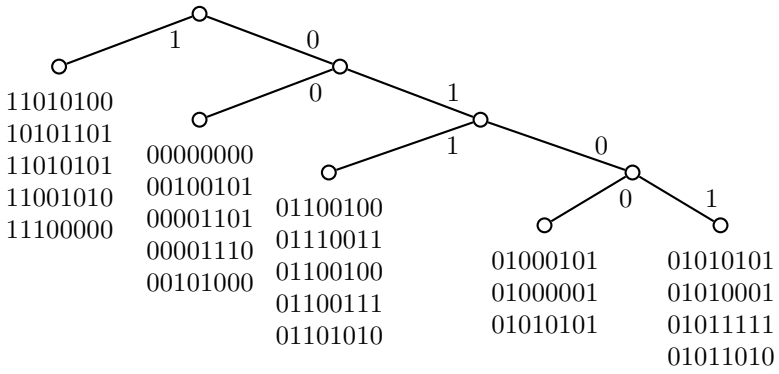


Figura 1: Routing table for node `01010000`

- If the contact to add already exists in the bucket, it is moved to the tail of the list.
- If the contact is not already in the bucket and the bucket has fewer than k entries, then the new contact is just inserted at the tail of the bucket.
- If the bucket is full and the contact at the head's bucket stayed online, it is moved to the tail and the new contact is discarded.

  So, buckets effectively implement a least-recently seen eviction policy, except that live nodes are never removed from the bucket.

K-buckets are represented by the sort `Bucket{X}`, where the parameter represents the contacts that are stored in the bucket . We differentiate the non-empty buckets with the subsort `NeBucket{X}.`

```
sort NeBucket{X} .
sort Bucket{X} .
subsort NeBucket{X} < Bucket{X} .
```

Terms are obtained with two constructors:

- `op empty-bucket : -> Bucket{X} [ctor]` , represents an empty bucket,
- `_!_ : Bucket{X} X$Contact -> [Bucket{X}] [ctor]` , represents the structure of a bucket as a list of contacts. It may generate terms that are not of sort `Bucket{X}`, since it may have repeated contacts or more than the allowed number of contacts. This fact is specified by writting the result of the operation in the *Kind* represented as the sort between square brackets. The terms generated by this constructor that are of sort `NeBucket{X}` are specified by a membership axiom.

The membership axiom :

```
var T : X$Contact .
var B : Bucket{X} .
cmb B ! T : NeBucket{X} if length-b(B) < bucketDim /\ not T in B .,
```

states that a bucket cannot have more than a certain number of contacts given by the constant `bucketDim` and that a bucket does not have repeated contacts.

For example, suppose contacts are 8-bit strings and k-buckets can hold up to 10 contacts, then the term:

```
empty-bucket ! 00110011 ! 01010101 ! 10101010 ! 11110000
```

represents a bucket, but

```
empty-bucket ! 00110011 ! 01010101 ! 10101010 ! 11110000 ! 11100011 !
   00011100 ! 00100100 ! 01011010 ! 00110011 ! 01010101 ! 11101110
```

does not, since it has more than ten contacts and it has repeated ones.

Normally, k-buckets will hold contacts with the same prefix, but the prefix can be empty and the contacts be anyone if the routing table has only one bucket.

```
empty-bucket ! 11001010 ! 11101010 ! 11001110 ! 11110111 ! 11111111 !
        11011111 ! 11000000 ! 11110000 ! 11001100 ! 11100011
```

is a bucket with prefix `11`.

We declare the following operations over the k-bucket:

- `op move-tail-b : X$Contact Bucket{X} -> Bucket{X} .`, moves a contact to the tail of the bucket. It has no effect if the contact is not in the bucket.
- `op add-contact : X$Contact Bucket{X} Bool -> Bucket{X }.`, adds a given contact to the bucket. If the contact is already in the bucket it moves the contact to the bucket tail. If the contact is not in the bucket and the bucket is not full, it adds the contact at the bucket tail. If the bucket is full and the contact is not in the bucket, but the first contact in the bucket is still alive then the first contact of the bucket is moved to the tail and the new contact is discarded. And, if the bucket is full and the contact is not in the bucket, but the first contact in the bucket is not alive then the first contact is discarded and the new contact is added to the tail of the bucket. The third parameter indicates if the first contact in the bucket is still alive (`on` = *true*) or not.
- `op rem-contact-b : X$Contact Bucket{X} -> Bucket{X} .`, removes a given contact from the bucket. It has no effect if the contact is not in the bucket.
- `op _in_ : X$Contact Bucket{X} -> Bool .`, checks if a contact is in a bucket .
- `op first-contact : NeBucket{X} -> X$Contact .`, returns the first contact of a bucket. It is not defined for the empty bucket.
- `op N-contact : NeBucket{X} NzNat -> X$Contact .`, returns the nth contact of a bucket. It is not defined for the empty bucket nor for an n greater than the number of contacts in the bucket.
- `op length-b : Bucket{X} -> Nat .`, returns the number of contacts in a bucket. It is zero if the bucket is empty.
- `op closestN : X$Contact Bucket{X} NzNat -> Set{vCONTACT}{X} .`, returns a list with the `n` closest nodes to the contact, stored in the bucket. Since the order of the returned contacts is not important it is defined as a set of contacts. It is not allowed to ask for zero contacts. If `n` is greater than the number of contacts in the bucket it returns all contacts. If the bucket is empty, returns the empty set.

- `op fix-bucket? : Bucket{X} NzNat BitString -> Bool .`, checks if all the contacts in the bucket have the same prefix of length $n-1$. The prefix should be the first $n-1$ bits of the given bit string and the n bit should be different from the one in the given bit string. The prefix could not be empty, since bit strings are not empty.
- `op fix-last-bucket? : Bucket{X} NzNat BitString -> Bool .`, similar to the previous one, checks if all the contacts in the bucket have the same prefix, but in this case, the prefix should be the first n bits of the given bit string. As previously the prefix could not be empty.
- `op full-bucket? : Bucket{X} -> Bool .`, checks if a bucket has the maximum allowed number of contacts, which is given in the specification by the constant `bucketDim`. The number of contacts in the bucket cannot be greater than the `bucketDim` constant because the operation is applied to values of sort `Bucket{X}`
- `op empty-bucket? : Bucket{X} -> Bool .`, checks if a bucket is empty.
- `op nearest-b : X$Contact NeBucket{X} -> X$Contact .`, returns the closest contact in the bucket to the given contact. It is not defined on empty-buckets.
- `op distance-min-b : X$Contact NeBucket{X} -> Nat .`, returns the distance of the given contact to the closest contact in the bucket. It is not defined on empty-buckets.
- `op equal : Bucket{X} Bucket{X} -> Bool .`, checks if two buckets have the same contacts, in the same order.

3.2. Routing table.

A routing table is a binary tree whose leaves are k-buckets. Each k-bucket contains contacts with some common prefix of their IDs. The prefix gives the k-bucket's position in the binary tree. Thus, each k-bucket covers some range of the ID space, and together the k-buckets cover the entire ID space with no overlap. If the routing table has more than one k-bucket, then it has one k-bucket by level, except at last level where there are two. The k-bucket in the first level contains contacts whose first bit is the opposite to the first bit of the owner of the routing table. The ID of the owner of the routing table is in the range of the last bucket (the right most k-bucket in the last level) but it is not inserted into it. In fact, Kademlia routing table is a list of k- buckets.

There are not empty routing tables. They have at least one non-empty bucket, although they can have empty buckets. Below we prove that one of the last bucket or the next to the last bucket could not be empty.

Routing tables are defined as lists of buckets. We do not consider in this version of the specification the optimizations proposed in [4] that allow keeping more contacts in the routing table, therefore there is no need for a tree of buckets.

```
sort RoutingTable{X} .
subsort NeBucket{X} < RoutingTable{X} .
```

It is declared one constructor:
`op _!!_ : Bucket{X} [RoutingTable{X}] -> [RoutingTable{X}] [ctor] .`.

The second parameter is in the `RoutingTable{X}` kind because it has the structure of a routing table, but it may not be complete nor have the correct contacts in the buckets. For example, its first bucket may not have contacts complementary to the first bit of the routing table owner. Notice that if the

constructed term is of sort `RoutingTable{X}`, then the second parameter of the constructor could not be of sort `RoutingTable{X}`, since it could not be complete.

A single non-empty bucket is a routing table by the previous subsort relationship.

Normally, k-buckets will hold contacts with the same prefix, where the prefix depends on the position of the bucket in the routing table, but the prefix can be empty and the contacts be anyone, if for example the routing table consist on only one bucket.

The following membership axiom sets when a list with more than one bucket is a routing table. It asks for all the buckets to have the appropriate contacts by means of the `is-RT` operation, and it checks whether the table does not exceed the maximum size (e.g.the number of buckets is not greater than the contacts ID length), and that at least one of the two last buckets is not empty. The contacts in a bucket should all have the same prefix. The prefix depends on the position of the bucket in the routing table, and the times that the last bucket has been divided, which is the number of buckets in the routing table minus one. Notice that we are not verifying the routing table of a concrete peer ID, but if a given term represents a possible routing table. This means that we are checking if the buckets have the appropriate contacts for a routing table with respect to the prefix of the last bucket of the table.

```
var KR : [RoutingTable{X}] .
vars B B1 B2 : Bucket{X} .

cmb KR : RoutingTable{X} if num-buckets(KR) > 1 /\
               num-buckets(KR) <= length(give-contact(KR))  /\
               atLeastOne?(KR) /\ is-RT(KR,1,peer-prefix(KR)) .
```

where

```
op is-RT : [RoutingTable{X}] NzNat BitString -> Bool .
eq [is-RT1] : is-RT(B1 !! B2,Nz,prefix) =
       fix-bucket?(B1,Nz,prefix) and fix-last-bucket?(B2,Nz,prefix)
       and (not empty-bucket?(B1) or not empty-bucket?(B2)) .
 eq [is-RT2] : is-RT(B1 !! (B2 !! KR),Nz,prefix) =
        fix-bucket?(B1,Nz,prefix) and is-RT(B2 !! KR,s Nz,prefix) .
 eq [is-RT3] : is-RT(B,Nz,prefix) = not empty-bucket?(B) .
```

We use the following private operations to define the membership axiom. These operations are defined in the Kind `[RoutingTable{X}]`, which means that they can be applied on routing tables that are not defined properly.

- `op peer-prefix : [RoutingTable{X}] -> BitString .`, a contact ID prefix of the owner of the routing table can be computed, since the contacts in the last bucket should have the same prefix. The length of the peer ID prefix is the number of buckets in the routing table minus 1. The operation is only defined on routing tables with more than one bucket. If the routing table has only one bucket, all the contacts are in it and there is no common prefix. If the routing table has more than one bucket, the peer ID prefix is the corresponding to the last bucket in the routing table. So, all contacts in the last bucket should have the same prefix of length equal to the routing table height (number of buckets in the routing table minus 1). Note that the owner ID of the routing table is in the range of the last bucket, although it is not inserted into it.
- `op num-buckets : [RoutingTable{X}] -> NzNat .`, computes the number of buckets of a routing table. The operation cannot return zero buckets, since the routing table cannot be empty.

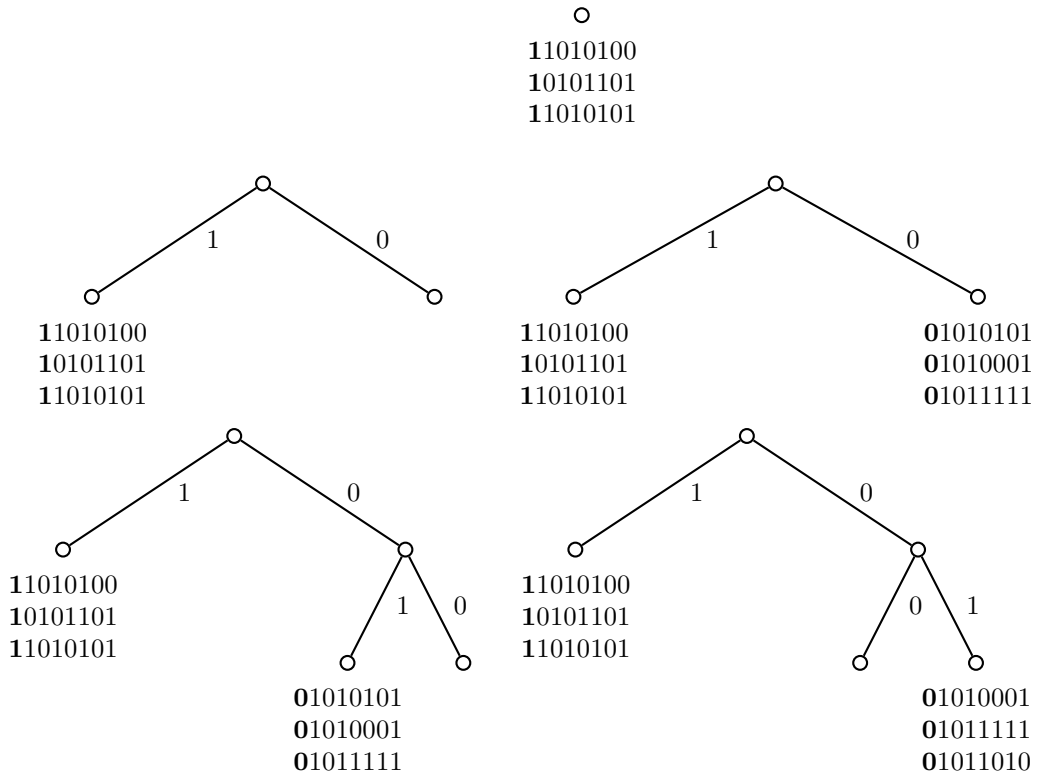


Figura 2: Nodes in the rouing table are allocated dynamically.

- `op give-contact : [RoutingTable{X}] -> X$Contact .`, returns the first contact of the last bucket if it is not empty. In other case, returns the first contact of the next- last bucket. One of the last bucket or the next to the last bucket could not be empty.
- `op last-bucket : [RoutingTable{X}] -> Bucket{X} .`, obtains the last bucket of the routing table, that is the bucket with the contact ID prefix of the owner of the routing table.
- `op atLeastOne? : [RoutingTable{X}] -> Bool .`, checks if the last or the next to the last bucket is not empty. The operation is only defined on routing tables with more than one bucket.
- `op Nbucket : [RoutingTable{X}] NzNat -> Bucket{X} .`, obtains the n^{th} bucket of a routing table. It is not defined for a number greater than the number of buckets.

Contacts in Kademlia are added to the routing table, but they are never explicitly removed. That is, a contact is only removed from the routing table when it is not online, it is the first contact of a full bucket, and there is a contact to be added to that bucket. Due to the manner in which the buckets are divided,the consequence is that only one among the last bucket and the other bucket at last level can be empty. This is because the only bucket that can be split is the last bucket, and it is split in two new buckets, the old contents divided between the two.

In the Figure 2 we suppose a bucket length of three, we suppose contacts are 8-bit strings and that the first bit of the owner ID of the routing table is 0. First, we suppose that the routing table has a single

node, that it is a full k-bucket that contains three contacts with first bit equal 1. If a new contact with a first bit equal 1 tries to be inserted and the first contact in the bucket is online, the bucket is divided into two buckets, the first bucket, with prefix 1, will be full and the second one, with prefix 0, will be empty. Then, if the first contact in the first bucket is still online, the new contact with first bit equal 1 is discarded, since only the last bucket can be divided. Then, suppose three new contacts arrive with prefix `01`, and are added to the last bucket. When a new contact with prefix 0 tries to be inserted, if the first contact of the last bucket is online, the last bucket is divided. If the new contact has also prefix `01` (e.g. its ID is `01011010`), then, if the owner ID has prefix `00` the last bucket will be empty, but if the owner ID has prefix `01`, and the first contact of the last bucket is now off line, it is removed from the bucket, the new contact is added at the tail of the bucket, and the next to last bucket will be empty.

We specify the following operations:

- `op add-entry : X$Contact RoutingTable{X} X$Contact -> RoutingTable{X} .`, adds a contact to the routing table. The third parameter is the owner of the routing table.
- `op closest-nodes : X$Contact RoutingTable{X} NzNat -> Set{vCONTACT}{X} .`, obtains the list of the n closest nodes to the given contact, stored in the routing table. The original paper on the Kademlia DHT [4] specifies *it takes α nodes from its closest non-empty k-bucket, or if that bucket has fewer than α entries, it just takes the α closest nodes it knows of.* It is not written that it takes the closest nodes in the selected bucket, nor which ones does it select. Since the Maude specification should be deterministic, if the bucket has at least α entries, we select the α closest nodes to the given contact, in the selected bucket, otherwise we select the α closest nodes in the routing table. It is not allowed to ask for zero nodes.

We use the following private operations:

- `op closest-nodes-aux : X$Contact RoutingTable{X} NzNat -> Set{vCONTACT}{X} .`, it is an auxiliary operation for `op closest-nodes` that selects the α closest nodes to the given contact, in the routing table, not only in the bucket that stores the given contact.
- `op rem-contact-rt : X$Contact [RoutingTable{X}] -> RoutingTable{X} .`, removes a contact from the routing table. It has no effect if the contact is not in the routing table.
- `op div-bucket : Bucket{X} NzNat Bit  -> [RoutingTable{X}] .`, generates a routing table by splitting the bucket into two new buckets, the old contents divide between the two. The last bucket contains the old contacts whose n-th bit is the same to the one of the owner of the routing table. The third parameter is the n-th bit of the owner ID of the routing table.
- `op goTo : Bucket{X} NzNat Bit NzNat -> [RoutingTable{X}] .`, it is an auxiliary operation for `op div-bucket . goTo(B,Nz,bi,Nzi)` sends the Nzi-th contact of `B` to the new last bucket if the Nz-th bit of its ID is equal to `bi,` or to the new next-last-bucket in other case. The third parameter (`bi`) is the Nz-th bit of the owner ID of the routing table.
- `op next-last-bucket : [RoutingTable{X}] -> Bucket{X} .`, returns the next to last bucket of the routing table. It is not defined on routing tables with less than two buckets.
- `op isLastBucket? : Bucket{X} [RoutingTable{X}] -> Bool .`, checks if the given bucket is the last one in the routing table.

- `op conc : [RoutingTable{X}] [RoutingTable{X}] -> [RoutingTable{X}] .`, concatenates lists of buckets by adding the second one in the place of the last bucket of the first routing table.
- `op find-bucket : X$Contact RoutingTable{X} NzNat -> Bucket{X} .` , finds the appropriate bucket for a contact. The last parameter indicates the level of the routing table that is been checked for the contact. It should be one when the function is called. The operation is not defined for values greater than the number of buckets in the routing table.
- `op add-entry-aux : X$Contact [RoutingTable{X}] NzNat X$Contact Bool -> RoutingTable{X} .`, finds the appropriate bucket for insert a contact, and adds the new contact according with the rules indicated in the beginning of this section. To insert a new contact in the appropriate k-bucket, the search runs along the routing table according to the common prefix between the contact ID and the owner ID. The third parameter indicates the level of the routing table that is been checked for the contact. It should be one when the function is called. The operation is not defined for values greater than the number of buckets in the routing table. Kademlia uses, when add a contact to the routing table, a tighter splitting rule where it is only allowed to split the bucket with common prefix equal to the one of the routing table owner. The fourth parameter is the owner of the routing table.The last parameter indicates if the bucket is not full or if it is full but the first contact in the bucket is still alive.
- `op add-entry2 : X$Contact RoutingTable{X} X$Contact -> RoutingTable{X} .`, it is an auxiliary operation of the `add-entry` operation, necessary because the `add-entry` operation of the routing tables is defined by rewrite rules since it changes the state of the routing table configuration.
- `op add-buckets : [RoutingTable{X}] NzNat NzNat -> [RoutingTable{X}] .`,

  add-buckets(KR,Nz1,Nz2) generates a routing table with the Nz1-th bucket of the routing table given as argument, as the leaf at level 1 in the new routing table, the Nz1+1-th bucket as leaf at level 2, and so on... if $Nz1 < Nz2 \leq$ number of buckets in the routing table given as argument. In other case, generates a routing table with a single bucket equal to the Nz1-th bucket of the routing table given as argument.
- `op num-contacts-table : [RoutingTable{X}] -> NzNat .`, returns the total number of contacts in the routing table.
- `op nearest-rt : X$Contact RoutingTable{X} -> X$Contact .`, returns the nearest contact to the contact given, stored in the routing table.
- `op distance-min-rt : X$Contact RoutingTable{X} -> Nat .`, returns the distance between the contact given and its nearest contact, stored in the routing table.

Time aspects of the routing table In Maude, a *configuration* is a soup of *objects* and *messages*.

```
sorts Object Msg Configuration .
subsort Object Msg < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration [ctor config assoc comm id: none] .
```

Objects can be declared using an object oriented syntax with the operation

```
op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .,
```

where `Oid` is an object identifier, `Cid` is a class identifier and `AttributeSet` is a set of attributes. Since we are using the full-maude system, we use the `CONFIGURATION` module declared in the `full-maude.maude` file [2, chapter 19].

We represent the P2P network as a Maude *configuration*, where peers are *objects* and the protocol messages are *messages*.

As part of the P2P network specification we declare in the `KADEMLIA-PROTOCOL` module the protocol messages.

The operation that adds a contact to the routing table checks, when the bucket is full, if the first contact is still alive by sending it a `PING` message.

```
msg PING : X$Contact X$Contact Time Time -> Msg .
msg PING-REPLY : X$Contact X$Contact Time Time -> Msg .
```

where the first parameter is the peer that sends the message; the second parameter is the peer that receives the message; the third parameter is the time to remove the message from the *soup* and the last one, the time that passes when an RPC is send. It is set to 1, for our proofs.

The time constant `RPCRemove` defines the time to remove a message because it is not answered.

```
op RPCRemove : -> Time .
eq RPCRemove = 20 .
```

We only present here the part of the protocol that is used in the routing table specification.

The `KADEMLIA-SYSTEM-SIMPL` module defines the peers, and the routing tables in them. It is part of the system specification. We define an attribute `RT` of sort `Configuration` in the peer objects that represents the routing tables. In this way the routing tables may send and receive messages by means of the two rewrite rules of the module. Notice that the reply is captured by the routing table only when a message to that object has been sent before. As before we present only the part related to the routing table.

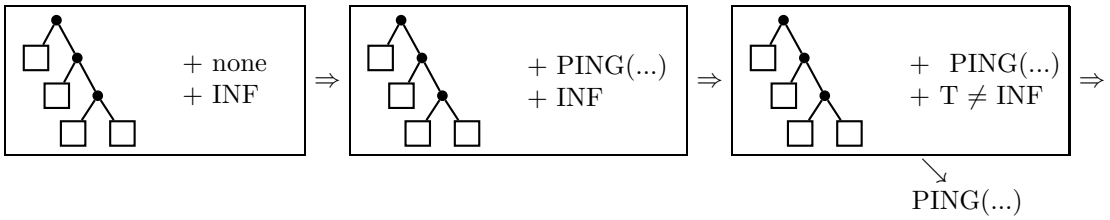
```
class Peer | RT : Configuration .
subsort X$Contact < Oid .

var R : RoutingTable{X} .
var P : Oid .
vars Z1 Z2 : X$Contact .
vars T1 T2 T3 T4 : Time .
var T : TimeInf .

crl [RT-receive] : PING-REPLY(Z1,Z2,T1,T2)
  < P : Peer | RT : ( R + PING(Z2,Z1,T3,T4) + T ) >
=>
  < P : Peer | RT : ( R + PING-REPLY(Z1,Z2,T1,T2) + T ) >  if T =/= INF .
rl [RT-send] : < P : Peer | RT : ( R + PING(Z1,Z2,T1,T2) + INF ) >
=>
 PING(Z1,Z2,T1,T2) < P : Peer | RT : ( R + PING(Z1,Z2,T1,T2) + T1 )  > .
```

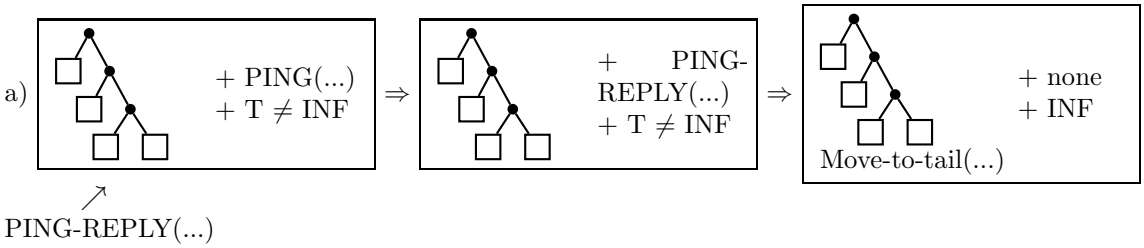
In the `ROUTING-TABLE` module we define *configurations* as routing tables plus a possible message plus the time to wait for the message reply. The `add-entry` operation of the routing tables is defined by rewrite rules since it changes the state of the routing table configuration.

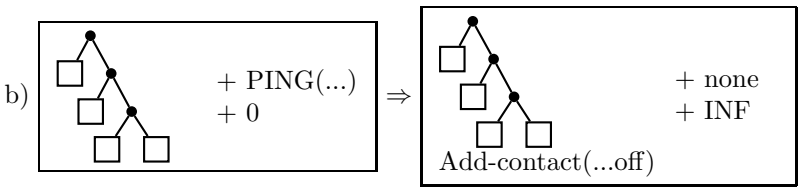
The `add-entry` operation receives the contact to be added, the routing table and the contact of the routing table owner. The behaviour is as follows: equation `add0` deals with the case in which the bucket is not full and the new entry is added at the tail. If the bucket is full, by rule `add1` the routing table puts a `PING` message in the `RT` configuration, sets the time to the `PT` constant value, which represents the maximum time it will wait for a reply before consider the contact off line, and calls the auxiliary function `add-entry2` on the routing table. If the first contact is off line, the time value of the configuration will raise zero and rule `add2` will remove the off line contact and will add the new contact at the tail of the bucket by means of the `add-entry-aux` operation. Besides the message is removed from the configuration, and the time is set to the `INF` value. If the first contact replies, and the bucket is full and it cannot be divided (rule `add3`), the `add-entry-aux` operation moves the first contact to the tail of the bucket and discards the new contact. Finally if the bucket can be divided (rule `add4`), the entry is inserted again after splitting the bucket.

```
op _+_+_ : RoutingTable{X} Msg TimeInf -> Configuration [ctor] .

 --- Bucket not full. New contact added at the tail
crl [add0] : add-entry(Z1,R,Z2) + none + T
=>
    add-entry-aux(Z1,R,1,Z2,true) + none + T
if not full-bucket?(find-bucket(Z1,R,1)) .

 --- Bucket full. Ask if first bucket contact is on
crl [add1] : add-entry(Z1,R,Z2) + none + INF
  =>
    add-entry2(Z1,R,Z2) + PING(Z2,first-contact(find-bucket(Z1,R,1)),RPCRemove,1) + 1
    if full-bucket?(find-bucket(Z1,R,1)) .

  --- First bucket contact off
rl [add2] : add-entry2(Z1,R,Z2) + M + 0
  =>
    add-entry-aux(Z1,R,1,Z2,false) + none + INF .
 --- First bucket contact on. Not split bucket
crl [add3] : add-entry2(Z1,R,Z2) + PING-REPLY(Z,Z2,T1,0) + T =>
             add-entry-aux(Z1,R,1,Z2,true) + none + INF
    if B1 := find-bucket(Z1,R,1) /\ equal(Z,first-contact(B1)) /\
       full-bucket?(B1) /\ not isLastBucket?(B1,R) /\ T1 > 0 /\ T > 0 .
  --- First bucket contact on. Split bucket
crl [add4] : add-entry2(Z1,R,Z2) + PING-REPLY(Z,Z2,T1,0) + T =>
   add-entry(Z1,conc(R,div-bucket(last-bucket(R),Nz,NBit(Z2,Nz))),Z2) + none + INF
    if B1 := find-bucket(Z1,R,1) /\ equal(Z,first-contact(B1)) /\
    Nz := num-buckets(R) /\
       full-bucket?(B1) /\ isLastBucket?(B1,R) /\ T1 > 0 /\ T > 0 .
```

In this version, the routing tables can send only one message at a time. They cannot send more messages until the first one has received a reply or the time to answer has expired. The following figure shows the sequence of routing table configuration states when a new contact is added to a full bucket.


+ none
+ INF
⇒
+ PING(...)
+ INF
⇒
+ PING(...)
+ T ≠ INF
⇒
PING(...)


a)
+ PING(...)
+ T ≠ INF
⇒
+ PING-
REPLY(...)
+ T ≠ INF
⇒
+ none
+ INF
Move-to-tail(...)
PING-REPLY(...)


b)
+ PING(...)
+ 0
⇒
+ none
+ INF
Add-contact(...off)

4. The Kad routing tables.

Kad routing tables differ from the Kademlia ones both in their structure and in their behaviour. In the structure they allow for more k-buckets, called now *routing bins*, to store more contact information in their routing tables. Regarding the behaviour, the table keeps its contacts updated by two processes that allow searching for new contacts and delete contacts that are no longer active, rather than update the table each time a message is received. A discrete notion of time is used to control the process activation.

The Kad routing table is also a binary tree. The main differences with the Kademlia binary tree are:

- The binary tree nodes are called *routing zones* . *Routing bins* are placed in the leaf nodes.
- Kad divides the table structure in levels and positions (*ZoneIndex*). The *ZoneIndex* represents the horizontal distance in number of routing zones from the leftmost one on the same level. Both, levels and *ZoneIndex*, start numbering at 0.
- Kad uses the XOR distance to the owner ID of the routing table, to store *Contacts* in routing bins. The routing bin, in which a specific contact is placed, depends on its distance to the owner of the table. Now, every bit in the distance corresponds to a level in the tree beginning with bit number 0 which corresponds to level 0. The more significant bit corresponds to the root zone.

To insert a new contact, the search runs along the routing table according to the contact's XOR distance to the owner ID. Therefore it checks the n-th bit of the distance, where n is the current level of the routing zone. If the value is 0 the contact belongs to the left subtree and if it is 1 it belongs to the right subtree. When the distance to a leaf is covered, there are the following possibilities:

- If the contact already exists in the leaf, which is a routing-bin, then the procedure is aborted.
- If not, there are the following possibilities:
  - The bin is not full, then the new contact is inserted in the tail of this bin. The order is important when contacts are removed.
  - The bin is full and the leaf level is less than 4, then the bin is split into two. The splitting of a leaf on level n will add two new leaves on level $n+1$, which become the children of the current leaf node in the tree. All the contacts from the old bin are moved to the two new bins and the new contact is inserted in the appropriate bin which may be split again if it is required and possible. All contacts whose distance to the routing table owner ID, expressed as a bit string, have bit 1 on position $n+1$ will be added to the right leaf and the others will be added to the left leaf. The old bin is no longer located on a leaf node and is therefore removed.
  - Otherwise, when the bin is full and the leaf level is greater than 3, the bin is split into two only if the value of the zone index is smaller than 5. Thus, in deeper levels only the first five bins on the left can be split.
  - If the tree reaches level 127 splitting is not required anymore since there could not be enough contacts in the bin to fill it.

Kad has a looser splitting rule than Kademlia It allows to split a bin if its level is smaller than 4 or its index is smaller than 5, which results in more possible contacts. Figure 3 (borrowed from [5]) shows a complete Kad routing table. KAD has actual maintenance tasks that run at set periods. Each peer has a creation and expiration time, and a type with scale from 0 to 4. Type 4 means that the contact is rarely connected and will probably be eliminated at the next occasion.

Kad defines three processes that are automatically executed by the routing table from time to time.

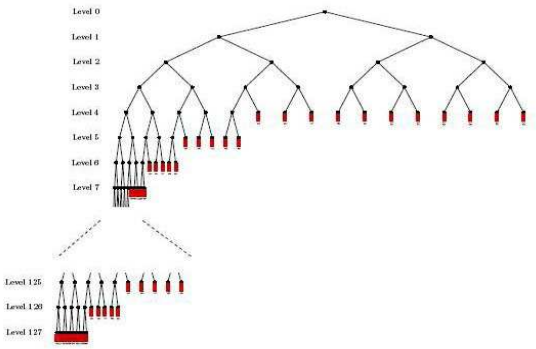


Figura 3: Complete Kad routing table.

- *Consolidate*: two adjacent bins in the same level and with few contacts are joined into one bin. The consolidation process runs every 45 minutes, and checks the whole routing tree. Since the process runs automatically every 45 minutes, we need the time since the last consolidation. When the operation joins two adjacent bins, the level, zone index and the time to remove off line contacts of the new bin is the one of its parent, while the time to populate the new bin is set to ten seconds. Since the remove time of the parent node is needed, we must keep it in the specification of the internal tree nodes, not only in the leaves.
- *Populate*: new contacts are added to the routing table bins if they can be split or if they have few contacts. It is done automatically for each bin once an hour. Since it is done independently for each bin, it is necessary to keep the time since the last operation for each bin.
- *Remove off line contacts*: the operation removes all contacts with type 4 in the bin. Then if the expiration time of the first contact is less than the current time, the routing table sends a HELLO-REQ message to check if the contact is still alive. At the same time it puts the expire time in two minutes and increases the type of the contact, so if the contact does not answer it will be removed the next time the operation is executed. This means that the constant `HELLO-TIMEOUT`, defined in the Kad implementation, is not needed for the routing table specification since no special action is taken when the message is not answered.

4.1. Kad constants.

The following constants, defined in the Kad implementation, are used in the specification. We use the Kad implementation identifiers whenever possible.

- Network structure parameters:
  - `K = 10 .`, maximum number of contacts in a bin.
  - `KBASE = 4 .`, maximum level for splitting.

- `KK = 5 .`, *Zone Index* allowed for splitting.

We use also, the following time constants. The basic time unit corresponds to a second in the real Kad implementation.

- Time parameters:
  - `POPULATE-TIME = 10 .`, time for looking for new contacts when a new bin has been created by the consolidation or the splitting processes. It is set to 10 seconds.
  - `POPULATE-TIME-BIG = 3600 .`, time for looking for new contacts in bins that can be split or in almost empty bins. It is set to one hour.
  - `CONSOLIDATE-TIME = 2700 .`, time for the next consolidation process. It is set to 45 minutes.
  - `OFFLINE-CHECK = 120 .`, time given to the first contact of a bin to answer a requirement of been alive before removing it from the bin. It is set to two seconds.
  - `NEXT-REM = 60 .`, time for removing off line contacts in a zone. It is set to one minute.
  - `EXPIRE-TIME-1 = 3600 .`, expire time of a contact of type 2. It is set to one hour.
  - `EXPIRE-TIME-15 = 5400 .`, expire time of a contact of type 1. It is set to one hour and a half.
  - `EXPIRE-TIME-2 = 7200 .`, expire time of a contact of type 0. It is set to two hours.

4.2. Kad contacts

Kad assigns 128-bit quantities, obtained from the MD4 hash algorithm, to the node IDs. This difference with regard to Kademlia, is due to the fact that the implementation of Kad by eMule had to be compatible with the previous protocol. Kad adds a lot of information to the contacts. We abstract from much of this information and we are left with the Kademlia information, the Kad type, the creation time and the expire time. There are five types of contacts in Kad, Type 0 are contacts known for more than two hours, and are rated as the most reliable ones. Type 1 contacts were added between one and two hours ago. Type 2 are known for less than an hour. Type 3 are newly inserted contacts, and type 4 are contacts that fail to respond for at least one time and are marked for deletion. The creation time and the expire time are specified as discrete time taken from the `NAT-TIME-DOMAIN-WITH-INF` module of the Real-Time-Maude system. The creation time holds the time that the node has been active, and the expire time is the time left to check if the node is still alive. The creation time increases as the time passes in the system, while the expire time decreases.

Kad contacts consist of 128-bit ID, an IP address, a UDP and TCP port, a type, a creation time and an expire time. We define sorts `IP`, `UDP`, and `Contact-types` for the IP address, the UDP and TCP port and the contact type respectively. The 128-bit ID is of sort `BitString128`, which is a subsort of arbitrary bit strings with the required length. The creation time and the expire time are of sort `Time`.

```
sort   BitString128 < BitString
var S : BitString .
cmb S : BitString128 if length(S) == 128 .
op c : BitString128 IP UDP Contact-types Time Time  -> Kad-Contact [ctor] .
```

We define the following operations over the Kad contacts:

- `op get-ID : Kad-Contact -> BitString128 .`, obtains the contact identifier.

- `op get-type : Kad-Contact -> Contact-types .`, obtains the contact type.
- `op get-CTime : Kad-Contact -> Time .`, obtains the contact creation time.
- `op get-ETime : Kad-Contact -> Time .`, obtains the contact expire time.
- `op incr-type : Kad-Contact -> Kad-Contact .`, increases the type of a contact. On contacts of type 4 it has no effect.
- `op set-ETime : Kad-Contact Time -> Kad-Contact .`, sets the expire time of a contact to the given value.
- `op update-Type : Kad-Contact -> Kad-Contact .`, updates the type of a contact depending on its expire time.
- `op NBit : Kad-Contact NzNat -> Bit .`, obtains the n bit of the contact identifier.
- `op fix-contact? : Kad-Contact NzNat BitString -> Bool .`, checks if the first left-hand n bits of contact identifier fix with the first left-hand n bits of the given bit string.
- `op equal : Kad-Contact Kad-Contact -> Bool .`, checks if the contact identifiers are equal.
- `op XORd : Kad-Contact Kad-Contact -> BitString128 .`, obtains the XOR metric of the two contact identifiers.
- `op new : Kad-Contact -> Kad-Contact .`, sets the type of a contact to 3 and the creation time and expire time to zero.

4.3. Routing bins in the Kad network.

Routing bins are a sort of K-bucket that store the *Contacts*.

4.3.1. The routing bin sort and constructors

Routing bins are specified as lists of non-repeated contacts. The maximum number of contacts in a bin is given by the constant `K`, which is set to 10 in the Kad implementation. Normally, contacts are retrieved from the front and inserted at the end. However, they can be removed from any position of the list, as it happens, for example, in the process that removes type 4 contacts.

Routing bins may be empty, since contacts can be removed from the bins when there is no insertion. However, the empty bins are joined by the *consolidation* process.

To insert a contact in a routing bin:

- If the contact already exists in the bin or the bin is full, it is not inserted nor moved, and the process is aborted.
- If the bin is not full the contact is inserted at the end of the bin.

Routing bins are represented by two sorts:

- `sort NeRoutingBin .` which represents non-empty routing bins and
- `sort RoutingBin .` which represents all the routing bins,

- with a subsort relation `subsort NeRoutingBin < RoutingBin .`.

Values are obtained with two constructors:

- `op empty-bin : -> RoutingBin [ctor] .`, represents an empty bin,
- `op _!_ : RoutingBin Kad-Contact -> [NeRoutingBin] [ctor] .`, represents the structure of a bin as a list of contacts. It may generate terms that are not of sort `RoutingBin`, since it may have repeated contacts or more than the allowed number of contacts. This fact is specified by writting the result of the operation in the *Kind* represented as the sort between square brackets. The terms generated by this constructor that are of sort `RoutingBin` are specified by a membership axiom:

  ```
  var B : RoutingBin . var T : Kad-Contact .
  cmb B ! T : NeRoutingBin if getSize(B) < K /\ not T in B .
  ```

  which states that a bin cannot have more than a certain number of contacts given by the constant `K` and that a bin does not have repeated contacts.

4.3.2. Routing bin functions

We declare the following operations:

- `op add-contact : Kad-Contact RoutingBin -> RoutingBin .`, adds a given contact to the tail of the bin. It has no effect if the contact is already in the bin or if the bin is full.
- `op rem-contact-b : Kad-Contact RoutingBin -> RoutingBin .`, removes a given contact from the bin. It has no effect if the contact is not in the bin.
- `op rem4 : RoutingBin -> RoutingBin .`, removes contacts of type 4 from the bin. It has no effect if there are no contacts of type 4 in the bin.
- `op pushToBottom : Kad-Contact RoutingBin -> RoutingBin .`, moves the contact at the end of the list if the contact is in the bin. In other case, if there is free space, the operation adds the contact at the end.
- `op _in_ : Kad-Contact RoutingBin -> Bool .`, checks if a contact is in a bin.
- `op first-contact : NeRoutingBin -> Kad-Contact .`, returns the first contact of a bin. It is not defined for the empty bin.
- `op getSize : RoutingBin -> Nat .`, returns the number of contacts in a bin. It is zero if the bin is empty.
- `op full-bin? : RoutingBin -> Bool .`, checks if a bin has the maximum number of contacts, given by the `K` constant.
- `op empty-bin? : RoutingBin -> Bool .`, checks if a bin is empty.
- `op fix-bin? : RoutingBin NeNat BitString -> Bool .`, checks if all the contacts in the bin have the same prefix. The prefix should be the first $n-1$ bits of the given bit string and the n bit should be different from the one in the given bit string. It is not allowed to ask for an empty prefix.
- `op fix-last-bin? : RoutingBin Nat BitString -> Bool .`, checks if all the contacts in the bin have the same prefix. The prefix should be the first n bits of the given bit string.

- `op add-bins : RoutingBin RoutingBin -> [RoutingBin] .`, adds contacs from the second bin to the tail of the first bin. The order of the contacts in the second bin is preserved. The result may not be of sort `RoutingBin`, since it can have more than the allowed number of contacts or they may be repeated.
- `op equal : RoutingBin RoutingBin -> Bool .`, checks if two bins have the same contacts, in the same order.
- `op incr-type : Kad-Contact RoutingBin -> RoutingBin .`, increases the type of a given contact in a bin. It has no effect if the contact is not in the bin or it has type 4.
- `op set-ETime : Kad-Contact RoutingBin Time -> RoutingBin .`, sets the expire time of a given contact in a bin. It has no effect if the contact is not in the bin.
- `op mod-contact : Kad-Contact RoutingBin -> RoutingBin .`, moves the contact to the bottom of the bin, and changes its type and expire time according to the time it has been alive, if the contact is in the bin. In other case, if there is free space, the operation adds the contact at the end.
- `op change-first : NeRoutingBin Time -> NeRoutingBin .`, increases the type and changes the expire time of the first contact in the bin.

4.4. The Kad routing tables.

Routing tables are specified as binary trees, represented by the root node together with the time to realize the consolidation process. Since we need the distance from the contact to the routing table owner to locate the contacts in the tree, information about the table owner is also kept in the routing table.

The time to realize the node processes is kept in the nodes.

There are not empty routing tables. They have at least one routing bin, although when the system is initialized, the routing bin is empty. So, there can be routing tables with no contacts.

4.4.1. The routing zone and routing table sorts

We have sorts to represent routing zones and the routing table:

```
sort RoutingZone .
sort RoutingTable .
```

A routing zone may be a leaf of the tree, in which case it is represented by a routing bin, its level and zone index and the time that rest to start the populate process and the remove process.

```
op rb : RoutingBin Nat Nat Time Time -> [RoutingZone] [ctor format (d d)] .
```

Internal nodes are represented by their two subtrees, given by the routing zones that represent them and the time that rest to start the populate precess and the remove process. Internal nodes do not execute neither the populate, nor the remove processes, but these times are used in the consolidation process to fix the times for the new bin. Notice that only the time to remove contacts is used, we maintain the time to populate for consistency with the real implementation.

```
op rz : [RoutingZone] [RoutingZone] Time Time -> [RoutingZone] [ctor format (d d)] .
```

Notice that, since not all valuesof the parameters produce *correct* routing zones, the result is in the *Kind*. The *correct* routing zones are defined below by means of two memberships.

Finally the routing table is defined by the root of the tree, given by a routing zone, the contact of the table owner and the time to start the consolidation process, which is done for the whole tree.

```
op rt : RoutingZone Kad-Contact Time -> RoutingTable [ctor format (d d)] .
```

The `RoutingZone` sort represents the complete routing tree. Subtrees are defined at the kind level. The following membership axiom

```
mb(rb(B,0,0,T1,T2)) : RoutingZone .
```

defines a routing zone as a tree with only one node, the root. The *Level* and *Zone Index* are equal to zero and the time for the population and remove proccesses may be anyone.

Routing zones with more than one node are defined by the following membership axiom:

```
cmb(rz(KR1,KR2,T1,T2)) : RoutingZone
 if is-subRT(0,0,rz(KR1,KR2,T1,T2)) /\ is-RT(rz(KR1,KR2,T1,T2)) .
```

where

- `op is-subRT : Nat Nat [RoutingZone] -> Bool .` checks if a routing zone has the tree structure required for a routing table.

```
eq [subRT1] : is-subRT(N1,N2,rb(B,N3,N4,T1,T2)) = (N1 == N3) and (N2 == N4) and
                        (N1 <= KBASE and N2 < 2 ^ N1 or
                        N1 > KBASE and N1 <= sd(128BIT,1) and N2 < 2 * KK) .
eq [subRT2] : is-subRT(N1,N2,rz(KR1,KR2,T1,T2)) =
                  is-subRT(s N1, 2 * N2,KR1) and is-subRT(s N1,2 * N2 + 1, KR2) .
```

- `op is-RT : [RoutingZone] -> Bool .` checks if the contents of the bins are correct with respect to the structure.

```
eq [isRT1] : is-RT(rb(B,0,N2,T1,T2)) = true .
eq [isRT2] : is-RT(rb(B,Nz1,N2,T1,T2)) = fix-last-bin?(B,Nz1,bWord(Nz1,s N2)) .
eq [isRT3] : is-RT(rz(KR1,KR2,T1,T2)) =
                  is-RT(KR1) and is-RT(KR2) .
```

For example, suppose we have node IDs of length 8, a routing table with only one bin, and three contacts in it. The routing table is represented by the term:

```
rt( rz(empty-bin !
         c(0 ; 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0,ip,udp,type0,40000,7200)!
         c(0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1,ip,udp,type2,5,3600) !
         c(1 ; 1 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1,ip,udp,type3,0,1), 0 ,0 ,600 ,20),
c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,14392,8) , 200)
```

where `c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,14392,8)` is the owner contact of the routing table. The time to populate is set to 600. The time to remove off line contacts is set to 20 and the time to consolidate the table is set to 200.

Other possible routing table of the same owner could be the following, which has one level and two bins, each one with two contacts:

```
rt(rz(rb(empty-bin !
         c(0 ; 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0,ip,udp,type0,40000,7200) !
         c(0 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1,ip,udp,type2,5,3600),1,0,3000,10),
       rb(empty-bin !
          c(1 ; 0 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1,ip,udp,type4,10,43) !
          c(1 ; 1 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1,ip,udp,type3,0,1) ,1 ,1,700,3), 500 ,30) ,
    c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,14392,8), 2500)
```

A routing table with seven bins and 26 contacts can be found in Anex A.

4.4.2. Routing zone functions

We declare the public operations:

- `op add-entry : Kad-Contact RoutingTable -> RoutingTable .`, adds a contact to a routing table. If the contact is already in the routing table or the routing bin to insert the contact is full and cannot be split the operation has no effect. If the routing bin to insert the contact is not full the contact is inserted at the bin's tail. In other case, the bin is divided in two new bins, and the contact is added to the new bins, following this insertion algorithm. The operation makes use of the `add-entry-aux` operation.
- `add-entry-hello : Kad-Contact RoutingTable -> RoutingTable .`, changes the type and remove time of a contact after receiving a `HELLO-RES` message. The operation makes use of the `add-hello-aux` operation.
- `op consolidate : [RoutingZone] -> RoutingZone .`, the consolidation process is defined recursively on the nodes of the routing tree. The consolidation process is done on adjacent bins that have the same parent routing zone and both bins together have less than half the maximum number of contacts allowed in a routing bin. The bins are removed and the parent routing zone is transformed into a routing bin. Contacts of the old bins are added to the new bin, first the contacts of the left bin are added from the head to the tail, and after, contacts of the right bin are added in the same way. Since contacts are added at the tail of the bin, the new bin has in its head the head contact of the left bin, and in its tail the tail contact of the right bin. The process is done only once from the the root to the leaves, consolided bins are not consolided again by the same process. The population time of the new bin is set to 10 seconds, represented by the constant `POPULATE-TIME`, and the time to remove off line contacts is the one of the parent routing zone.

We use the following private operations:

- `op add-entry-aux : Kad-Contact BitString [RoutingZone]  Kad-Contact -> RoutingZone .`, adds a contact to a routing zone. It is used by the `add-entry` operation. The parameters are:
  1. Contact to be added to the routing zone.
  2. Distance, expressed by the XORd operation, between the ID of the contact to be added and the ID of the routing table owner.

3. Routing zone of the routing table in which the contact is to be added. In the initial call the parameter will be of sort `RoutingZone`. Since the operation is defined recursively the parameter is in the *kind* to allow the traversal of the tree.
4. ID of the routing table owner.

- `op div-bin : Nat Nat RoutingBin Kad-Contact Time Time -> [RoutingZone] .`, splits a bin and distributes the contacts between the two new ones. The relative order of the contacts in the bin is not change. The parameters are:
  1. The routing bin *level*;
  2. The routing bin *zone index*;
  3. The routing bin to be split;
  4. The ID of the routing table owner. It is used to divide the bin when necessary;
  5. The time that rest to start the populate process;
  6. and the time to start the remove process of the bin. It is set to the time of the split bin, instead of the time used in the real implementation.
- `op add-hello-aux : Kad-Contact BitString [RoutingZone] -> RoutingZone .`, moves the contact to the bottom of its bin and changes its type and expire time after receiving a `HELLO-RES` message. If the contact does not exist in the bin, it is added at the bottom. The parameters are:
  1. Contact to be added to the routing zone;
  2. Distance, expressed by the XORd operation, between the ID of the contact to be added and the ID of the routing table owner;
  3. Routing zone of the routing table in which the contact is to be added.
- `op getNumContacts : [RoutingZone] -> Nat .`, obtains the number of contacts of all the bins in a routing zone. It can be zero because the routing bins can be empty.
- `op isLeaf? : [RoutingZone] -> Bool .`, checks if a routing zone is a leaf, that is a routing bin.
- `op getRZ : [RoutingZone] PairLevelZoneI -> [RoutingZone] .`, obtains the routing bin at a given level and zone index. It is not defined if the tree does not have a bin at the given level and zone index.
- `op getBin : [RoutingZone] -> RoutingBin .`, obtains the routing bin of a leaf routing zone. If the routing zone is not a leaf, the operation is not defined.
- `op getRT : [RoutingZone] -> Time .`, obtains the remove time of a leaf routing zone. If the routing zone is not a leaf, the operation is not defined.
- `op getPT : [RoutingZone] -> Time .`, obtains the population time of a leaf routing zone. If the routing zone is not a leaf, the operation is not defined.
- `op changePop : [RoutingZone] Nat Nat Time -> [RoutingZone] .`, changes the population time of the routing zone with the given level and zone index. If the given level and zone index do not correspond to a leaf node, the operation is not defined.

- `op timePop : RoutingZone Nat Nat -> Bool .`, checks if it is time to populate (the population time is zero) a routing zone with a given level and zone index. If the given level and zone index do not correspond to a leaf node, the operation is not defined.
- `op changeRem1 : RoutingZone Nat Nat -> RoutingZone .`, changes the remove time of the routing zone with a given level and zone index. It also pushes to the bottom of the bin the first contact and removes all type 4 contacts of the bin. If the given level and zone index do not correspond to a leaf node, the operation is not defined.
- `op changeRem2 : RoutingZone Nat Nat -> RoutingZone .`, changes the remove time of the routing zone with a given level and zone index. It also increases the type and changes the expiration time of the first contact in the bin. It also removes all type 4 contacts of the bin. If the given level and zone index do not correspond to a leaf node, the operation is not defined.
- `op timeRem : RoutingZone Nat Nat -> Nat .`, searchs for the routing zone of the given level and zone index. Returns 0 if the routing zone is a bin and the time to remove off line elements is different to zero. Returns 1 if the time to remove off line elements is zero and the expire time of the first non-type 4 contact of the bin is greater than zero. Returns 2 if the time to remove off line elements is zero and the expire time of the first non-type 4 contact of the bin is zero. If the given level and zone index do not correspond to a leaf node, the operation is not defined.
- `op first-contact-bucket : RoutingZone Nat Nat -> Kad-Contact .`, returns the first contact of the routing bin on the given level and zone index. If the given level and zone index do not correspond to a leaf node, the operation is not defined.

4.5. Kad nodes specification.

The routing table is related with the rest of the node processes, since it requires searching for nodes or the sending messages. In Kad, events are controlled by the event Map process which iterates over all the events, in particular those generated by the routing table.

We have developed a basic specification of the network nodes that contains the processes related with the routing table that allow us to execute the specification and proof properties. We do not specify the complete behaviour of the nodes, but only the part we need for our specification. We abstract the node behaviour when necessary. We use the object oriented syntax of full-maude.

A node is defined as an object of class `Peer`, with attributes:

- `CRoutingTable` defines the node routing table.
- `CSearchManager` a list of file keys the node has to searc for.
- `CMessages` a list with the messages the node has to send to other nodes.
- `CEventMap` a list of the routing bins in the routing table, to iterate over them to execute the processes.

The object ID is the node contact.

```
class Peer | CRoutingTable : RoutingTable,
             CSearchManager  : List{vKEY},
             CMessages : MsgList,
             CEventMap : EventMap .
subsort Kad-Contact < Oid .
```

There are only two relevant messages for the routing table.

- `msg HELLO-REQ : Kad-Contact Kad-Contact Time Time -> Msg .`
- `msg HELLO-RES : Kad-Contact Kad-Contact Time Time -> Msg .`

The parameters are:

1. Peer that sends the message;
2. Peer that receives the message;
3. Time to remove the message;
4. Time that passes when an RPC is sent. Set to 1.

4.6. Routing table processes.

Kad has three routing table processes: The *populate* process, the *remove off line nodes* process, and the *consolidation* process. The two first ones are realized on the routing bins, the last one is realized in the whole routing tree.

The routing table processes are realized in the Kad implementation using an iterator over the events generated by the nodes. Only once all the node events have been done, the consolidation process is taken.

In the node specification we declare an attribute, called `CEventMap`, that has the list of all the routing bins in the routing tree. The routing bins are represented in this list by pairs of *Level* and *Zone Index*. The list has an additional value, `< 0 , 0 >`, which represents the consolidation process. This representation allows us to realize all the routing bin processes and later on the consolidation process. This event map simulates the iterator implemented in the real Kad network.

4.6.1. Consolidation process.

The *consolidation process* takes place when its time, represented by the last parameter of the routing table, expires. The rule that controls the process is:

```
rl [consolidate1] : < Z : Peer | CRoutingTable : rt(R,Z2,0) ,
               CEventMap : < 0 ; 0 > EM , CSearchManager : L , CMessages : LM >
 =>
  < Z : Peer | CRoutingTable : rt(consolidate(R),Z2,CONSOLIDATE-TIME) ,
               CEventMap : EM < 0 ; 0 > , CSearchManager : L , CMessages : LM > .
```

It is executed when the first item of the `CEventMap` list is the flag value `< 0 ; 0 >`. It takes the routing table, executes the consolidation proccess in its routing zone and sets the new consolidation time. The flag value of the `CEventMap` list is put at the tail of the list.

Consolidation of the routing zone is done with the `consolidation` function... The process consolidates almost empty bins, that have the same parent routing zone.The consolidation is done if they both together have less than 5 contacts. The consolidation process runs only every 45 minutes, but it checks the whole routing tree. It adds first the contacts of the 0 child and after that the contacts of the 1 child. Contacts of each bin are added first the oldest and at the end the tail of the list. The process is done only once from the the root to the leaves, consolided bins are not consolided again by the same process. The time to populate the new bin is set to the constant `POPULATE-TIME` and the time to remove contacts is set to the time stated in the parent routing zone.

4.6.2. Population process.

The *population process* takes place when the bin can be split or the number of contacts in the bin is equal or less than 20 % of the maximum allowed contacts in a bin. The rule is executed on the bin represented by the first item of the `CEventMap` list if its time to populate is zero.

The process generates a random ID in the bin and starts a search for that ID. It modifies the search manager and appends the new search. Contacts are then added to the routing table by the search process. The routing table changes the time for population of that bin and the first item of the `CEventMap` list is moved to the tail.

```
crl [populate1] :
  < Z : Peer | CRoutingTable : rt(R1,Z2,T2) ,
               CSearchManager : L , CEventMap : (< N1 ; N2 > EM) , CMessages : LM >
=>
  < Z : Peer | CRoutingTable : rt(changePop(R1,N1,N2,POPULATE-TIME-BIG),Z2,T2) ,
               CSearchManager : append(L, randomKey(N1,N2)) ,
               CEventMap : (EM < N1 ; N2 >) , CMessages : LM >
    if  (N1 =/= 0 or N2 =/= 0) /\ timePop(R1,N1,N2) /\
        (N2 < KK or N1 < KBASE or
        getNumContacts(getRZ(R1,< N1 ; N2 >)) <= (K * 2) quo 10 ) .
```

4.6.3. Deletion of off line contacts.

The *deletion of off line contacts* process, is specified by two rules:

1. If the expiration time of the first contact is not zero, the contact is moved to the bottom of the bin, and all type 4 contacts are removed from the bin.

   ```
   crl [del-dead1] :
     < Z : Peer | CRoutingTable : rt(R1,Z2,T2),
                  CMessages : LM ,
                  CEventMap : (< N1 ; N2 > EM) ,
                  CSearchManager : L >
   =>
     < Z : Peer | CRoutingTable : rt(changeRem1(R1,N1,N2),Z2,T2),
                  CMessages : LM ,
                  CEventMap : EM < N1 ; N2 > ,
                  CSearchManager : L >
       if (N1 =/= 0 or N2 =/= 0) /\ timeRem(R1,N1,N2) == 1 .
   ```

   The`timeRem` operation takes the routing zone of the routing table and the level and zone index of the routing bin to be considered and returns 1 if the time for removing off line contacts is zero and the expiration time of the first contact of the bin is greater than zero.

   The operation `changeRem1` takes the routing zone of the routing table, and the level and zone index of the routing bin to be considered and removes all type 4 contacts from the bin and changes the time for removed again.

2. If the expiration time of the first contact has expired a `HELLO-REQ` message is sent, the type increased in one and the expire time set to two minutes.

```
crl [del-dead2] :
  < Z : Peer | CRoutingTable : rt(R1,Z2,T2) ,
               CMessages : LM ,
               CEventMap : (< N1 ; N2 > EM) ,
               CSearchManager : L >
=>
  < Z : Peer | CRoutingTable : rt(changeRem2(R1,N1,N2),Z2,T2) ,
               CMessages : append(LM, HELLO-REQ(Z2,first-contact-bucket(R1,N1,N2),T2,1)) ,
               CEventMap : EM < N1 ; N2 >,
               CSearchManager : L  >
    if (N1 =/= 0 or N2 =/= 0) /\ timeRem(R1,N1,N2) == 2 .
```

The`timeRem` operation returns 2 if the time for removing off line contacts is zero and the expiration time of the first contact of the bin is also zero.

The `changeRem2` operation, changes the remove time of the routing zone with a given level and zone index. It also increases the type and changes the expiration time of the first contact in the bin. It also removes all type 4 contacts of the bin.

4.6.4. When nothing has to be done.

There are two rules that specify the case when the first item of the `CEventMap` list does not have to realize any process. In this case, the first item of the `CEventMap` is moved to the tail of the list.

The first rule deals with the routing bin processes. It checks the first item on the event list. If it is not the consolidate process it checks the time for populate and the time for remove off line contacts and if their are not zero, it puts the the bin identification at the end of the event list.

```
crl [next] :
      < Z : Peer | CRoutingTable : rt(R1,Z2,T2) ,
                   CMessages : LM ,
                   CEventMap : (< N1 ; N2 > EM) ,
                   CSearchManager : L >
=>
  < Z : Peer | CRoutingTable : rt(R1,Z2,T2) ,
               CMessages : LM ,
               CEventMap : EM < N1 ; N2 >,
               CSearchManager : L  >
    if N1 =/= 0 and timeRem(R1,N1,N2) == 0 and timePop(R1,N1,N2) == false and
       minTime(rt(R1,Z2,T2)) == 0 .
```

The second rule deals with the consolidation process. If the first item of the event list is the flag value `< 0 ; 0 >`, the rule checks the time for consolidate, if it not zero it moves the flag to the end of the list.

```
rl [next2] :
      < Z : Peer | CRoutingTable : rt(R1,Z2,T2) ,
                   CMessages : LM ,
                   CEventMap : (< 0 ; 0 > EM) ,
                   CSearchManager : L >
=>
```

```
< Z : Peer | CRoutingTable : rt(R1,Z2,T2) ,
             CMessages : LM ,
             CEventMap : EM < 0 ; 0 >,
             CSearchManager : L  > .
```

5. Routing table properties.

We define a routing table with four levels and 26 contacts over seven routing bins (see anex A).

We explore, the shortest and the longest time it takes to reach a desired state using the `find earliest` and `find latest` commands of real-time Maude.

The syntax of these commands is as follows:
(`find earliest` *initState* `=>*` *searchPattern* [`such that` *cond*] .)
(`find latest` *initState* `=>*` *searchPattern* [`such that` *cond*] `with no time limit` .)
(`find latest` *initState* `=>` *searchPattern* [`such that` *cond*] `in time` *timeLimit* .)
where:

- *initState* is a real-time system of sort `GlobalSystem`. Values are obtained by enclosing a system, in our specification a configuration of objects and messages, between brackets.
- *searchPattern* is the description of the global system we are looking for.

For example, we are interested in the maximum time a type 4 contact will be in the table. Thus, we obtain the time it takes to remove all type 4 contacts from the initial configuration. The property is important since having offline contacts in the table prevent new contacts to access it, limiting the chances of finding information.

We execute the command:

```
(find earliest  {
   < c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,68760,5400) : Peer |
      CroutingTable : RT1,  CSearchManager : nil ,
      CEventMap : (< 3 ; 0 > < 4 ; 2 > < 4 ; 3 > < 2 ; 1 > < 2 ; 2 >
                   < 3 ; 6 > < 3 ; 7 > < 0 ; 0 > , CMessages : nilMsgL > }
    =>* C:Configuration} such that (num-type4(C:Configuration) == 0) .)
```

where the `num-type4` operation computes the number of contacts of type 4 in a routing table.

The `find earliest` command explores in parallel all possible executions from the given configuration until one of them reaches a configuration that fulfills the given conditions. This means, that all the actions that can take place in the given configuration will be explored. In particular, type 4 contacts are removed from a bin when the time to execute the remove offline process is reached.

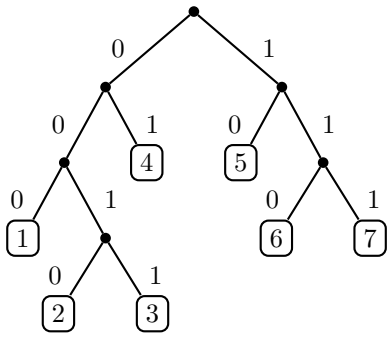
The result includes the reached configuration and the time it takes to obtain it. As we are interested only in the time value, we do not show the value of the node attributes.

```
Result:
{< c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,68760,5400): Peer |
    CEventMap : ..., CMessages : ..., CroutingTable : ...>} in time 48
```

Notice that, in this example, we have changed the `OFFLINE-CHECK` constant value to a big number to prevent the remove offline contacts process from generating new type 4 contacts.

6. Anex A. Proof terms.

Suppose the following routing tree, with owner contact `c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,14392,8)` :



Contacts in bin 1 are at a distance less than 0001 from the owner, that is, the most significative bits of the contacts in this bin are 011. In bin 2 the significative bits of the contacts are 0101 in bin 3 0100 and so on. A Maude term that represent it is:

```
rt(rz(rz(rz(rb(empty-bin !
                  c(0 ; 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0,ip,udp,type0,40000,7200) !
                  c(0 ; 1 ; 1 ; 0 ; 0 ; 0 ; 1 ; 0,ip,udp,type1,3602,5400) !
                  c(0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0,ip,udp,type4,15000,60) !
                  c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1,ip,udp,type2,1,3600) !
                  c(0 ; 1 ; 1 ; 1 ; 1 ; 1 ; 0 ; 1,ip,udp,type3,0,1) !
                  c(0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0 ; 1,ip,udp,type3,0,1),3,0,3000,48),
         rz(rb(empty-bin ! c(0 ; 1 ; 0 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type1,3700,5300) !
                         c(0 ; 1 ; 0 ; 1 ; 0 ; 1 ; 0 ; 1,ip,udp,type2,5,3600) !
                         c(0 ; 1 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1,ip,udp,type2,52,3548),4,2,1500,23),
          rb(empty-bin ! c(0 ; 1 ; 0 ; 0 ; 0 ; 1 ; 1 ; 1 ,ip,udp,type0,94000,7140) !
                   c(0 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0,ip,udp,type4,6000,460),4,3,8,1),2300,46),3090,24),
      rb(empty-bin ! c(0 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1 ; 0,ip,udp,type0,83000,5000) !
                  c(0 ; 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1,ip,udp,type4,1000,456) !
                  c(0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 1,ip,udp,type4,40000,432) !
                  c(0 ; 0 ; 0 ; 1 ; 1 ; 0 ; 0 ; 1,ip,udp,type2,400,3200) !
                  c(0 ; 0 ; 0 ; 1 ; 1 ; 1 ; 1 ; 1,ip,udp,type3,20,1),2,1,2690,12),2300,25),

      rz(rb(empty-bin ! c(1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 0 ; 1,ip,udp,type2,60,3540) !
                     c(1 ; 1 ; 1 ; 0 ; 0 ; 1 ; 1 ; 1,ip,udp,type4,1000,43) !
                     c(1 ; 1 ; 1 ; 1 ; 1 ; 0 ; 1 ; 1 ; 1,ip,udp,type1,3900,5100) !
                     c(1 ; 1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0,ip,udp,type4,2000,160) !
                     c(1 ; 1 ; 0 ; 1 ; 1 ; 0 ; 1 ; 1,ip,udp,type3,0,1),2,2,2400,44),
         rz(rb(empty-bin ! c(1 ; 0 ; 1 ; 1 ; 0 ; 1 ; 0 ; 0,ip,udp,type0,9318,5090) !
                         c(1 ; 0 ; 1 ; 0 ; 0 ; 0 ; 0 ; 1,ip,udp,type0,9732,4670),3,6,2560,55),
           rb(empty-bin ! c(1 ; 0 ; 0 ; 1 ; 0 ; 1 ; 0 ; 0,ip,udp,type1,3780,5230) !
                          c(1 ; 0 ; 0 ; 0 ; 0 ; 1 ; 0 ; 1,ip,udp,type2,1300,2300) !
                          c(1 ; 0 ; 0 ; 1 ; 0 ; 0 ; 1 ; 0,ip,udp,type4,2000,370),3,7,1300,31),3001,45) ,
                                                                                  3110,2),2121,22),
                  c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,14392,8), 2500) .
```

- Bin 1 has 6 contacts. One of type 0, one of type 1, one of type 2, two of type 3 and one of type 4. Contact 5 has creation time set to zero, and the expire time set to 1. Contact 6 has also the creation time set to zero and the expire time set to 1. Time is expressed in seconds, which is the smallest unit of time used in the system. Contact one was created 40000 seconds ago and its expire time is set to 2 hours.

  Bin 1 is on level 3 and zone index 0, its population time is set to 50 minutes (expressed in seconds) and its time to remove off line contacts is set to 48 seconds.

- Bin 2 has three contacts: two of type 2 with expire time less than an hour and one of type 1 with an expire time between an hour and an hour and a half. It has no contacts of type 4. It is in level 4 and zone index 2, its populate time is set to 25 minutes, and its remove off line contacts to 23 seconds.

- Bin 3 has two contacts, the second one has type 4. The expire time of the non type 4 contacts are greater then an hour. The bin is in level 4 and zone index 3, its populate time is 8 seconds and its remove time for off line contacts is set to 1

- The internal node which is the father of nodes two and three has a population time of 38 minutes and 20 seconds and a time to remove off line contacts of 46 seconds. The father of node 1 has a population time of 3090 seconds and a remove time of 24 seconds.

- Bin 4 has 5 contacts, two of them are of type 4, and one of them has type 3 and an expire time of 1. It is in level 2 and zone index 1. Its population time is of 2690 seconds and its remove time is of 12 seconds.

- ...

- The owner of the routing table is contact

  ```
  c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,14392,8)
  ```

  and the time for consolidation is 41 minutes and 40 seconds.

The node with ID `0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0` is represented by the term:

```
< c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,14392,8) : Peer |
  CRoutingTable : RT1,
  CEventMap : (< 3 ; 0 > < 4 ; 2 > < 4 ; 3 > < 2 ; 1 > < 2 ; 2 > < 3 ; 6 >
               < 3 ; 7 > < 0 ; 0 >) ,
  CSearchManager : nil ,
  CMessages : nilMsgL >
```

and the command to rewrite the term is

```
(tfrew in PROOF-TERM : {
  < c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,14392,8) : Peer |
    CRoutingTable : RT1,
    CEventMap : (< 3 ; 0 > < 4 ; 2 > < 4 ; 3 > < 2 ; 1 > < 2 ; 2 > < 3 ; 6 >
                 < 3 ; 7 > < 0 ; 0 >) ,
    CSearchManager : nil ,
    CMessages : nilMsgL >
 }  in time < 100  .)
```

7. Anex B. Maude specification code.

See the web page `http://maude.sip.ucm.es/kademlia`

References

[1] R. Bakhshi and D. Gurov. Verification of peer-to-peer algorithms: A case study. In *Combined Proceedings of the Second International Workshop on Coordination and Organization, CoOrg 2006, and the Second International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, MTCoord 2006*, volume 181 of *Electronic Notes in Theoretical Computer Science*, pages 35–47. Elsevier, 2007.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[3] S. Merz, T. Lu, and C. Weidenbach. Towards Verification of the Pastry Protocol using TLA+. In R. Bruni and J. Dingel editors, *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems, FORTE 2011*, vol. 6722, Reykjavik, Islande, 2011.

[4] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In P. Druschel, M. F. Kaashoek, and A. I. T. Rowstron, editors, *Revised Papers from the First International Workshop on Peer-to-Peer Systems, IPTPS 2001*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2002.

[5] D. Mysicka. Reverse Engineering of eMule. An analysis of the implementation of Kademlia in eMule. Semester thesis, Dept. of Computer Science, Distributed Computing group, ETH Zurich, 2006.

[6] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20:161–196, 2007.

[7] I. Pita. A formal specification of the Kademlia distributed hash table. In V. M. Gulías, J. Silva, and A. Villanueva, editors, *Proceedings of the 10 Spanish Workshop on Programming Languages, PROLE 2010*, pages 223–234. Ibergarceta Publicaciones, 2010. `http://www.maude.sip.ucm.es/kademlia`. Informal publication–Work in progress.

[8] Pita, I. and Riesco, A., Specifying and Analyzing the Kademlia Protocol in Maude. 9th International Workshop on Rewriting Logic and its Applications, WRLA 2012.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *ACM SIGCOMM Computer Communication Review - Proceedings of the 2001 SIGCOMM conference*, 31:161–172, October 2001.

[10] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In R. Guerraoui, editor, *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31:149–160, October 2001.