

Formal Specification of the Kademia and the Kad Routing Tables in Maude

Isabel Pita, María-Inés Fernández-Camacho

► **To cite this version:**

Isabel Pita, María-Inés Fernández-Camacho. Formal Specification of the Kademia and the Kad Routing Tables in Maude. Narciso Martí-Oliet; Miguel Palomino. 21th International Workshop on Algebraic Development Techniques (WADT), Jun 2012, Salamanca, Spain. Springer, Lecture Notes in Computer Science, LNCS-7841, pp.231-247, 2013, Recent Trends in Algebraic Development Techniques. <10.1007/978-3-642-37635-1_14>. <hal-01485972>

HAL Id: hal-01485972

<https://hal.inria.fr/hal-01485972>

Submitted on 9 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Formal Specification of the Kademlia and the Kad Routing Tables in Maude^{*}

Isabel Pita and María Inés Fernández Camacho

Dept. Sistemas Informáticos y Computación, Universidad Complutense de Madrid,
Spain

`ipandreu@sip.ucm.es, minesfc@sip.ucm.es`

Abstract. Kad is the implementation by eMule and aMule of the Kademlia peer-to-peer distributed hash table protocol. Although it agrees with the basic behaviour of the protocol, there are some significant differences. This paper presents the specification of both the Kademlia and the Kad routing tables, using the specification language Maude. As far as we know, this is the first such a formal development. The routing tables present a dynamic behavior in the sense that they should be able to send messages to other peers and they should have a notion of time for raising events and detect no answered messages. Our main contribution is the integration of these dynamic aspects in the protocol specification.

Keywords: Kademlia, Kad, distributed specification, formal analysis, Maude, Real-Time Maude.

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 [?], Pastry [?], CAN [?], and Kademlia [?]. However, only Kademlia has been implemented in P2P networks through the eMule¹ and aMule² 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 clients³ and one by many other clients including Mainline⁴ and BitComet⁵.

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 [?] a formal verification of Chord's stabilization algorithm using the π -calculus. Lately Lu, Merz, and Weidenbach [?] have

^{*} Research supported by MEC Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC1465).

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>

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 Kademia 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 Kademia differs from the real implementation made in Kad. A Kademia routing table is a proper binary tree whose leaves are lists of at most k contacts, called k -buckets. A 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, Kademia 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 the ID of the owner is in the range of the last bucket. Kademia 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 Kademia, 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. Kademia has no specific actions to update its routing table periodically, checking for offline 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 [?]. 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 offline 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 [?]. 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 offline 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.

The structure of the paper is as follows: In Section ?? we give some notions about the Maude language and the Kademlia and Kad DHTs. Sections ?? and ?? present the specification of the Kademlia and the Kad networks respectively. Section ?? defines a Kad routing table and proves some properties about it. Finally Section ?? presents some conclusions and future work.

2 Preliminaries

We present in this section the basic notions about Maude, Kademlia, and Kad.

2.1 Maude

Maude [?] is a formal specification language based on rewriting logic. In Maude, the state of a system is formally specified as an algebraic data type by means of a membership equational specification. We define new types (by means of keyword `sort(s)`); subtype relations between types (`subsort`); operators (`op`)

for building values of these types; equations (**eq**) that identify terms built with these operators; and membership axioms (**mb**) which specify terms as having a given sort.

Both equations and membership axioms may be conditional (**ceq** and **cmb**). Conditions, introduced by the keyword **if** can be either a single equation, a single membership, or a conjunction of equations and memberships using the binary conjunction connective \wedge . The equations can be ordinary equations ($t = t'$), matching equations ($t := t'$), or abbreviate Boolean equations (t).

The sorts connected by a subtype relationship are grouped into equivalence classes called *Kinds*. Kinds are implicitly associated with connected components of sorts and are considered as *error supersorts*. They are written by enclosing the name of one of the connected sorts in square brackets.

The *dynamic* behaviour of the system is specified by rewrite rules of the form $t \longrightarrow t'$ if C , that describe the local, concurrent transitions of the system. That is, when a part of a system matches the pattern t and satisfies the condition C , it can be transformed into the corresponding instance of the pattern t' .

In object-oriented specifications, *classes* are declared with the syntax **class** $C \mid a_1 : S_1, \dots, a_n : S_n$, where C is the class name, a_i is an attribute identifier, and S_i is the sort of the values this attribute can have. An *object* is represented as a term $\langle O : C \mid a_1 : v_1, \dots, a_n : v_n \rangle$ where O is the object's name, belonging to a set **Obj** of object identifiers, and the v_i 's are the current values of its attributes. The order of the attributes in an object is irrelevant. *Messages* are defined by the user for each application (introduced with syntax **msg**).

In a concurrent object-oriented system the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting. The behaviour associated with the messages is specify by rules of the form: $m_1 \dots m_k o_1 \dots o_l \longrightarrow m'_1 \dots m'_s o'_1 \dots o'_t$ if C where the m_i are messages of type **Msg**, and the o_i are objects, with their class and attributes. Objects that appear only in the lefthand side of the rule are removed from the configuration, and objects that appear only in the righthand side of the rule are added to the configuration. Regarding the attributes, the only object attributes made explicit in a rule are those relevant for that rule. Attributes mentioned only in the lefthand side of a rule are preserved unchanged, the original values of the attributes mentioned only on the righthand side do not matter, and all attributes not explicitly mentioned are left unchanged. We use Full Maude's object-oriented notation and conventions [?, Part II] throughout the whole paper.

The notion of time is specified by using the real time extension of Maude [?]. In the Real-Time Maude language, one can make explicit the way in which change in a system is dependent on time. The language provides two sorts: **GlobalSystem** and **System** to represent the whole system and the state of the system. The whole system is obtained from the state by the constructor $\{ _ \} : \mathbf{System} \rightarrow \mathbf{GlobalSystem}$.. The transitions of the system that are assumed to take zero time are specified over the **System** sort. The elapse of time in a system is modelled by the *tick* rule: $l : \{t\} \longrightarrow \{u\}$ if C . Tick rules must

only be applied on the system as a whole, i.e., on terms of sort `GlobalSystem`, to ensure that time advances uniformly in all parts of the system.

The time domain may be discrete or dense. In particular, we use the discrete time definition with natural numbers, that provides us with a sort `Time` such that `subsort Nat < Time` ., a sort extension with the infinite value: `subsort Time < TimeInf` ., and the infinite value: `op INF : -> TimeInf` .

2.2 Kademlia

Kademlia [?] is a P2P distributed hash table used by the peers to access files shared by other peers. In Kademlia both peers and files are identified with n -bit quantities, computed by a hash function. Information of shared files is kept in the peers with an ID *close* to the ID file, where the notion of distance between two IDs is defined as the bitwise exclusive-or of the n -bit quantities. Then, the lookup algorithm which is based on locating successively *closer* nodes to any desired key has $\mathcal{O}(\log n)$ complexity.

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 $i = 0, \dots, n$ and n the ID length. Note that each list contains IDs that differ in the i th 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. Buckets are kept sorted by the time contacts were last seen.

The routing table is organized as a binary tree whose leaves are buckets. Each bucket is identified by the common prefix of the IDs it contains. The binary tree in the basic version of Kademlia is a list of buckets, since the only buckets that can be split are those with common prefix equal to the one of the routing table owner. Figure ?? (partially borrowed from [?]) shows a routing table for node 00000000.

Kademlia does not have explicit operations to maintain the information of the routing table. When a node receives any message (request or reply) from another node, it updates the appropriate bucket for the sender's node ID. If the sender node exists, it is moved to the tail of the list. If it does not exist and there is free space, it is inserted at the tail of the list. Otherwise, if the bucket has not free space, the node at the head of the list is contacted and 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 and the new contact added to the appropriate bucket; if the bucket cannot be divided the first contact is moved to the tail, and the new node is discarded.

2.3 The Kad Routing Table

Kad [?] is an implementation of the Kademlia distributed hash table. Its routing tables allow for more contacts than the Kademlia ones. They are not lists anymore, but left-balanced binary trees. The routing tree nodes are called *routing zones*. The buckets, now called *routing bins* are located at the leaf nodes and

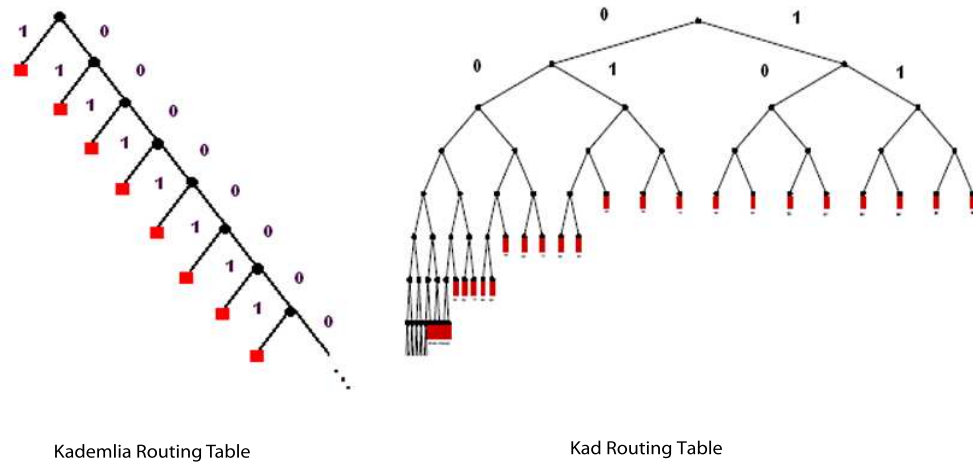


Fig. 1. Kademlia and Kad Routing table structures.

are lists of at most 10 contacts. Routing zones may be identified by their level and their zone index, that is the distance of the node to the leftmost node of its level. Kad has a looser splitting rule than Kademlia, it allows to split a bin if its level is smaller than 4 or its zone index is smaller than 5. Figure ?? shows a Kad routing table.

Kad has explicit operations to maintain the information of the routing table, and does not rely on receiving messages as Kademlia. The process that adds new contacts to a bin is done once an hour. If the bin has less than three contacts or if splitting is allowed, the process selects a random ID from the bin ID space and starts a search for it. It will find new contacts that will be inserted into the bin. The process that deletes dead contacts from the bin is executed once a minute. It checks if the *expire time* of the first contact in the bin is ok. Every contact has an expire time variable which holds the time when the contact has to be checked whether it is still alive. If this time has expired, the routing table sends a HELLO-REQ message to the contact, increases a type variable, with values from 0 to 4, that indicates how long the contact has been online, and sets its expire time to 2 minutes. The process also removes all contacts of type 4 from the bin. If the node receives a HELLO-RES message indicating that the contact is alive, it places the contact at the end of the list, re-adjusts the type of the contact and sets a new expire time. If the contact does not reply, it will be removed from the bin when its type raises level 4.

Finally, there is a process that consolidates almost empty bins. It is done on adjacent bins that have the same parent routing zone if they both together have less than 5 contacts. It is run every 45 minutes on the whole routing tree.

3 The Maude Specification of Kademia

Kademia buckets are represented by the sort `Bucket{X}` as a list of contacts, where the contacts are parameters of the specification, represented by the sort `X$Contact`. The subsort `NeBucket{X}` represents non empty buckets.

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

op empty-bucket : -> Bucket{X} [ctor] .
op !_ : Bucket{X} X$Contact -> [Bucket{X}] [ctor] .
```

Non-empty k -buckets are bounded and do not have repeated contacts by means of the membership axiom, where the `bucketDim` constant is set to k :

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

We have defined operations to perform all the bucket functions, like moving a contact to the tail, adding contacts, and removing contacts (see [?]).

Routing tables are defined as lists of buckets. We do not consider in this version of the specification the optimizations proposed in [?] that allow keeping more contacts in the routing table, therefore there is no need for a tree of buckets. Notice that routing tables have at least one bucket that cannot be empty.

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

op !_ : Bucket{X} [RoutingTable{X}] -> [RoutingTable{X}] [ctor] .
```

The following membership axiom sets when a list with more than one bucket is a routing table. It checks whether the table does not exceed the maximum size, that the contacts in each bucket are the appropriate and that at least one of the two last buckets is not empty. A single non-empty bucket is a routing table by the previous subsort relationship.

```
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)) .

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 have defined operations to divide a bucket and to compute the nearest contact to a given one among others.

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 [?]. To encapsulate the behaviour of the routing table in the peer we define the RT attribute of the object Peer as a configuration of a routing table, a message, and a value of sort Time. By means of the rules RT-**receive** and RT-**send** the routing table messages are sent out from the peer and the reply messages go inside the routing table. Notice that the reply is captured by the routing table only when a message to that object has been sent before.

```

op Peer : -> Cid .
op RT :_ : Configuration -> Attribute [ctor] .
op {+_+_} : RoutingTable{X} Msg TimeInf -> Configuration [ctor] .

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

r1 [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 } > .
cr1 [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 .

```

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 offline, and calls the auxiliary function **add-entry2** on the routing table. If the first contact is offline, the time value of the configuration will raise zero and rule **add2** will remove the offline 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.

```

cr1 [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)) .
cr1 [add1] : { add-entry(Z1,R,Z2) + none + INF } =>
  { add-entry2(Z1,R,Z2) +

```

```

    PING(Z2,first-contact(find-bucket(Z1,R,1)),RPCRemove,1) + PT }
if full-bucket?(find-bucket(Z1,R,1)) .
rl [add2] : { add-entry2(Z1,R,Z2) + M + 0 } =>
  { add-entry-aux(Z1,R,1,Z2,false) + none + INF } .
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 .
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 .

```

The Kademia specification is written in 7 modules and it defines about 60 operations over the defined sorts. It makes use of 6 rewriting rules to simulate the routing table interaction with the node. (<http://maude.sip.ucm.es/kademia> [?])

4 Kad Processes in Maude.

The Kad processes responsible for the routing table maintenance are executed from periodically, instead of relying on the occurrence of an event as in the Kademia routing table. For this reason time values are present all over the routing table specification, since contacts, bins, and the routing table must keep the time left to perform their actions.

We define the moment at which a process must occur with a constant value that is decreased as the time passes. When the value reaches zero the process is executed and the time is reset to the initial value. Time is expressed in seconds.

The Kad specification is written in 17 modules and it defines about 80 operations over the defined sorts. It makes use of 8 rewriting rules to simulate the routing table interaction with the node. (<http://maude.sip.ucm.es/kademia> [?])

4.1 The Kad Routing Table

The routing table representation in Kad differs from the one in Kademia. Since Kad routing tables may be a complete tree up to level 4, the specification is no more a list of buckets, but a binary tree. Besides this, it is necessary to keep the time left for the processes to run.

Kad routing tables are non-empty binary trees of sort `RoutingZone` together with the contact information of the routing table owner and the time left to execute the consolidation process, which is performed on the whole tree.

```

sort RoutingTable .
op rt : RoutingZone Kad-Contact Time -> RoutingTable [ctor] .

```

Routing bins are defined as lists of contacts of length K .

```
sorts NeRoutingBin RoutingBin .
subsort NeRoutingBin < RoutingBin .

op empty-bin : -> RoutingBin [ctor] .
op !_ : RoutingBin Kad-Contact -> [NeRoutingBin] [ctor] .

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

Each contact, besides its identifier, IP address, and UDP port, keeps the time it has been active and the time remaining for checking whether it is still active. The time a contact has been active is the only time value that is increased as time passes.

```
sort Kad-Contact .
op c : BitString128 IP UDP Contact-types Time Time -> Kad-Contact [ctor] .
```

The `RoutingZone` sort represents a Kad routing tree. Subtrees are represented in the *Kind*, since the level and zone index values of their leaves do not correspond to a complete tree. A subtree may only be one routing bin, together with its level, zone index, and the time to populate and to remove the offline contacts. Both processes are performed in each bin independently.

```
op rb : RoutingBin Nat Nat Time Time -> [RoutingZone] [ctor] .
```

The routing zone may also be the composition of its two subtrees together with the time to populate and the time to remove offline contacts. Kad uses the time to populate in the internal tree nodes during the consolidation process to define the population time for the new node that consolidates the two old ones. We maintain the time to remove offline contacts because it is in the Kad implementation, although it is not used in our specification. There is no need in the specification to maintain the level and zone index values in the internal tree nodes.

```
op rz : [RoutingZone] [RoutingZone] Time Time -> [RoutingZone] [ctor] .
```

A membership axiom defines which of the binary trees formed with the previous constructors are Kad routing trees. In particular it checks whether the routing bin (with its level, zone index, and time values) is the root of the tree, that is, if its level and zone index are both zero or if it is a tree with more than one bin that has the correct tree structure, and the appropriate contacts in each bin.

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

4.2 A Kad Peer

The Kad routing table can request the node to send messages and start a looking-for process. Therefore, the routing table specification needs to consider other parts of the node. In the specification a node/peer is an object of class `Peer`, with four attributes: the routing table, defined in Section ??; the search manager, which is a list of the keys the peer is looking for; a list of the messages to send; and a list of events, that simulates the list of events in the Kad implementation and is used to go over the bins looking for the process that its time to execute.

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

4.3 The Passage of Time

The passage of time is defined by the functions `delta` and `mte`. The `delta` function spreads the passage of time to all time values of the specification as follows:

```
op delta : Configuration Time -> Configuration [frozen (1)] .
op delta : RoutingTable Time -> RoutingTable .
op delta : RoutingZone Time -> RoutingZone .
op delta : RoutingBin Time -> RoutingBin .

vars CF CF' : Configuration . vars T T1 T2 TC : Time .
var Z1 : Kad-Contact . var R : RoutingZone . var B : RoutingBin .
vars KR1 KR2 : [RoutingZone] . var RT : RoutingTable . var S : BitString128 .
var ip : IP . var udp : UDP . var CT : Contact-types . vars N1 N2 : Nat .

eq delta(none,T) = none .
ceq delta(CF CF', T) = delta(CF,T) delta(CF',T)
  if CF /= none and CF' /= none .
eq delta(< Z1 : Peer | CroutingTable : RT, ... >,TC) =
  < Z1 : Peer | CroutingTable : delta(RT,TC), ... > .
eq delta(rt(R,Z1,T2), TC) = rt(delta(R,TC),Z1,T2 monus TC) .
eq delta(rb(B,N1,N2,T1,T2),TC) = rb(delta(B,TC),N1,N2,T1 monus TC,T2 monus TC) .
eq delta(rz(KR1,KR2,T1,T2),TC) =
  rz(delta(KR1,TC),delta(KR2,TC),T1 monus TC,T2 monus TC) .
eq delta(empty-bin,TC) = empty-bin .
eq delta(B ! c(S,ip,udp,CT,T1,T2),TC) =
  delta(B,TC) ! c(S,ip,udp,CT,T1 plus TC,T2 monus TC) .
```

while the `mte` function computes the time that passes in each step as the minimum of all the time values in the system.

```
op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(none) = INF .
ceq mte(CF CF') = min(mte(CF), mte(CF')) if CF /= none and CF' /= none .
```

```
eq mte(< Z1 : Peer | CroutingTable : RT ,...>) = minTime(RT) .
```

```
op minTime : RoutingTable -> Time .
op minTime : RoutingZone -> Time .
eq minTime(rt(R,Z2,T2)) = min(minTime(R),T2) .
eq minTime(rb(B,N1,N2,T1,T2)) = min(T1,T2) .
eq minTime(rz(KR1,KR2,T1,T2)) = min(minTime(KR1),minTime(KR2)) .
```

The time values of the internal nodes that appear in the last equation are not taken into account in the minimum time computation, because they are not directly associated with any process.

4.4 The Populate Process

The population process is done in each routing bin when the time defined by the constant POPULATE-TIME-BIG in each routing bin reaches zero and the number of contacts in the bin is less than 20% of the bin size, or it is a bin that can be split. It is also required to be this the next bin in the event map attribute of the node. The process adds a random contact to the search process; and resets the population time of the bin.

```
cr1 [populate1] :
  < Z : Peer | CroutingTable : rt(R1,Z2,T2), CSearchManager : L ,
    CEventMap : (< N1 ; N2 > EM) , ...> =>
  < Z : Peer |
    CroutingTable : rt(changePop(R1,N1,N2,POPULATE-TIME-BIG),Z2,T2) ,
    CSearchManager : append(L, randomKey(N1,N2)) ,
    CEventMap : (EM < N1 ; N2 >) , ... >
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) .
```

where:

- changePop changes the population time of the bin to the given value.
- append(L, randomKey(N1,N2)) appends a new look-up for a random key value to the search manager.
- timePop checks whether the time to populate in the bin is set to zero.
- getNumContacts obtains the number of contacts of a bin.
- getRZ gets the routing zone of a level and zone index.

4.5 The Remove Offline Contacts Process

The remove offline contacts process is done in each routing bin when the time defined by the constant NEXT-REM in each routing bin reaches zero.

Kad classifies contacts in five groups: type 0: very good contact, known for at least two hours; types 1 and 2: less good contact; type 3: newly inserted contact; type 4: the contact has not responded and will be removed in the next process. In addition each contact keeps two time values: the creation time, which is the

time to determine how long the contact has been known, and the expire time, which is the time to check if the contact is still alive.

The process checks the first contact of the bin. If its expire time is not zero, the process removes type 4 contacts from the bin; push to the bottom the first contact (by the operation `changeRem1`); and puts a new remove time in the bin.

```

cr1 [del-dead1] :
  < Z : Peer | CroutingTable : rt(R1,Z2,T2), CEventMap : < N1 ; N2 > EM ,. >
=> < Z : Peer | CroutingTable : rt(changeRem1(R1,N1,N2),Z2,T2),
      CEventMap : EM < N1 ; N2 > ,...>
if (N1 /= 0 or N2 /= 0) /\ timeRem(R1,N1,N2) == 1 .

eq [cr11] : changeRem1(rb(B1,N3,N4,T1,T2),N1,N2) =
  rb(pushToBottom(first-contact(rem4(B1)),rem4(B1)),N3,N4,T1,NEXT-REM) .
ceq [cr12] : changeRem1(rz(KR1,KR2,T5,T6),N1,N2) =
  rz(changeRem1(KR1,sd(N1,1),N2),KR2,T5,T6)
if 2 ^ sd(N1,1) > N2 .
ceq [cr13] : changeRem1(rz(KR1,KR2,T5,T6),N1,N2) =
  rz(KR1,changeRem1(KR2,sd(N1,1),sd(N2,2 ^ sd(N1,1))),T5,T6)
if 2 ^ sd(N1,1) <= N2 .

```

where the `rem4` operation removes all contacts with type 4 in the bin.

If the expire time of the first contact in the bin is zero, the process: removes type 4 contacts from the bin; increases the first contact type; puts the expire time of the first contact to 2 minutes (`OFFLINE-CHECK` constant) by means of the `changeRem2` operation; and sends a `HELLO-REQ` message to the first contact. If the first contact answers, it will be moved to the tail of the bin, in other case it will be declared of type 4 and removed from the routing table in 2 minutes.

```

cr1 [del-dead2] :
  < Z : Peer | CroutingTable : rt(R1,Z2,T2), CMessages : LM ,
      CEventMap : (< N1 ; N2 > EM) ,... >
=> < 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 > ,...>
if (N1 /= 0 or N2 /= 0) /\ timeRem(R1,N1,N2) == 2 .

eq [cr21] : changeRem2(rb(B1,N3,N4,T1,T2),N1,N2) =
  rb(change-first(rem4(B1),OFFLINE-CHECK),N3,N4,T1,NEXT-REM) .
op changeRem2 : RoutingZone Nat Nat -> RoutingZone .
ceq [cr22] : changeRem2(rz(KR1,KR2,T5,T6),N1,N2) =
  rz(changeRem2(KR1,sd(N1,1),N2),KR2,T5,T6)
if 2 ^ sd(N1,1) > N2 .
ceq [cr23] : changeRem2(rz(KR1,KR2,T5,T6),N1,N2) =
  rz(KR1,changeRem2(KR2,sd(N1,1),sd(N2,2 ^ sd(N1,1))),T5,T6)
if 2 ^ sd(N1,1) <= N2 .

```

4.6 The Consolidation Process

The consolidation process is done in the routing tree when its time defined by the constant `CONSOLIDATE-TIME` of the routing table, reaches zero.

```
rl [consolidate1] :
  < Z : Peer | CroutingTable : rt(R,Z2,0), ... > =>
  < Z : Peer | CroutingTable : rt(consolidate(R),Z2,CONSOLIDATE-TIME),... > .
```

The consolidation of a routing tree, done by the `consolidate` function is a recursive process. The first equation: `con1` defines the case of a subtree with one bin, which cannot be consolidated. The equation `con2` defines the recursive case when at most one of the subtrees is a leaf. Equations `con3` and `con4` define the cases where the two subtrees are leaves. If the number of contacts of the two bins is less than half the size of the bin, bins are aggregated (`con3`). Otherwise no consolidation is needed for these bins (`con4`).

```
eq [con1] : consolidate(rb(B,N1,N2,T1,T2)) = rb(B,N1,N2,T1,T2) .
ceq [con2] : consolidate(rz(KR1,KR2,T1,T2)) =
  rz(consolidate(KR1),consolidate(KR2),T1,T2)
if not isLeaf?(KR1) or not isLeaf?(KR2) .
ceq [con3] : consolidate(rz(rb(B1,N3,N4,T1,T2),rb(B2,N3,N6,T3,T4),T5,T6)) =
  rb(add-bins(B1,B2),sd(N3,1),N4 quo 2,POPULATE-TIME,T6)
if getNumContacts(rz(rb(B1,N3,N4,T1,T2),rb(B2,N3,N6,T3,T4),T5,T6)) <
  K quo 2 /\ N6 == N4 + 1 .
ceq [con4] : consolidate(rz(KR1,KR2,T5,T6)) = rz(KR1,KR2,T5,T6)
if isLeaf?(KR1) /\ isLeaf?(KR2) /\
  getNumContacts(rz(KR1,KR2,T5,T6)) >= K quo 2 .
```

5 Case Study

We define a routing table with 4 levels and 26 contacts over 7 routing bins (Figure ??). The maximum number of contacts in a bin is set to 10. Node IDs are abbreviated to 8 bits. Contacts are of different types, from 0 to 4 and we consider realistic time values for the different process. The specification, initial terms and results of each case study can be found in <http://maude.sip.ucm.es/kademlia>.

The owner of the routing table is supposed to have the ID 01110000. The contacts of the routing bin at level 3 and zone index 0, that are shown below, are at distance less or equal to 31 from the owner ID. This means that the three most significant bits of the contacts in the bin should agree with the ones of the routing table owner.

```
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,type1,30000,5400) !
  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,type2,20000,3600) !
  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)
```

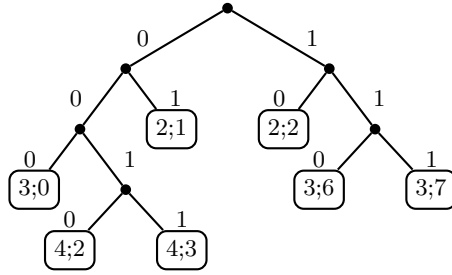


Fig. 2. Routing tree example.

There are two contacts of type 3 and one contact for types 0, 1, 2, and 4. The last two contacts have just joined the network since their creation time is set to 0 and their expire time is set to 1 second. The expire time of the first four contacts varies from 1 minute to 2 hours. The population time of the bin is set to 50 minutes and its time to remove offline contacts to 48 seconds.

Now, we can explore, for example, 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 Conclusions and Ongoing Work

We have developed a formal specification of the Kademia and the Kad routing tables that can be used by other system developers to use or update their DHTs. The specification includes a notion of time that allows for triggering events and detecting messages timeouts.

The specification will allow us to verify properties of the protocol. In particular properties that include time aspects, like the ones mentioned in the introduction. In this paper we have shown how to obtain the time it takes to remove offline contacts from a table. Similar properties, that will ease us the study of the behavior of the routing table are, for example to obtain the time it will take to execute the next populate process, or which will be the next bin to be populate, under different conditions.

As future work we plan to complete the study of the behavior of the routing table as time goes on, specially on the aspects related with the routing table correctness and the order in which the processes take place. We would also like to integrate the routing table specifications in a distributed system specification using the Maude sockets, and introduce the strategy language of Maude to control the execution of Kad processes.

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. Kademia: 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 Kademia 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 Kademia 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/kademia>. Informal publication—Work in progress.
8. I. Pita and M.I. Fernández Camacho. Formal Specification of the Kademia Routing Table and the Kad Routing Table in Maude. Technical Report 1/2013. Dept. Sistemas Informáticos y Computación Universidad Complutense de Madrid. January, 2013. <http://www.maude.sip.ucm.es/kademia>.
9. Pita, I. and Riesco, A., Specifying and Analyzing the Kademia Protocol in Maude. 9th International Workshop on Rewriting Logic and its Applications, WRLA 2012.
10. 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.
11. 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.
12. 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.