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. Using our two formal specifications, we can isolate some of those differences and detect some open issues on the original description, mainly related with message passing. In this context, our main contribution is the integration of the dynamic aspects of the routing table with the full 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-perr (P2P) systems, without the need for a centralized server. There are a large number of existing DHTs proposals; the best known are: Chord [12], Pastry [11], CAN [10], 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

* 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

algorithm using the π-calculus. Lately Lu, Merz, and Weidenbach [3] have modeled Pastry's core routing algorithms in the specification language TLA^+ and have proved properties of correctness and consistency using its model checker. There is a preliminary study of the Kademlia searching process protocol by the first author in [8], and a distributed specification of the protocol in [9]. Based on these preliminary studies we have realized a detailed specification of the Kademlia and the eMule/aMule Kad routing tables in the formal specification language Maude which include features not previously specified as sending messages to update the Kademlia routing table or raising events for populate and remove offline nodes in the Kad routing table .

The Maude algebraic specification language we use, is based on rewriting logic [2] and it supports both equational and rewriting logic computations while it offers simple an 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. In particular the Real-Time Maude tool [7,6] provides facilities to analyze the effect of time on the system.

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 are 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. It also allows us to detect some open issues on the original description, mainly related with message passing.

However, our main contribution is the integration of the dynamic aspects of the routing table with 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 time out when sending the message and triggering the appropriate action when the time expires. The actions that are performed automatically in Kad from time to time, like populate almost empty buckets, or remove offline contacts from the buckets are triggered when their time expire. These actions require to have 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 2 we give some notions about the Maude language, and the Kademlia and Kad DHTs. Sections 3 and 4 present the specification of the Kademlia and the Kad networks respectively. Section 5 defines a Kad routing table and proves some properties about it. Finally section 6 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 [2] 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 `/\`. 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 named 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 `|` $a_1 : S_1, \ldots,$ $a_n : S_n$, 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 `<` O `:` C `|` a_1 `:` v_1`, ...,` a_n `:` v_n `>` where O is the object's name, belonging to a set `Oid` of object identifiers, and the v_i's are the current values of its attributes. *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 rewrite rules specify the behaviour associated with the messages. By convention, the only object attributes made explicit in a rule are those relevant for that rule. We use Full Maude's object-oriented notation and conventions [2, Part II] throughout the whole paper.

To represent the notion of time in the specification we use the real time extension of Maude [6] . 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 [4] 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 $i = 0, \ldots, n$ and n the ID length. These lists, called *k-buckets*, have at most k elements, where k is chosen such that any given k nodes are very unlikely to fail within an hour of each other. k-buckets are kept sorted by the time contacts were last seen.

The routing table is organized as a binary tree whose leaves are k-buckets. Each k-bucket is identified by the common prefix of the IDs it contains. Since only k-buckets with common prefix equal to the one of the routing table owner can be split in two new k-buckets when they are full, the binary tree in the basic version of Kademlia is a list of k-buckets. Figure 1 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 k-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, the k-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 k-bucket can be divided, it is split and the new contact added to the appropriate k-bucket; if the k-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 [5] is an implementation of the Kademlia distributed hash table. Its routing tables allow for more contacts than Kademlia ones. They are not lists anymore, but left-balanced binary trees. The routing tree nodes are called *routing zones*. The k-buckets, now called *routing bins* are located at the leaf nodes and 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 that Kademlia, it allows to split a bin if its level is smaller than 4 or its routing zone is smaller than 5. Figure 1 (partially borrowed from [5]) 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

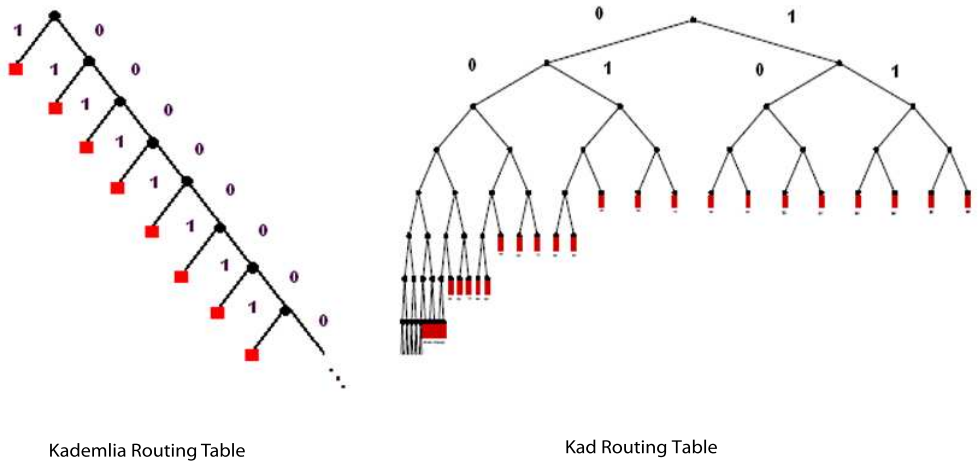

Fig. 1. Kademlia and Kad Routing table structures.

contacts has an expire time variable which holds the time when the contact has to be checked if 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 two 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 consolidate 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 Kademlia

Kademlia k-buckets are represented by the sort `Bucket{X}` as a list of contacts, where the contacts are parameters of the specification. 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:

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

We define operations to perform all the bucket functions, like move a contact to the tail, add contacts and remove contacts.

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. 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 k-bucket is a routing table. It checks whether the table does not exceed the maximum size, that the contacts in each k-bucket are the appropriate and that at least one of the two last k-buckets is not empty . A single non empty k-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 define 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 object we define the `RT` attribute of the 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 .

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 }  > .
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 .
```

The behaviour of the `add-entry` operation 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 configuration time to the `PT` value, which is the time it will wait for a reply 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.

```
ceq [add0] : add-entry(Z1,R,Z2) = add-entry-aux(Z1,R,1,Z2,true)
  if not full-bucket?(find-bucket(Z1,R,1)) .
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) + 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 .
```

4 Kad Processes in Maude.

The Kad processes responsible of the routing table maintenance are executed from time to time, instead of relying on the occurrence of an event as in the

Kademlia 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 reset to the initial value. The time is expressed in seconds.

4.1 The Kad Routing Table

The routing table representation differs in Kad from the one in Kademlia. 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 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 if 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 be only 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 if 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 requests 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 4.1; the search manager, 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 .
```

In the following, when we refer to an object we will only show the relevant part for the process.

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 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 its 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.

```
crl [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 if 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 its time defined by the constant `NEXT-REM` in each routing bin reaches zero.

Kad classifies contacts in five groups: type 0: very good contact, know for at least two hours; types 1 and 2: less good contacts; 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.

```
crl [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 move to the tail of the bin, in other case it will be declared of type 4 and removed from the routing table in two minutes.

```
crl [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 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

The current study is focused on the behaviour of the routing table as time goes on, specially on the order in which the processes take place.

We define a routing table with four levels and 26 contacts over seven routing bins (Figure 2). The maximum number of contacts in a bin is set to ten. Node IDs are abbreviated to 8 bits. Contacts are of different types, from zero to four 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`.

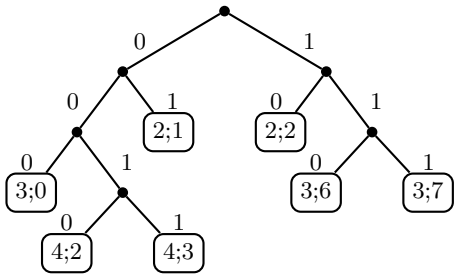

Fig. 2. Routing table example.

The owner of the routing table is suppose 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 significat 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)
```

There are two contacts of type three and one contact for types one, two, and four. The last two contacts have just joint to the network since their creation time is set to zero and their expire time is set to one second. The expire time of the first four contacts varies from one minute to two 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* `=>` *searchPatter* [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, to obtain the shortest time it takes to remove all the type 4 contacts of a given routing table we define an operation `num-type4` that computes the number of contacts of type 4 in a routing table, and 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) .)
```

The result includes the reached configuration and the time it takes to obtain it:

```
Result:
{< c(0 ; 1 ; 1 ; 1 ; 0 ; 0 ; 0 ; 0,ip,udp,type0,6876000,5400): Peer |
    CEventMap :..., CMessages : ..., CroutingTable : ...>} in time 48
```

At the moment we are analyzing static configurations in which contacts do not change its state, therefore only the contacts of type 4 in the initial configuration are considered.

6 Conclusions and Ongoing Work

We have developed a formal specification of the Kademlia and the Kad routing tables that can be used by other system developers to use or update their DHTs. The specification integrates the dynamic aspects of the routing table with the full protocol specification. The notion of time included in the specification allows triggering events and detecting messages time outs.

The specification allows to verify properties of the protocol, in particular time aspects, like the time it takes to reach a given configuration or the interaction of the different process.

As future work we would like to integrate the routing table specifications in the distributed system specification of the network presented in the WRLA12 workshop by the first author and Adrian Riesco [9]. We would also like to introduce the strategy language of Maude to control the process 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. T. Lu, S. Merz, and C. Weidenbach. Model checking the Pastry routing protocol. In J. Bendisposto, M. Leuschel, and M. Roggenbach, editors, *10th International Workshop Automatic Verification of Critical Systems, AVOCS 2010*, pages 19–21. Universität Düsseldorf, 2010.
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. *Real-Time Maude 2.3 Manual*, 2007. `http://heim.ifi.uio.no/~peterol/RealTimeMaude`.
7. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20:161–196, 2007.
8. 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.
9. Pita, I. and Riesco, A., Specifying and Analyzing the Kademlia 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.