A Maude specification of the Kademlia distributed hash table: centralized version*

Isabel Pita and Adrián Riesco

Technical Report SIC-03-17

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

*Partially supported by MINECO Spanish projects StrongSoft (TIN2012-39391-C04-04), and Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731).

Abstract

Kademlia is the most popular peer-to-peer distributed hash table (DHT) currently in use. It offers a number of desirable features that result from the use of a notion of *distance* between objects based on the bitwise exclusive *or* of n-bit quantities that represent both nodes and files. Nodes keep information about files *close* or *near* to them in the key space and the search algorithm is based on looking for the *closest* node to the file key. The structure of the routing table defined in each peer guarantees that the lookup algorithm takes no longer than $\log n$ steps.

We have developed a formal specification of a P2P network that uses the Kademlia DHT in the Maude language. We use sockets to connect different Maude instances and create a P2P network where the Kademlia protocol can be used, hence providing an implementation of the protocol which is correct by design. Then, we show how to abstract this system in order to analyze it using *Real-Time Maude*. The model is fully parameterized regarding the time taken by the different actions to facilitate the analysis of various scenarios. Finally, we use time-bounded model-checking and exhaustive search to prove properties of the protocol over different scenarios. This report focuses on the implementation details of the centralized specification.

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

Contents

1 Introduction

Peer-to-peer (P2P) systems have seen a great growth in the last years mainly due to file sharing applications. There are two basic approaches for searching contents in P2P networks: the unstructured approach is based on flooding the network and was used in the first implementations of P2P networks, like Gnutella. The structured approach uses a distributed hash table (DHT) and is the one currently in use in most P2P networks. A large number of DHTs have been studied through theoretical simulations and analysis, such as Chord [20], CAN [16], and Pastry [17]. But, despite the large effort devoted to the topic only Kademlia [8] is being used in real P2P networks through the eMule [6] and aMule [1] clients which give access to millions of users. Also BitTorrent has introduced a Kademlia DHT in its P2P network [5], although it is not compatible with the eMule or aMule ones.

The large number of users involved in current P2P networks and the lack of a central authority that certificates the trust of the participating nodes imply that the system must be able to operate even though some participants are malicious. DHT security, in particular, the problem of ensuring efficient and correct peer discovery despite adversarial interference, has been addressed in a number of works [19, 22, 11]. However, the majority of these studies examine the types of problems, drawing examples from existing systems, or experimentally evaluate the attacks over the networks. Despite the great success formal methods have had in the analysis of distributed networks and protocols, their contribution to P2P networks is scarce. In [9], Mühl gives formal semantics of publish/subscribe systems based on sequential traces using the syntax of linear temporal logic. The work formalizes and studies the correctness of several routing configurations: flooding, simple routing, identity-based routing, ... However, it does not include DHT based routing algorithms. Borgström et al. in [3], prove correctness of the lookup operation of the DHT-based DKS system, developed in the context of the EU-project [7], for a static model of the network using value-passing CCS. Finally, Bakhshi and Gurov [2] give a formal verification of Chord's stabilization algorithm using the π-calculus. But, as it is said in [11], the question is whether the P2P approach is mature enough to step outside of its *comfort zone* of file sharing and related applications. In particular, not much is known about the ability of DHTs to meet critical security requirements (as those required nowadays, e.g., for domain name servers) and its ability to withstand attacks.

Our goal is to study the possibilities offered by formal methods to prove the correctness of the dynamic aspects of P2P networks and find possible attacks to them. We start with the Kademlia network, as it is the one already implemented and in use, and focus our work on the routing algorithms. We use the initial description of the Kademlia DHT [8] and fill some open issues with the eMule real implementation. See [10] for a thorough analysis of the source code of eMule version 0.47a and [6] for the source code (version 0.50a). We are using the Maude formal specification language based on rewriting logic [4, 12] as it has been successfully applied in similar problems, like network communication protocol analysis [21] and it offers simple an elegant time simulation resources. This work has been published in [15, 14]. In this technical report we focus on the implementation details of the centralized prototype.

The paper is organized as follows: Section 2 gives a short overview of the Kademlia DHT, focused on the aspects we have considered for the moment. Next, we explain the formalization of the different parts of the network and the interaction among them. Then, we introduce the notion of time and show the formalization of the processes of looking for a file and publishing files. Finally some open issues are outlined.

2 The Kademlia DHT

Nodes in a P2P network realize two basic tasks: they put their files at the disposal of other users and access the files shared by the others. The networks that use a DHT table have similar approaches for solving these problems; they identify both nodes and files with n-bit quantities, and keep the information of shared files in the nodes with an ID *close* to the file ID. Then, the look-up algorithm is based on locating successively *closer* nodes to any desired key. The DHTs differ on the notion of *close to* they applied. In particular, Kademlia defines the distance between two IDs as the bitwise exclusive (XOR) of the n-bit quantities.

Each node stores contact information about others. 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 $i \in \{1, \ldots, n\}$ and n the ID length. In the Kademlia paper [8] 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 time last seen. 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 in the appropriate k-bucket 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 of the head of the list responds, it is moved to the tail, and the new node is discarded. This policy gives preference to old contacts, and it is due to the analysis of Gnutella data collected by Saroiu et al. [18] which states that the longer a node has been up, the more likely it is to remain up another hour.

k-buckets are organized in a binary tree called the routing table. Each k-bucket is identified by the common prefix of the IDs it contains. Internal tree nodes are the common prefix of the k-buckets, while the leaves are the k-buckets. Thus, each k-bucket covers some range of the ID space, and together the k-buckets cover the entire ID space with no overlap.

The Kademlia protocol consists of four Remote Procedure Calls (RPCs):

- `PING` probes a node to see if it is online.
- `STORE` instructs a node to store a file ID together with the contact of the node that shares the file.
- `FIND-NODE` takes an ID as argument and the recipient returns the contacts of the k nodes it knows about closest to the target ID.
- `FIND-VALUE` takes an ID as argument. If the recipient has information about the argument, it returns the contact of the node that shares the file, otherwise, it returns a list of the k contacts it knows about closest to the target.

In the following we summarize the processes of looking for a value and publishing a shared file from the Kademlia paper [8].

Looking for a value. To find a file ID, a node starts by performing a look up to find the k nodes with closest IDs to the file ID. First, the node sends a `FIND-VALUE` RPC to the α nodes it knows with an ID closer to the file ID, where α is a system concurrency parameter. As nodes reply, the initiator sends new `FIND-VALUE` RPCs to nodes it has learned about from previous RPCs, maintaining α active RPCs. Nodes that fail to respond quickly are removed from consideration. If a round of `FIND-VALUE` RPCs fails to return a node any closer than the closest already seen, the initiator resends the `FIND-VALUE` to all of the k closest nodes it has not already queried. The process terminates when any node returns the value or when the initiator has queried and gotten responses from the k closest nodes it has seen.

Publishing a shared file. Publishing is performed automatically whenever a file needs it. To maintain persistence of the data, files are published by the node that shares them every 24 hours. Nodes that know about a file publish it every hour.

To publish a file, a peer locates the k closest nodes to the key, as it is done in the *looking for a value* process, although it uses the `FIND-NODE` RPC. Once it has located the nodes, the initiator sends the first ten a `STORE` RPC.

3 Network representation

The Kademlia network is modeled as a Maude configuration of objects and messages. The objects represent the peers. The specification is defined in the `P2P-NETWORK` module in the `Kademlia.maude` file.

```
class Peer |
        RT : RTConfiguration , --- Routing table
        Files : TFileTable ,  --- Files of other nodes kept by the node
        Publish : TPublishFile , ---  Peer shared files
        SearchFiles : TSearchFile ,   ---Files the peer wants to search
        SearchList : TemporaryList{vContact-BitString} ,  --- Temporary table
        Life : TimeInf,  --- Time life of the peer. INF if it is disconnected
        Reconnect : TimeInf,  --- Time for reconnect the peer. INF if the peer is connected
        NumTimesConnected : Nat . -- For stadistics
```

where the object identification is defined by the operation

```
op peer :  X$Contact -> Oid [format (r!  o)] .
```

defined in the Maude module `KADEMLIA-PROTOCOL` (`KademliaRT.maude` file).

The attributes related to the Kademlia network are:

- `RT` keeps the information of the routing table.
- `Files` keeps the information of the files the peer is responsible for. It includes the files ID and the identification of the peer that shares the file.
- `Publish` keeps the information of the files the peer wants to share. The information includes the files ID and the file's location in the peer.
- `SearchFiles` keeps the files a peer is looking for. A peer may want to look for many files. In the current version of the prototype files are searched one by one.
- `SearchList` is a temporary list used in the search process.

The attributes used for the Maude simulation are:

- `Life`, is the time the peer will remain connected. The value is updated as time passes. When it is set to zero it means that the peer has left the network. It is set to a random value when the peer is connected.
- `Reconnect`, is the time to be connected again. It is set to a random value when a node leaves the network.
- `NumTimesConnected` keeps information about the number of times a peer has connected to the network. It is only used for information purposes.

The messages represent the RPCs. There is a message for each RPC defined in the Kademlia protocol. The first parameter of the message is the peer that receives the message, and the second parameter is of sort `TravelingContents`, defined in the `KADEMLIA-PROTOCOL` module (`KademliaRT.maude` file). The first parameter of the `TravelingContents` operations is always the peer that sends the message and the last two parameters are used to control the course of time. The last but one controls the messages that are not attended because the receiver has left the network. When a message is sent it is assigned a time, and when this time passes the message is removed from the configuration. The last parameter is the time it takes in the Real-Time-Maude system the RPC. For the time being, each RPC is assigned one time unit.

The `Msg` sort and the `TravelingContents` sort are defined in the `KADEMLIA-PROTOCOL` module (`KademliaRT.maude` file).

```
sort TravelingContents .
msg to_:_ : Oid TravelingContents -> Msg .
```

RPCs are defined in the `KADEMLIA-TRANSMITTED-SYNTAX` module (`Kademlia.maude` file), except for the `PING` RPC which is defined in the `KADEMLIA-PROTOCOL` module (`KademliaRT.maude` file).

The `PING` RPC syntax is:

```
op PING : X$Contact TimeInf TimeInf -> TravelingContents [ctor] .
op PING-REPLY : X$Contact TimeInf TimeInf -> TravelingContents [ctor] .
```

The `STORE` message has an additional parameter that represents the file ID to be stored by the node and the identification of the node that shares the file.

```
--- 1 parameter peer that sends the message
--- 2 parameter File ID to store
--- 3 parameter time to remove the message
--- 4 parameter time that passes when an RPC i send. Set to 1
op STORE : MyContact TFileTable TimeInf TimeInf -> TravelingContents [ctor] .
op STORE-REPLY : MyContact BitString TimeInf TimeInf -> TravelingContents [ctor] .
```

The `FIND-NODE` message has an additional parameter that represents the key the sender is looking for. The reply has another additional parameter that keeps a list of the k nodes the peer knows about closest to the target, where k is the bucket dimension. The information is obtained from the routing table of the node that receives the RPC.

```
--- 1 parameter peer that sends the message
--- 2 parameter peer the sender is looking for
--- 3 parameter time to remove the message
--- 4 parameter time that passes when an RPC i send. Set to 1
op FIND-NODE : MyContact BitString TimeInf TimeInf -> TravelingContents [ctor] .
op FIND-NODE-REPLY : MyContact BitString Set{vCONTACT}{vContact-BitString} TimeInf
                     TimeInf -> TravelingContents [ctor] .
```

The `FIND-VALUE` message has an additional parameter that represents the file ID the sender is looking for. The message has two possible replies. If the receiver has information about the file in its `Files` table it returns the contact of the node that shares the file. If the receiver has not information about the file, it returns the closest nodes to the file ID, like the `FIND-NODE` message.

```
--- 1 parameter peer that sends the message
--- 2 parameter file the sender is looking for
--- 3 parameter time to remove the message
--- 4 parameter time that passes when an RPC i send. Set to 1
op FIND-VALUE : MyContact BitString TimeInf TimeInf -> TravelingContents [ctor] .

op FIND-VALUE-REPLY1 : MyContact BitString Set{vCONTACT}{vContact-BitString} TimeInf
                       TimeInf -> TravelingContents [ctor] .

--- 3 parameter. searched file
--- 4 parameter. peer that stores the file
op FIND-VALUE-REPLY2 : MyContact BitString BitString TimeInf TimeInf ->
                       TravelingContents [ctor] .
```

3.1 The routing table

Although the routing table is depicted in [8] as a binary tree, it can be represented as a list of k-buckets since for each internal tree node the subtree whose prefix does not match with the peer ID is a leave. For the same reason it is not worth representing it as a trie ADT. The k-bucket's position in the list is given by its prefix so looking for a k-bucket is done sequentially following the prefix. The steps are the same as if we were looking for it in the tree. Although it is proposed in [8] a routing table optimization that allows more contacts for IDs *close* to the peer ID, we have not considered it in the specification. Nevertheless we expect it will not be necessary to build a complete binary tree. The eMule routing table [10] also has more k-buckets in each node than the routing table considered in [8], since the subtree whose prefix does not match with the peer ID may be a semi-complete tree of height four. Again the modification is local and bounded so we expect to find a more efficient representation than a binary tree. Modules about the routing table are located in the `KademliaRT.maude` file.

3.1.1 k-buckets representation

Kademlia buckets are defined in the `BUCKET` module (`KademliaRT.maude` file). They 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`. Contacts are added by the right hand side of the list, called the tail, while the first contact is the one placed on the left hand side, so our list really behaves like a queue. We also have a subsort, called `NeBucket{X}`, that 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 following membership axiom.

```
var T : X$Contact .  var B : Bucket{X} .

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

where the `bucketDim` constant is also defined in this module.

```
op bucketDim : -> NzNat .
eq bucketDim = 3 .
```

We define operations to perform all the bucket's functions:

1. **Compute the number of contacts in a bucket.**

   ```
   op length-b : Bucket{X} -> Nat .
   var T : X$Contact .  var B : Bucket{X} .
   eq [num1] : length-b(empty-bucket) = 0 .
   eq [num2] : length-b(B ! T) = 1 + length-b(B) .
   ```

2. **Remove a contact from the bucket.** If the contact is in the bucket it is removed, in other case the operation has no effect.

   ```
   op rem-contact-b : X$Contact Bucket{X} -> Bucket{X} .
   vars T T1 T2 : X$Contact .  var B : Bucket{X} .
   eq [remove1] : rem-contact-b(T,empty-bucket) = empty-bucket .
   eq [remove2] : rem-contact-b(T, B ! T) = B .
   ceq [remove3] : rem-contact-b(T1, B ! T2) = rem-contact-b(T1,B) ! T2
     if not equal(T1,T2) .
   ```

3. **Ask if a contact is in a bucket.**

   ```
   op _in_ : X$Contact Bucket{X} -> Bool .
   vars T T1 T2 : X$Contact .  var B : Bucket{X} .
   eq [in1] : T in empty-bucket = false .
   eq [in2] : T1 in (B ! T2) = equal(T1,T2) or T1 in B .
   ```

4. **Move a contact to the bucket's tail**. If the contact is in the bucket it is moved to the bucket's tail. If it is not in the bucket, the action has no effect.

```
op move-tail-b : X$Contact Bucket{X} -> Bucket{X} .
var T : X$Contact .  var B : Bucket{X} .
ceq [move1] : move-tail-b(T,B) = rem-contact-b(T,B) ! T if T in B .
ceq [move2] : move-tail-b(T,B) = B if not T in B .
```

5. **Ask for the first contact of a bucket.** Obtains the left contact, the one that we have not hear of for more time. The operation is not defined for the empty bucket.

```
op first-contact : NeBucket{X} -> X$Contact .
var T : X$Contact .  var NeB : NeBucket{X} .
eq [first1] : first-contact(empty-bucket ! T) = T .
eq [first2] : first-contact(NeB ! T) = first-contact(NeB) .
```

6. **Ask if the bucket is full.**

```
op full-bucket? : Bucket{X} -> Bool .
var B : Bucket{X} .
eq [full1] : full-bucket?(B) = length-b(B) == bucketDim .
```

7. **Ask if the bucket is empty.**

```
op empty-bucket? : Bucket{X} -> Bool .
var B : Bucket{X} .
 eq [empty1] : empty-bucket?(B) = length-b(B) == 0 .
```

8. **Checks if the contacts in a bucket have a common prefix.** The prefix is given by the first `Nz` bits of the `prefix` parameter.

```
op fix-bucket? : Bucket{X} NzNat BitString -> Bool .
var Nz : NzNat .  var prefix : BitString .    var T : X$Contact .
var B : Bucket{X} .
eq [fix1] : fix-bucket?(empty-bucket, Nz, prefix) = true .
eq [fix2] : fix-bucket?(B ! T, 1, prefix) =
     not equal(first(get-ID(T)),first(prefix)) and fix-bucket?(B,1,prefix) .
ceq [fix3] : fix-bucket?(B ! T, Nz, prefix) = fix-contact?(T,sd(Nz,1),prefix) and
           not equal(NBit(T,Nz),NBit(prefix,Nz)) and fix-bucket?(B,Nz,prefix)
  if Nz > 1 .
```

9. **Checks if the contacts in the last bucket have a common prefix.** The prefix is given by the first `Nz` bits of the `prefix` parameter.

```
op fix-last-bucket? : Bucket{X} NzNat BitString -> Bool .
var Nz : NzNat .  var prefix : BitString .    var T : X$Contact .
var B : Bucket{X} .
eq [fix-last1] : fix-last-bucket?(empty-bucket, Nz, prefix) = true .
eq [fix-last2] : fix-last-bucket?(B ! T, Nz, prefix) =
               fix-contact?(T,Nz,prefix) and fix-last-bucket?(B,Nz,prefix) .
```

10. **Add a contact to a bucket.** If the contact is already in the bucket, it is moved to the tail (`add1`). In other case, if the bucket is not full the new contact is added to the tail (`add4`). If the bucket is full and the contact is not in the bucket, we have to make place for the new contact. A signal should have been sent to the first contact in the bucket, the one that we have not hear for more time. If that contact responded to the signal, it is placed in the tail of the bucket and the new one is not added(`add2`). In other case it is discharged and the new contact is added to the tail (`add3`).

```
op add-contact : X$Contact Bucket{X} Bool -> Bucket{X} .
var T : X$Contact .  var B : Bucket{X} .   var on : Bool .
ceq [add1] : add-contact(T,B,on) = rem-contact-b(T,B) ! T  if T in B .
ceq [add2] : add-contact(T,B,on) =
             rem-contact-b(first-contact(B),B) ! first-contact(B)
  if full-bucket?(B) and not T in B and on .
ceq [add3] : add-contact(T,B,on) = rem-contact-b(first-contact(B),B) ! T
  if full-bucket?(B) and not T in B and not on .
ceq [add4] : add-contact(T,B,on) = B ! T if not full-bucket?(B) and not T in B .
```

11. **Compute the minimum distance between a bitstring and the contacts of a bucket.** The specification makes use of the `get-ID` and the `distance` operations defined in the `CONTACT` theory (`KademliaRT.maude` file). The first one gives the bitstring related to a contact, and the second one computes de XOR distance between two bitstrings. The operation is not defined for empty buckets.

```
op distance-min-b : BitString NeBucket{X} -> Nat .
var T : X$Contact . var BS : BitString .  var NeB : NeBucket{X} .
 eq [distance1] : distance-min-b(BS,empty-bucket ! T) = distance(BS,get-ID(T)) .
 ceq [distance2] : distance-min-b(BS, (NeB ! T)) = distance(BS,get-ID(T))
    if distance(BS,get-ID(T)) <= distance-min-b(BS,NeB) .
 ceq [distance3] : distance-min-b(BS, (NeB ! T)) = distance-min-b(BS,NeB)
    if distance(BS,get-ID(T)) > distance-min-b(BS,NeB) .
```

12. **Compute the nearest contact to a given bitstring.** The specification makes use of the `get-ID` and the `distance` operations defined in the `CONTACT` theory (`KademliaRT.maude` file). The operation is not defined for empty buckets.

```
op nearest-b : BitString NeBucket{X} -> X$Contact .
var T : X$Contact . var BS : BitString .  var NeB : NeBucket{X} .
eq [near1] : nearest-b(BS, empty-bucket ! T) = T .
ceq [near2] : nearest-b(BS, (NeB ! T)) = T
  if distance(BS,get-ID(T)) <= distance-min-b(BS,NeB) .
ceq [near3] : nearest-b(BS, (NeB ! T)) = nearest-b(BS,NeB)
  if distance(BS,get-ID(T)) > distance-min-b(BS,NeB) .
```

13. **Computes the set of the N closest contacts to the given bitstring.**

```
 op closestN : BitString Bucket{X} NzNat -> Set{vCONTACT}{X} .
 var T : X$Contact . var BS : BitString .  var NeB : NeBucket{X} .
 var Nz : NzNat .
 eq [clos0] : closestN(BS,empty-bucket,Nz) = empty .
 eq [clos1] : closestN(BS,NeB,1) = nearest-b(BS,NeB) .
 ceq [clos2] : closestN(BS,NeB,Nz) =
               insert(T,closestN(BS,rem-contact-b(T,NeB),sd(Nz,1)))
   if T := nearest-b(BS,NeB) /\ Nz > 1 .
```

14. **Checks if two buckets have the same contacts.** Contacts must be in the same order in the buckets.

```
op equal : Bucket{X} Bucket{X} -> Bool .
vars T1 T2 : X$Contact .  vars B B1 B2 : Bucket{X} .
eq equal(empty-bucket,B) = empty-bucket?(B) .
eq equal(B,empty-bucket) = empty-bucket?(B) .
eq equal(B1 ! T1, B2 ! T2) = equal(T1,T2) and equal(B1,B2) .
```

3.1.2 Routing table configuration

The routing table configuration is defined to encapsulate the behavior of the routing table inside the peer. It consists on a routing table and four items to manage the `PING` messages that are sent/received to/from the routing table. The first item is the contact of the peer the `PING` message is sent to. The second one is the message, the third one is a contact list and the last one is the time the peer will wait for the response.

```
sort RTConfiguration .
op _+_+_+_+_ : RoutingTable{X} X$Contact+ MayBeMessage ContactList{X} TimeInf
   -> RTConfiguration [ctor] .
```

3.1.3 Routing table representation

Routing tables are defined in the module `ROUTING-TABLE` (`KademliaRT.maude` file). It is defined as a list of buckets. The first bucket of the list is the one that less fits with the peer contact bitstring and the last bucket is the one that almost fits with it. Routing tables have at least one bucket that cannot be empty, and at most as many buckets as the length of the bitstring defined in the contact parameter. We also have a super sort, called `KRoutingTable{X}`, to represent erroneous routing tables. Routing tables are defined by means of a membership axiom.

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

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

cmb KR : RoutingTable{X}
      if num-buckets(KR) > 0 /\
         num-buckets(KR) <= length(give-contact(KR)) /\
         atLeastOne?(KR) /\
         is-RT(KR,1,peer-prefix(KR)) .
```

We use the following functions to define the routing table:

1. **Get the number of buckets in a routing table.**

   ```
   op num-buckets : KRoutingTable{X} -> NzNat .
   var B : Bucket{X} .  var KR : KRoutingTable{X} .
   eq [num-buckets1] : num-buckets(B) = 1 .
   eq [num-buckets2] : num-buckets(B !! KR) = 1 + num-buckets(KR) .
   ```

2. **Get the last bucket of a routing table.**

   ```
   op last-bucket : KRoutingTable{X} -> Bucket{X} .
   var B : Bucket{X} .  var KR : KRoutingTable{X} .
   eq [last1] : last-bucket(B) = B .
   eq [last2] : last-bucket(B !! KR) = last-bucket(KR) .
   ```

3. **Get the last bucket of a routing table.** It is defined for routing tables with more than one bucket.

   ```
   op next-last-bucket : KRoutingTable{X} -> Bucket{X} .
   var B : Bucket{X} .  var KR : KRoutingTable{X} .
   eq [nlast1] : next-last-bucket(B1 !! B2) = B1 .
   eq [nlast2] : next-last-bucket(B1 !! (B2 !! KR)) = next-last-bucket(B2 !! KR) .
   ```

4. **Obtain one contact of the routing table.** It obtains a contact from the routing table. It is not specified which contact it is. The operation is based in the fact that either the last bucket or the next to the last one should be non-empty. Notice that since the routing tables should have at least one It is used to compute the length of the bitstrings used for the contacts parameters.

```
op give-contact : KRoutingTable{X} -> X$Contact .
var KR : KRoutingTable{X} .
ceq [contact1] : give-contact(KR) = first-contact(last-bucket(KR))
   if not empty-bucket?(last-bucket(KR)) .
ceq [contact2] : give-contact(KR) = first-contact(next-last-bucket(KR))
   if empty-bucket?(last-bucket(KR)) .
```

5. **Check whether there is at least a contact in the last bucket or in the next to the last one.**

```
op atLeastOne? : KRoutingTable{X}  -> Bool .
var KR : KRoutingTable{X} .
ceq [atLeastOne?] : atLeastOne?(KR) = not empty-bucket?(last-bucket(KR)) or
                                      not empty-bucket?(next-last-bucket(KR))
  if num-buckets(KR) > 1 .
```

6. **Check whether the given structure is a routing table.** Checks that the contacts in each bucket are appropriate. It uses the operation `fix-bucket` that checks the contacts in a bucket. Parameter `Nz` is used to control the bit of the `prefix` bitstring that is used in each bucket. Initially should be 1.

```
op is-RT : KRoutingTable{X} NzNat BitString -> Bool .
vars B B1 B2 : Bucket{X} . var Nz : NzNat . var prefix : BitString .
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) .
```

7. **Obtain the bitstring's prefix of the owner of a routing table.** It looks to the last or next to the last bucket of the routing table. It gets as many bits as there are buckets in the routing table. The operation is not defined for routing tables with less than two buckets.

```
op peer-prefix : KRoutingTable{X} -> BitString .
var KR : KRoutingTable{X} .
ceq [prefix1] : peer-prefix(KR) =
    getPrefix(first-contact(last-bucket(KR)),sd(num-buckets(KR),1))
     if num-buckets(KR) > 1 /\ not empty-bucket?(last-bucket(KR)) .
ceq [prefix2] : peer-prefix(KR) =
    compN(getPrefix(first-contact(next-last-bucket(KR)),sd(num-buckets(KR),1)))
     if num-buckets(KR) > 1 /\ empty-bucket?(last-bucket(KR)) .
```

8. **Obtain the nth bucket of a routing table.**

```
op Nbucket : KRoutingTable{X} NzNat -> Bucket{X} .
var KR : KRoutingTable{X} .  var B : Bucket{X} .
var Nz : NzNat .
  eq [Nbucket1] : Nbucket(B,1) = B .
  eq [Nbucket2] : Nbucket(B !! KR,1) = B .
  ceq [Nbucket3] : Nbucket(B !! KR,Nz) = Nbucket(KR,sd(Nz,1))
     if (Nz > 1) /\ Nz <= num-buckets(B !! KR) .
```

9. **Remove a contact from a routing table.** There can be empty buckets in the table, but the operation ensures that one of the last bucket or the next to the last one should not be empty. It uses the `rem-contact-b` operation of the buckets.

```
op rem-contact-rt : X$Contact KRoutingTable{X} -> RoutingTable{X} .
var Z : X$Contact .    var B : Bucket{X} .   var KR : KRoutingTable{X} .
eq [elim1] : rem-contact-rt(Z,B) = rem-contact-b(Z,B) .
ceq [elim2] : rem-contact-rt(Z, B !! KR) = rem-contact-b(Z,B) !! KR
    if  Z in B /\ (num-nodes-table(KR) > 0 or length-b(B) > 1) .
ceq [elim3] : rem-contact-rt(Z, B !! KR) = KR
      if  Z in B /\ length-b(B) == 1 /\ num-nodes-table(KR) == 0 .
ceq [elim4] : rem-contact-rt(Z, B !! KR) = B !! rem-contact-rt(Z,KR)
      if  not Z in B /\ (length-b(B) > 0 or num-nodes-table(KR) > 1) .
ceq [elim5] : rem-contact-rt(Z, B !! KR) = rem-contact-rt(Z,KR)
      if  not Z in B /\ length-b(B) == 0 /\ num-nodes-table(KR) == 1 .
```

10. **Count the number of contacts in the routing table**. It uses the `length-b` operation to count the contact on a bucket.

```
  op num-nodes-table : KRoutingTable{X} -> NzNat .
  var B : Bucket{X} .   var KR : KRoutingTable{X} .
  eq [nnt1] : num-nodes-table(B) = length-b(B) .
  eq [nnt2] : num-nodes-table(B !! KR) = length-b(B) + num-nodes-table(KR) .
```

The routing table defines two operations, one to add elements to the table and the other to compute the closest nodes to a given bitstring in order to find the peer that keeps some information.

1. **Add a contact to a routing table.**

   Adding a contact to a full routing table requires sending messages to old contacts to verify if they are alive, in order to keep or remove them. For this reason, the result of the `add-entry` operation is of sort `RTConfiguration`. The `add-entry2` operation is used when the peer receives the answer to the requesting message and knows if the contact is alive or not.

```
op add-entry : X$Contact RoutingTable{X} X$Contact ContactList{X} ->
               RTConfiguration .
vars Z1 Z2 : X$Contact .    var R : RoutingTable{X} .    var L : ContactList{X} .
var B : Bucket{X} .
  --- Bucket not full. New contact added at the tail
  ceq [add0] : add-entry(Z1,R,Z2,L) =
      add-entry-aux(Z1,R,1,Z2,true) + noneContact + noneMessage + L + INF
      if B := find-bucket(get-ID(Z1),R) /\ not full-bucket?(B) /\
                not Z1 in B /\ not equal(Z1,Z2) .

  --- Bucket full. Ask if first bucket contact is on
  ceq [add1] : add-entry(Z1,R,Z2, L) =
      R + Z1 +
      PING(Z2,first-contact(find-bucket(get-ID(Z1),R)),RPCRemove,1) + L + INF
```

```
      if B := find-bucket(get-ID(Z1),R) /\ full-bucket?(B) /\
         not Z1 in B /\ not equal(Z1,Z2) .

  --- The contact is already in the bucket. Do nothing
  ceq [add2] : add-entry(Z1,R,Z2, L) = R + noneContact + noneMessage + L + INF
      if B := find-bucket(get-ID(Z1),R) /\ Z1 in B /\ not equal(Z1,Z2) .

  --- new contact is the routing table owner. Do nothing
  ceq [add3] : add-entry(Z1,R,Z2, L) = R + noneContact + noneMessage + L + INF
      if equal(Z1,Z2) .

  --- First bucket contact off
  op add-entry2 : X$Contact RoutingTable{X} X$Contact ContactList{X} Bool ->
                  RTConfiguration .
vars Z1 Z2 : X$Contact .    var R : RoutingTable{X} .    var L : ContactList{X} .
var B : Bucket{X} .
--- First bucket contact off. Remove it
eq [add4] : add-entry2(Z1,R,Z2, L, false) =
    add-entry-aux(Z1,R,1,Z2,false) + noneContact + noneMessage + L + INF .

--- First bucket contact on. Not split bucket
ceq [add5] : add-entry2(Z1,R,Z2, L, true) =
         add-entry-aux(Z1,R,1,Z2,true) + noneContact + noneMessage + L + INF
if not isLastBucket?(find-bucket(get-ID(Z1),R),R) .

--- First bucket contact on. Split bucket
ceq [add6] : add-entry2(Z1,R,Z2, L, true) =
    add-entry(Z1,conc(R,div-bucket(last-bucket(R),Nz,NBit(Z2,Nz))),Z2, L)
if  Nz := num-buckets(R) /\ isLastBucket?(find-bucket(get-ID(Z1),R),R) .
```

2. **Finds the closest nodes to a given one** The operation may have to look on several buckets.

```
  --- 1 param. key (node or file) to be compared
  --- 2 param. routing table to obtain the closest nodes of the first parameter
  --- 3 param. number of closest nodes required
  op closest-nodes : BitString RoutingTable{X} NzNat -> Set{vCONTACT}{X} .
  op closest-nodes-aux : BitString RoutingTable{X} NzNat -> Set{vCONTACT}{X} .
  var BS : BitString .    var R : RoutingTable{X} .
  var Nz : NzNat .    var Z : X$Contact .
  ceq [closest1] : closest-nodes(BS,R,Nz) = closestN(BS,find-bucket(BS,R),Nz)
     if length-b(find-bucket(BS,R)) >= Nz .
  ceq [closest2] : closest-nodes(BS,R,Nz) = closest-nodes-aux(BS,R,Nz)
     if length-b(find-bucket(BS,R)) < Nz .

  ceq [closest3] : closest-nodes-aux(BS,R,Nz) = nearest-rt(BS,R)
       if Nz == 1 or num-nodes-table(R) == 1 .
  ceq [closest4] : closest-nodes-aux(BS,R,Nz) =
        insert(Z,closest-nodes-aux(BS,rem-contact-rt(Z,R),sd(Nz,1)))
     if Z := nearest-rt(BS,R) /\ Nz > 1 /\ num-nodes-table(R) > 1 .
```

We use the following operations in the specification of the `add-entry` operation:

1. **Add a contact to a routing table**. The operation looks for the bucket in which it should be added the contact. It uses the operation `add-contact` of the buckets, and the operation

NBit of the contacts. The operation receives a flag to distinguish the case in which it should remove the first contact of the bucket or not.

```
op add-entry-aux : X$Contact KRoutingTable{X} NzNat X$Contact Bool ->
                   RoutingTable{X} .
vars Z1 Z2 : X$Contact .    var KR : KRoutingTable{X} .
var B : Bucket{X} .    var Nz : NzNat .
eq [add-aux1] : add-entry-aux(Z1,B,Nz,Z2,on) = add-contact(Z1,B,on) .
ceq [add-aux2] : add-entry-aux(Z1,B !! KR,Nz,Z2,on) =
               add-contact(Z1,B,on) !! KR
  if not equal(NBit(Z1,Nz),NBit(Z2,Nz)) .
ceq [add-aux3] : add-entry-aux(Z1,B !! KR,Nz,Z2,on) =
               B !! add-entry-aux(Z1,KR,s Nz,Z2,on)
  if equal(NBit(Z1,Nz),NBit(Z2,Nz))  .
```

2. **Check whether a given bucket is the last one of a routing table.**

```
op isLastBucket? : Bucket{X} KRoutingTable{X} -> Bool .
vars B B1 B2 : Bucket{X} .    var KR : KRoutingTable{X} .
eq [iLB1] : isLastBucket?(B,B) = true .
ceq [iLB2] : isLastBucket?(B1,B2) = false if not equal(B1,B2) .
eq [iLB3] : isLastBucket?(B,B1 !! KR) = isLastBucket?(B,KR) .
```

3. **Split a full bucket.**

```
op div-bucket : Bucket{X} NzNat Bit  -> KRoutingTable{X} .
op div-bucket-aux : Bucket{X} Bucket{X} Bucket{X} NzNat Bit  ->
                    KRoutingTable{X} .
vars B B1 B2 : Bucket{X} .    var Nz : NzNat .    var Z : X$Contact .
var bi : Bit .
eq [div] : div-bucket(B,Nz,bi) =
           div-bucket-aux(B,empty-bucket,empty-bucket, Nz,bi) .
ceq [div1] : div-bucket-aux(B ! Z,B1,B2,Nz,bi) =
           div-bucket-aux(B,B1,(B2 ! Z),Nz,bi)
   if equal(NBit(Z,Nz),bi) .
ceq [div2] : div-bucket-aux(B ! Z,B1,B2,Nz,bi) =
           div-bucket-aux(B,(B1 ! Z),B2,Nz,bi)
   if not equal(NBit(Z,Nz),bi) .
eq [div3] : div-bucket-aux(empty-bucket,B1,B2,Nz,bi) = B1 !! B2 .
```

4. **Change the last bucket of a routing table for a routing table of two buckets obtained by splitting the last bucket.**

```
op conc : KRoutingTable{X} KRoutingTable{X} -> KRoutingTable{X} .
var B : Bucket{X} .    vars KR1 KR2 : KRoutingTable{X} .
eq [conc1] : conc(B,KR1) = KR1 .
eq [conc2] : conc(B !! KR1, KR2) = B !! conc(KR1, KR2) .
```

5. **Find the bucket that contains a given contact.**

```
op find-bucket : BitString RoutingTable{X} -> Bucket{X} .
  var BS : BitString .    var R : RoutingTable{X} .
ceq [findB1] : find-bucket(BS,R) = last-bucket(R)
    if firstBitDiff(BS,peer-prefix(R)) == num-buckets(R) .
ceq [findB2] : find-bucket(BS,R) = Nbucket(R,firstBitDiff(BS,peer-prefix(R)))
    if firstBitDiff(BS,peer-prefix(R)) < num-buckets(R) .
```

We use the following operations in the specification of the `closest-nodes` operation:

1. **Find the bucket where the closest nodes to a bitstring are located.**

```
op nearest-rt : BitString KRoutingTable{X} -> X$Contact .
var BS : BitString .    var NeB : NeBucket{X} .    var B : Bucket{X} .
var KR : KRoutingTable{X} .
  eq [near1] : nearest-rt(BS,NeB) = nearest-b(BS,NeB) .
  eq [near2] : nearest-rt(BS, empty-bucket !! KR) = nearest-rt(BS,KR) .
  eq [near3] : nearest-rt(BS, NeB !! empty-bucket) = nearest-b(BS,NeB) .
  ceq [near4] : nearest-rt(BS, NeB !! KR) = nearest-b(BS,NeB)
      if KR =/= empty-bucket /\ distance-min-b(BS,NeB) <= distance-min-rt(BS,KR) .
  ceq [near5] : nearest-rt(BS, NeB !! KR) = nearest-rt(BS,KR)
      if KR =/= empty-bucket /\ distance-min-b(BS,NeB) > distance-min-rt(BS,KR) .
```

2. **Compute the minimum distance from a bitstring to the closest contact in the routing table.**

```
op distance-min-rt : BitString KRoutingTable{X} -> Nat .
var BS : BitString .    var NeB : NeBucket{X} .    var B : Bucket{X} .
var KR : KRoutingTable{X} .
eq [distance1] : distance-min-rt(BS,NeB) = distance-min-b(BS,NeB) .
eq [distance2] : distance-min-rt(BS,empty-bucket !! KR) = distance-min-rt(BS,KR) .
eq [distance3] : distance-min-rt(BS,NeB !! empty-bucket) = distance-min-b(BS,NeB) .
ceq [distance4] : distance-min-rt(BS,NeB !! KR) =
             min(distance-min-b(BS,NeB),distance-min-rt(BS,KR))
      if KR =/= empty-bucket .
```

3.1.4 About contacts

The system is parameterized with respect to the contact information. We define the `CONTACT` theory in the `KademliaRT.maude` file. The theory defines a sort `Contact` and a supersort `Contact+` that adds a constant `noneContact` to the values of the sort `Contact`. The operations defined on contacts are:

1. `op distance : Contact Contact -> Nat .`, obtains the distance between two contacts.
2. `op NBit : Contact NzNat -> Bit .`, obtains bit number n of the contact ID.
3. `op fix-contact? : Contact NzNat BitString -> Bool .`, checks if the first n bits of the bitstring fix with the first n bits of the contact.
4. `op getPrefix : Contact NzNat -> BitString .`, Obtains the first n bits of the bitstring.
5. `op get-ID : Contact -> BitString .`, Obtains the contact ID.
6. `op length : Contact -> NzNat . `, Obtains the number of bits of a bitstring.
7. `op equal : Contact Contact -> Bool .`, checks if two contacts are equal.

We define a module `MYCONTACT` (`KademliaRT.maude` file) to instantiate the routing table. The contact is defined by means of the constructor `c` as a bitstring. The operations defined in the module to instantiate the theory are based on the operations defined on bitstrings:

1. **Calculate the distance between two contacts.**

```
op distance : MyContact MyContact -> Nat .
vars c1 c2 : MyContact .
eq distance(c1,c2) = distance(get-ID(c1), get-ID(c2)) .
```

2. **Obtain bit number n of the contact ID.**

```
op NBit : MyContact NzNat -> Bit .
var c1 : MyContact .
var N : NzNat .
eq NBit(c1, N) = NBit(get-ID(c1),N) .
```

3. **Check whether the first n bits of the bitstring fix with the first n bits of the contact.**

```
op fix-contact? : MyContact NzNat BitString -> Bool .
var c1 : MyContact .  var N : NzNat .  var S : BitString .
eq fix-contact?(c1,N,S) = fixNBits(get-ID(c1),N,S) .
```

4. **Obtain the first n bits of the bitstring.**

```
op getPrefix : MyContact NzNat -> BitString .
var c1 : MyContact .  var N : NzNat .
eq getPrefix(c1,N) = getPrefix(get-ID(c1),N) .
```

5. **Obtain the contact ID.**

```
op get-ID : MyContact -> BitString .
var S : BitString .
eq get-ID(c(S)) = S .
```

6. **Obtain the number of bits of a bitstring.**

```
op length : MyContact -> NzNat .
var c1 : MyContact .
eq length(c1) = length(get-ID(c1)) .
```

7. **Check whether two contacts are equal.**

```
op equal : MyContact MyContact -> Bool .
vars c1 c2 : MyContact .
eq equal(c1,c2) = equal(get-ID(c1),get-ID(c2)) .
```

The module `BIT-STRING` (`BitString.maude` file) defines the sorts `BitString`, `BitString128` and `BitString160`. A bitstring is a bit or a sequence of bits:

```
sort BitString .
sort BitString128 .
sort BitString160 .
subsort Bit < BitString128 BitString160 < BitString .
op _;_ : BitString Bit -> BitString [ctor] .
```

The sort `Bit` is defined in the `BIT` module (`BitString.maude` file).

```
sort Bit .
op 0 : -> Bit [ctor] .
op 1 : -> Bit [ctor] .
op equal : Bit Bit -> Bool .
var B : Bit .
eq equal(B,B) = true .
eq equal(0,1) = false .
eq equal(1,0) = false .
```

The operations on bitstrings are the following:

1. **Calculate the distance between two bitstrings.**

```
op distance : BitString BitString -> Nat .
 vars B1 B2 : Bit .   vars S1 S2 : BitString .
 eq distance(0, 0) = 0 .
 eq distance(0, 1) = 1 .
 eq distance(1, 0) = 1 .
 eq distance(1, 1) = 0 .
 eq distance(S1 ; B1, S2 ; B2) = distance(B1,B2) + 2 * distance(S1,S2) .
```

2. **Obtain the bit in the position n of the contact ID**. The first bit in the left hand side of the bitstring is in position 1.

```
op NBit : BitString NzNat -> Bit .
var B : Bit .   vars S : BitString .   var Nz : NzNat .
eq NBit(B,1) = B .
ceq NBit(S ; B, Nz) = B if Nz == length(S ; B) .
ceq NBit(S ; B, Nz) = NBit(S,Nz) if Nz < length(S ; B) .
```

3. **Obtain the number of bits of a bitstring.**

```
op length : BitString -> NzNat .
var B : Bit .   var S : BitString .
eq length(B) = 1 .
eq length(S ; B) = 1 + length(S) .
```

4. **Obtain the first n bits of the bitstring.**

```
op getPrefix : BitString NzNat -> BitString .
var B : Bit .  var S : BitString .  var Nz : NzNat .
ceq getPrefix(S ; B, Nz) = getPrefix(S, Nz) if length(S ; B) > Nz .
ceq getPrefix(S, Nz) = S if length(S) <= Nz .
```

5. **Obtain the contact ID.**

```
op get-ID : BitString -> BitString .
var S : BitString .
eq get-ID(S) = S .
```

6. **Check whether two bitstring are equal.** They have the same length and the same bits in the same positions.

```
op equal : BitString BitString -> Bool .
vars B1 B2 : Bit .  vars S1 S2 : BitString .
eq equal(S1 ; B1,S2 ; B2) = equal(B1, B2) and equal(S1, S2) .
```

We define a view `vCONTACT` used to instantiate the routing table

```
(view vContact-BitString from CONTACT to MYCONTACT+ is
   sort Contact to MyContact .
   sort Contact+ to MyContact+ .
endv)

(mod ROUTING-TABLE-BITSTRING is
  pr ROUTING-TABLE{vContact-BitString} .
endm)
```

The user can define new modules to include more information about the nodes, like the IP address and the UDP port, and instantiate the routing table with the new module.

3.2 Routing Constants

Routing constants are defined in the module `ROUTING-CONSTANTS` (`Kademlia.maude` file).

```
--- Maximum number of files in the SearchFiles Table
op FilesInSearchTable : -> Nat .
--- Maximum number of FIND-NODE RPC that can be send in parallel
op ParallelSearchRPC : -> Nat .
--- Maximum number of peers that store a publish file
op RedundantPublish : -> Nat .
--- Maximum number of peers contacted in a search
op kSearched : -> Nat .
eq FilesInSearchTable = 10 .
eq ParallelSearchRPC = 3 .
eq RedundantPublish = 3 .
eq kSearched = 10 .
```

3.3 Shared files

We use a common generic table to keep the information of the files the peer is responsible of (`Files` attribute), for the files a peer wants to share (`Publish` attribute), and for the files the peer is looking for (`SearchFiles` attribute). The specification is in the `KademliaFiles.maude` file. The generic definition of the table is:

```
(mod TABLE { X :: TRIV , Y :: TRIV+} is
  sort InfoTable{X,Y} .
  sort Table{X,Y} .
  subsort InfoTable{X,Y} < Table{X,Y} .

  op <_&_> : X$Elt Y$Elt -> InfoTable{X,Y} [ctor] .
  op empty-table : -> Table{X,Y} [ctor] .
  op _#_ : Table{X,Y} Table{X,Y} -> Table{X,Y} [assoc comm id: empty-table ctor] .
  --- If the value already exists it is changed
  op store : X$Elt Y$Elt Table{X,Y} -> Table{X,Y} .
  op _in_ : X$Elt Table{X,Y} -> Bool .
  op remove : X$Elt Table{X,Y} -> Table{X,Y} .
  op find : X$Elt Table{X,Y} -> Y$Elt .
  op key? : InfoTable{X,Y} -> X$Elt .
  op value? : InfoTable{X,Y} -> Y$Elt .
  op init-table : Table{X,Y} -> Table{X,Y} .
  op delta : Table{X,Y} Time -> Table{X,Y} [frozen (1)] .
  op delta : Y$Elt TimeInf -> Y$Elt [frozen (1)] .
  op minTime : Table{X,Y} -> TimeInf .
  --- stadistics
  op num-files : Table{X,Y} -> Nat .
endm)
```

where the first parameter, `X`, represents the table key and the second parameter, `Y`, represents the content related to a given key. Theory `TRIV+` is declared at the beginning of the `KademliaFiles.maude` file, it adds a `monus` operation, that subtracts an amount of time to the content, to the `TRIV` theory. The sort `InfoTable` represents a single entry of the table. This sort can be avoided, but we keep it to have a total definition of the operations `key?` and `value?`. The `store` operation changes the content of the key if it already exists in the table. The `remove` operation is total, it has no effect if the key is not in the table. `find` returns the content of a key, it is not defined if the key is not in the table.

3.3.1 Published files

There are three different concepts concerning shared files. On the one hand, a node shares some files. Each node has a table with information about these files, the key is the file ID, while the

value includes the file's name and the time to republish it. The Maude specification is.

```
sort KeyPublishFile .
sort InfoPublishFile .
subsort BitString < KeyPublishFile .
  op _@_ : String TimeInf -> InfoPublishFile [ctor] .
  op init-time : InfoPublishFile -> InfoPublishFile .
  op delta : InfoPublishFile Time -> InfoPublishFile [frozen (1)] .
  op getTime : InfoPublishFile -> TimeInf .
```

The table is instantiated with the following views:

```
(view KeyPublishFile from TRIV to INFO-PUBLISH-FILE is
  sort Elt to KeyPublishFile .
endv)

(view InfoPublishFile from TRIV+ to INFO-PUBLISH-FILE is
  sort Elt to InfoPublishFile .
endv)
```

The concrete table used to represent the files a peer publishes is defined in the `T-PUBLISH-FILE` module (`Kademlia.maude` file):

```
 TABLE{KeyPublishFile,InfoPublishFile} *
       (sort Table{KeyPublishFile,InfoPublishFile} to  TPublishFile) .
```

We add an operation to the table:

- **Indicates whether there is some file being publishing at the moment.** A file is being published if it's related time is set to `INF`.

  ```
  op publishing? : TPublishFile -> Bool .
  vars P1 P2 : TPublishFile .  var K : KeyPublishFile .
  var S : String .  var TM : TimeInf .
  eq publishing?(empty-table) = false .
  eq publishing?(P1 # P2) = publishing?(P1) or publishing?(P2) .
  eq publishing?(< K & (S @ TM) >) = TM == INF .
  ```

3.3.2 Files of other nodes kept in this one

On the other hand, each node keeps information of the files that have a key value *close* to its own key identification. This information includes the file ID, the ID of the node that stores the file and a time value. The information about the file ID and the node ID is used in the search process. In [8] it is not specified the number of nodes that keep information about a file, we use the value defined in the eMule paper [10] which set it to ten. The time information is used to republish the files to ensure data persistence. The table specification is:

```
sort KeyFileTable .
sort InfoFileTable .
subsort BitString < KeyFileTable .
  op _;;_ : BitString TimeInf -> InfoFileTable [ctor] .
  op init-time : InfoFileTable -> InfoFileTable .
  op first? : InfoFileTable -> BitString .
  op getTime : InfoFileTable -> TimeInf .
  op delta : InfoFileTable Time -> InfoFileTable [frozen (1)] .
```

The views to instantiate the table are:

```
(view KeyFileTable from TRIV to INFO-FILE-TABLE is
  sort Elt to KeyFileTable .
endv)

(view InfoFileTable from TRIV+ to INFO-FILE-TABLE is
  sort Elt to InfoFileTable .
endv)
```

The table is defined in the `P2P-NETWORK` module (`Kademlia.maude` file).

```
TABLE{KeyFileTable,InfoFileTable} *
     (sort Table{KeyFileTable,InfoFileTable} to TFileTable) .
```

3.3.3 Files the node is searching in the P2P network

The last task a node performs in a P2P network is searching for information. Each node keeps a table of the files a peer is looking for. The key is the file ID and the value includes the file name and the time for expiration.

```
sort KeySearchFile .
sort InfoSearchFile .
subsort BitString < KeySearchFile .
  --- name of the file
  --- Time for expiration: 0 the file has already been searched and found.
  ---  > 0 < 50 the file is ready to be searched. > 50 the file is waiting.
  op _;_ : String TimeInf -> InfoSearchFile [ctor] .
  op init-time : InfoSearchFile -> InfoSearchFile .
  op delta : InfoSearchFile Time -> InfoSearchFile [frozen (1)] .
  op getTime : InfoSearchFile -> TimeInf .
```

The views that instantiate the table are:

```
(view KeySearchFile from TRIV to INFO-SEARCH-FILE is
  sort Elt to KeySearchFile .
endv)

(view InfoSearchFile from TRIV+ to INFO-SEARCH-FILE is
  sort Elt to InfoSearchFile .
endv)
```

The table is defined in the `P2P-NETWORK` module (`kademlia.maude` file).

```
TABLE{KeySearchFile,InfoSearchFile} *
        (sort Table{KeySearchFile,InfoSearchFile} to TSearchFile,
         sort InfoTable{KeySearchFile,InfoSearchFile} to TInfoSearchFile) .
```

3.4 The temporary search list

Many processes of the Kademlia network, like the search or the publish processes, need to find some contacts with the *closest* key to an ID. As the information in the node's routing table may not include the *closest* contacts, it should be searched. Now, we follow the eMule implementation of the process. The node looks for contacts that are as near as possible to the ID and keeps them, ordered by distance to the ID, in a temporary list. The temporary list keeps the following information about each contact:

1. The contact ID.
2. The distance between the contact and the searched IDs.
3. Time for the node to be updated or removed from the list.
4. A flag that indicates the step of the process in which the node is. It can be:

(a) 0: indicates that the FIND-VALUE RPC has not been sent already.

(b) 1: indicates that the FIND-VALUE RPC has been sent.

(c) 2: indicates that the FIND-NODE-REPLY RPC has been received.

(d) 3: a store message has been sent to this node

(e) 4: the store reply message is received.

In this version of the specification we admit only one search-publish process at a time. To admit more searches we need to define a map of temporary lists to keep the information about each search. The defined sorts are:

```
sort Node-Time{X} .
sort TemporaryList{X} .
subsort Node-Time{X} < TemporaryList{X} .

op <____> : X$Contact Nat TimeInf Nat -> Node-Time{X} [ctor] .
op temp-empty : -> TemporaryList{X} [ctor] .
op insert : Node-Time{X} TemporaryList{X} -> TemporaryList{X} [ctor] .
```

The operations that help the management of the temporary list are:

1. **Create the search list.** The operation receives a set of contacts and the ID of the searched key, and iterates on the set inserting the contacts in order in the temporary list. For each contact it inserts in the list: the contact ID, the distance of the contact ID to the searched ID, a time to remove the contact from the list, given by the constant SearchListRemove defined in the TIME-CONSTANTS module (TimeFiles.maude file) and the flag set to 0.

   ```
   op create-search-list : Set{vCONTACT}{X} BitString -> TemporaryList{X} .
   var Tr : X$Contact .  var ID : BitString .  var BS : Set{vCONTACT}{X} .
   eq create-search-list(empty, ID) = temp-empty .
   eq create-search-list((Tr, BS), ID) =
     insertOrd(< Tr distance(ID,get-ID(Tr)) SearchListRemove 0 >, create-search-list(BS, ID)) .
   ```

   The specification uses the operation distance and the operation get-ID defined in the CONTACT theory (KademliaRT.maude file).

2. **Change the node flag.**

   ```
   op set-flag-process : X$Contact Time TemporaryList{X} -> TemporaryList{X} .
   op set-flag-done : X$Contact Time TemporaryList{X} -> TemporaryList{X} .
   op set-flag-store : X$Contact Time TemporaryList{X} -> TemporaryList{X} .
   op set-flag-store-reply : X$Contact Time TemporaryList{X} -> TemporaryList{X} .

   eq set-flag-done(Tr,T,temp-empty) = temp-empty .
   ceq set-flag-done(Tr,T,insert(< Tr1 n TM1 F1 >,TL)) = insert(< Tr1 n T 2 >, TL)
     if equal(Tr1,Tr) .
   ceq set-flag-done(Tr,T,insert(< Tr1 n TM1 F1 >,TL)) =
       insert(< Tr1 n TM1 F1 >, set-flag-done(Tr,T,TL))
     if not equal(Tr1,Tr) .

   eq set-flag-store(Tr,T,temp-empty) = temp-empty .
   ceq set-flag-store(Tr,T,insert(< Tr1 n TM1 F1 >,TL)) =
       insert(< Tr1 n T 3 >, TL)
     if equal(Tr1,Tr) .
   ceq set-flag-store(Tr,T,insert(< Tr1 n TM1 F1 >,TL)) =
       insert(< Tr1 n TM1 F1 >, set-flag-store(Tr,T,TL))
     if not equal(Tr1,Tr) .

   eq set-flag-store-reply(Tr,T,temp-empty) = temp-empty .
   ceq set-flag-store-reply(Tr,T,insert(< Tr1 n TM1 F1 >,TL)) = insert(< Tr1 n T 4 >, TL)
   ```

```
  if equal(Tr1,Tr) .
ceq set-flag-store-reply(Tr,T,insert(< Tr1 n TM1 F1 >,TL)) =
    insert(< Tr1 n TM1 F1 >, set-flag-store-reply(Tr,T,TL))
  if not equal(Tr1,Tr) .
```

3. **Insert a list of contacts.** During the look-up process a node (A) sent a request to other nodes (B) of the closest contacts to a given key it has in its routing table. The nodes (B) reply with a set of contacts. Then node A proceed to insert these contacts in its temporary list by means of the `insertList` operation.

   The operation receives the set of contacts, the temporary list in which they will be inserted, the contact of the sender of the message (node B), the contact of the receiver of the message (node A), and the ID of the key. For each contact in the set if it is different from node A contact, and its distance to the key is less than the distance of node A ID to the key, it is inserted in the list, in other case it is discarded. The information stored in the list is the contact, the distance of the contact to the key, the time to be removed from the list, and the flag set to 0.

```
op insertList : Set{vCONTACT}{X} TemporaryList{X} X$Contact X$Contact BitString160

ceq insertList((Tr, BS), TL, SENDER , RECEIVER , I1) =
    insertList(BS,insertOrd(< Tr distance(get-ID(Tr),I1) SearchListRemove 0 >,TL),
                   SENDER , RECEIVER , I1)
  if RECEIVER =/= Tr /\ distance(get-ID(Tr),I1) < distance(get-ID(SENDER),I1) .
ceq insertList((Tr, BS), TL, SENDER , RECEIVER , I1) =
    insertList(BS,TL, SENDER , RECEIVER , I1)
  if RECEIVER =/= Tr /\ distance(get-ID(Tr),I1) >= distance(get-ID(SENDER),I1) .
eq insertList((Tr, BS), TL, SENDER , Tr , I1) = insertList(BS,TL, SENDER , Tr , I1) .
eq insertList(empty, TL, SENDER , Tr , I1) = set-flag-done(SENDER, SearchListRemove, TL) .
```

   The specification uses: the `insertOrd` operation, defined in this module, that inserts a node in order in the list; the operations `distance` and `get-ID` defined in the `CONTACT` theory (`KademliaRT.maude` file); the `SearchListRemove` constant defined in the `TIME-CONSTANTS` module (`TimeFiles.maude` file); and `set-flag-done` defined in this module.

4. **Insert in order into the list.** If the contact is in the list the operation has no effect. It is bounded by a constant. If there are more elements the ones with a greater value are removed. The distance is less than the distance of the ID to the peer.

```
op insertOrd : Node-Time{X} TemporaryList{X} -> TemporaryList{X} .
  vars Tr1 Tr2 : X$Contact .  var TL : TemporaryList{X} .  var TM1 TM2 : TimeInf .
  vars n n1 n2 : Nat .    var NT : Node-Time{X} .   vars F1 F2 : Nat .
  eq insertOrd(NT,temp-empty) = insert(NT,temp-empty) .
  ceq insertOrd(< Tr1 n1 TM1 F1 >,insert(< Tr2 n2 TM2 F2 >, TL)) =
      insert(< Tr2 n2 TM2 F2 >,TL) if equal(Tr1,Tr2) .
  ceq insertOrd(< Tr1 n1 TM1 F1 >, insert(< Tr2 n2 TM2 F2 >, TL)) =
      insert(< Tr1 n1 TM1 F1 >, insert(< Tr2 n2 TM2 F2 >, TL))
    if not equal(Tr1,Tr2) /\  n1 <= n2 /\ length(TL) < 9 .
  ceq insertOrd(< Tr1 n1 TM1 F1 >, insert(< Tr2 n2 TM2 F2 >, TL)) =
      insert(< Tr1 n1 TM1 F1 >, insert(< Tr2 n2 TM2 F2 >, remove-last(TL)))
    if not equal(Tr1,Tr2) /\  n1 <= n2 /\ length(TL) == 9 .
  ceq insertOrd(< Tr1 n1 TM1 F1 >, insert(< Tr2 n2 TM2 F2 >, TL)) =
      insert(< Tr2 n2 TM2 F2 >, insertOrd(< Tr1 n1 TM1 F1 >, TL))
    if not equal(Tr1,Tr2) /\  n1 > n2 .
```

5. **Check whether the first k contacts have sent the RPC.** The operation receives the temporary list and the number of contacts to check k. It iterates over the first k nodes checking if the flag is greater than 1. The remove from the list time is taken into account when the flag is set to 1 or 3, because in these cases the time passes. Time does not pass for nodes with the flag set to 0, 2, or 4.

```
op first-k-done : TemporaryList{X} Nat -> Bool .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
vars n n1 : Nat .    var nz1 : NzNat .
eq first-k-done(temp-empty, n1) = true .
eq first-k-done(TL, 0) = true .
eq first-k-done(insert(< Tr n TM1 0 >, TL), nz1) = false .
eq first-k-done(insert(< Tr n 0 1 >, TL),n1) = first-k-done(TL,n1) .
ceq first-k-done(insert(< Tr n TM1 1 >, TL),nz1) = false if TM1 =/= 0 .
eq first-k-done(insert(< Tr n TM1 2 >, TL),nz1) = first-k-done(TL,sd(nz1,1)) .
eq first-k-done(insert(< Tr n 0 3 >, TL),n1) = first-k-done(TL,n1) .
ceq first-k-done(insert(< Tr n TM1 3 >, TL),nz1) = first-k-done(TL,sd(nz1,1))
       if TM1 =/= 0 .
eq first-k-done(insert(< Tr n TM1 4 >, TL),nz1) = first-k-done(TL,sd(nz1,1)) .
```

6. **Check whether all the contacts in the list have sent the `FIND-VALUE` RPC messages.** The contacts that have not sent the `FIND-VALUE` RPC message should have their flag set to 0. Since time does not pass for these nodes because they are not waiting for any answer, the time of the contact is not taken into account when the contacts are count.

```
op all-sent : TemporaryList{X} -> Bool .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
var n : Nat .  var nz1 : NzNat .
eq all-sent(temp-empty) = true .
eq all-sent(insert(< Tr n TM1 0 >, TL)) = false .
eq all-sent(insert(< Tr n TM1 nz1 >, TL)) = all-sent(TL) .
```

7. **Check whether all the contacts in the list have received the `STORE` RPC message.** The contacts that have received the `STORE` RPC message should have their flag set to 4. Since time does not pass for these nodes because they are not waiting for any answer, the time of the contact is not taken into account when the contacts are count.

```
op all-store : TemporaryList{X} -> Bool .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
vars n n1 : Nat .
eq all-store(temp-empty) = true .
eq all-store(insert(< Tr n TM1 4 >, TL)) = all-store(TL) .
ceq all-store(insert(< Tr n TM1 n1 >, TL)) = false if n1 =/= 4 .
```

8. **Return the first contact of the temporary search list that have not send the `FIND-VALUE` RPC yet.** This is a partial operation defined only when there is a contact in the list that have not sent the `FIND-VALUE` RPC yet.

```
op first-not-send : TemporaryList{X} -> X$Contact .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
var n : Nat .  var F1 : Nat .
eq first-not-send(insert(< Tr n TM1 0 >, TL)) = Tr .
ceq first-not-send(insert(< Tr n TM1 F1 >, TL)) = first-not-send(TL)
    if F1 =/= 0 .
```

9. **Return the first contact of the temporary search list that have not send the `STORE` RPC yet.** This is a partial operation defined only when there is a contact in the list that have not send the `STORE` RPC yet and is ready for doing it. These contacts have the flag set to 2.

```
op first-not-stored : TemporaryList{X} -> X$Contact .
```

```
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
var n : Nat .  var F1 : Nat .
eq first-not-stored(insert(< Tr n TM1 2 >, TL)) = Tr .
eq first-not-stored(insert(< Tr n TM1 F1 >, TL)) = first-not-stored(TL) [owise] .
```

10. **Check whether there is a contact prepared for sending the `STORE` RPC.** Contacts prepared for sending the `STORE` RPC should have their flag set to 2.

```
op prepared-stored : TemporaryList{X} -> Bool .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
var n : Nat .  var F1 : Nat .
eq prepared-stored(temp-empty) = false .
eq prepared-stored(insert(< Tr n TM1 2 >, TL)) = true .
eq prepared-stored(insert(< Tr n TM1 F1 >, TL)) = prepared-stored(TL) [owise] .
```

11. **Count the number of contacts that have sent the `FIND-VALUE` RPC and have not received the response yet.** Contacts that have sent the `FIND-VALUE` RPC, and have not received the `FIND-VALUE-REPLY` RPC have the flag set to 1. The contacts are not counted if their time to remove is set to 0.

```
op messages-in-process : TemporaryList{X} -> Nat .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
var n : Nat .  var F1 : Nat .
eq messages-in-process(temp-empty) = 0 .
ceq messages-in-process(insert(< Tr n TM1 1 >, TL)) = 1 + messages-in-process(TL)
  if TM1 > 0 .
eq messages-in-process(insert(< Tr n 0 1 >, TL)) = messages-in-process(TL) .
ceq messages-in-process(insert(< Tr n TM1 F1 >, TL)) = messages-in-process(TL)
  if F1 =/= 1 .
```

12. **Count the number of contacts that have received the `FIND-VALUE-REPLY` RPC.**

```
op number-nodes-reply : TemporaryList{X} -> Nat .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
var n : Nat .  var F1 : Nat .
eq number-nodes-reply(temp-empty) = 0 .
eq number-nodes-reply(insert(< Tr n TM1 2 >, TL)) = 1 + number-nodes-reply(TL) .
eq number-nodes-reply(insert(< Tr n TM1 F1 >, TL)) = number-nodes-reply(TL) [owise] .
```

13. **Count the number of contacts that have sent the `STORE` RPC.**

```
op number-messages-store : TemporaryList{X} -> Nat .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
var n : Nat .  var F1 : Nat .
eq number-messages-store(temp-empty) = 0 .
eq number-messages-store(insert(< Tr n TM1 3 >, TL)) = 1 + number-messages-store(TL) .
ceq number-messages-store(insert(< Tr n TM1 F1 >, TL)) = number-messages-store(TL)
  if F1 =/= 3 .
```

14. **Count the number of contacts that have received the `STORE` RPC.** Contacts that have received the `STORE` RPC message should have their flag set to 4.

```
op number-messages-store-reply : TemporaryList{X} -> Nat .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
var n : Nat .  var F1 : Nat .
eq number-messages-store-reply(temp-empty) = 0 .
eq number-messages-store-reply(insert(< Tr n TM1 4 >, TL)) =
```

```
      1 + number-messages-store-reply(TL) .
ceq number-messages-store-reply(insert(< Tr n TM1 F1 >, TL)) =
      number-messages-store-reply(TL)
    if F1 =/= 4 .
```

15. **Remove a contact from the list.**

```
op remove : X$Contact TemporaryList{X} -> TemporaryList{X} .
vars Tr1 Tr2 : X$Contact .  var TL : TemporaryList{X} .
vars TM1 TM2 : TimeInf . var n : Nat .  vars F1 F2 : Nat .
eq remove(Tr1, temp-empty) = temp-empty .
ceq remove(Tr1, insert(< Tr2 n TM1 F1 >,TL)) =  TL if equal(Tr1,Tr2) .
ceq remove(Tr1,insert(< Tr2 n TM2 F2 >,TL)) =
    insert(< Tr2 n TM2 F2 >, remove(Tr1,TL))
  if not equal(Tr1,Tr2) .
```

16. **Return the number of nodes of the temporary list.**

```
  op length : TemporaryList{X} -> Nat .
  var TL : TemporaryList{X} .  var NT : Node-Time{X} .
  eq length(temp-empty) = 0 .
  eq length(insert(NT, TL)) = 1 + length(TL) .
```

17. **Remove the last node of the list.** The operation removes the node with the greatest distance to the searched key from the temporary list. If the list is empty it is not changed.

```
  op remove-last : TemporaryList{X} -> TemporaryList{X} .
  var TL : TemporaryList{X} .  vars NT NT1 NT2 : Node-Time{X} .
  eq remove-last(temp-empty) = temp-empty .
  eq remove-last(insert(NT,temp-empty)) = temp-empty .
  eq remove-last(insert(NT1, NT2,TL)) = insert(NT1,remove-last(NT2,TL)) .
```

18. **Count the number of nodes in the list with time for receiving a message equal to 0.**

```
op remove-last : TemporaryList{X} -> TemporaryList{X} .
var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
var n : Nat .  var F1 : Nat .
eq messages-time0(temp-empty) = 0 .
eq messages-time0(insert(< Tr n 0 F1 >, TL)) = 1 + messages-time0(TL) .
ceq messages-time0(insert(< Tr n TM1 F1 >, TL)) = messages-time0(TL)
  if TM1 =/= 0 .
```

19. **Remove the nodes of the list with time for receiving a message equal to 0.**

```
  op remove-time0 : TemporaryList{X} -> TemporaryList{X} .
  var Tr : X$Contact .  var TL : TemporaryList{X} .  var TM1 : TimeInf .
  var n : Nat .  var F1 : Nat .
  eq remove-time0(temp-empty) = temp-empty .
  eq remove-time0(insert(< Tr n 0 F1 >, TL)) = remove-time0(TL) .
  ceq remove-time0(insert(< Tr n TM1 F1 >, TL)) =
      insert(< Tr n TM1 F1 >, remove-time0(TL))
    if TM1 =/= 0 .
```

20. **Update time.**

```
op delta : TemporaryList{X} Time -> TemporaryList{X} [frozen (1)] .
var Tr : X$Contact .  var TL : TemporaryList{X} .  vars TM1 TM2 : TimeInf .
var n : Nat .  vars F1 F2 : Nat .
eq delta(temp-empty, T) = temp-empty .
ceq delta(insert(< Tr n TM1 F1 >, TL), T) =
       insert(< Tr n (TM1 monus T) F1 >, delta(TL,T))
    if F1 == 1 or F1 == 3 .
ceq delta(insert(< Tr n TM1 F1 >, TL), T) = insert(< Tr n TM1 F1 >, delta(TL,T))
    if F1 == 0 or F1 == 2 or F1 == 4 .

op minTime : TemporaryList{X} -> TimeInf .
eq minTime(temp-empty) = INF .
ceq minTime(insert(< Tr n TM1 F1 >, TL)) = min(TM1, minTime(TL))
   if F1 == 1 or F1 == 3 .
ceq minTime(insert(< Tr n TM1 F1 >, TL)) = INF
   if F1 == 0 or F1 == 2 or F1 == 4 .
```

4 Modeling time

Simulating the behavior of a P2P network requires a notion of time. In the current specification, time passes when some action occurs, in particular since the only actions are the RPCs, we assume that each of them takes a unit time.

We use Maude's `REAL-TIME-MAUDE` module with discrete time units to model time. Rules are divided into *tick rules*, that model the elapse of time on the system, and *instantaneous rules*, that model changes in (part of) the system and are assumed to take zero time.

The *tick rule*, defined in the module `P2P-NETWORK` (`Kademlia.maude` file) has the form:

```
crl [tick] : { C } => { delta(C,mte(C)) } in time mte(C)
  if mte(C) =/= INF and mte(C) =/= 0 .
```

where

- `op mte : Configuration -> TimeInf [frozen (1)] .`

  calculates the number of time units that occur as the minimum of the configuration messages and objects time units, and

- `op delta : Configuration TimeInf -> Configuration [frozen (1)] .`

  defines the effect of time elapse on a configuration. Both operations are declared by the Real Time system.

The effect of the `delta` and the `mte` operations is defined in in the modules that define the operations.

The `PING` message is defined in the `KADEMLIA-PROTOCOL` module (`KademliaRT.maude` file). Only the time to attend the message is changed.

```
ceq delta(to O : PING(SENDER,TM,TM1),TC) =
    to O : PING(SENDER,TM monus TC, TM1 monus TC)
if TM1 > 0 .
eq delta(to O : PING(SENDER,TM,0),TC) =
    to O : PING(SENDER,TM monus TC, 0) .

ceq delta(to O : PING-REPLY(SENDER,TM,TM1), TC) =
    to O : PING-REPLY(SENDER,TM monus TC, TM1 monus TC)
if TM1 > 0 .
eq delta(to O : PING-REPLY(SENDER,TM,0), TC) =
    to O : PING-REPLY(SENDER,TM monus TC, 0) .
```

Time passes in the routing table configuration, since it is waiting for the `PING` messages to be answered.

```
op delta : RTConfiguration Time -> RTConfiguration [frozen (1)] .
eq delta(R + CC + MM + L + T,T1) = R + CC + MM + L + T monus T1 .
```

Time also passes in the different tables (KademliaFile.maude file):

```
op delta : InfoFileTable Time -> InfoFileTable [frozen (1)] .
ceq delta(Tr ;; TI, T) = Tr ;; (TI monus T) if TI gt T .
ceq delta(Tr ;; T2, T) = Tr ;; 1 if T2 le T .

op delta : InfoSearchFile Time -> InfoSearchFile [frozen (1)] .
ceq delta(S ; TI, T) = S ; (TI monus T) if TI > T .
ceq delta(S ; T2, T) = S ; 1 if T2 <= T .

op delta : InfoPublishFile Time -> InfoPublishFile [frozen (1)] .
ceq delta(S @ TI, T) = S @ (TI monus T) if TI gt T .
ceq delta(S @ T2, T) = S @ 1 if T2 le T .

op delta : Table{X,Y} Time -> Table{X,Y} [frozen (1)] .
op delta : Y$Elt TimeInf -> Y$Elt [frozen (1)] .
eq delta(empty-table,T) = empty-table .
eq delta(< X1 & Y1 >, T) = < X1 & delta(Y1, T) > .
ceq delta(FT1 # FT2, T) = delta(FT1, T) # delta(FT2, T) if FT1 =/= empty-table /\
           FT2 =/= empty-table .

op delta : TemporaryList{X} Time -> TemporaryList{X} [frozen (1)] .
eq delta(temp-empty, T) = temp-empty .
ceq delta(insert(< Tr n TM1 F1 >, TL), T) = insert(< Tr n (TM1 monus T) F1 >, delta(TL,T))
  if F1 == 1 or F1 == 3 .
  --- Time does not pass if it is not waiting an answer
ceq delta(insert(< Tr n TM1 F1 >, TL), T) = insert(< Tr n TM1 F1 >, delta(TL,T))
  if F1 == 0 or F1 == 2 or F1 == 4 .
```

The definition of time for the rest of the messages is in the KADEMLIA-TRANSMITTED-SYNTAX module (Kademlia.maude file).

```
eq delta(to O : FIND-NODE(SENDER,P3,TM,TM1), TC) =
     to O : FIND-NODE(SENDER,P3,TM monus TC, TM1 monus TC) .
eq delta(to O : FIND-NODE-REPLY(SENDER,P3,CS,TM,TM1), TC) =
     to O : FIND-NODE-REPLY(SENDER,P3,CS,TM monus TC, TM1 monus TC) .
eq delta(to O : STORE(SENDER,FT,TM,TM1), TC) =
    to O : STORE(SENDER,FT,TM monus TC, TM1 monus TC) .
eq delta(to O : STORE-REPLY(SENDER,I1,TM,TM1), TC) =
    to O : STORE-REPLY(SENDER,I1,TM monus TC, TM1 monus TC) .
eq delta(to O : FIND-VALUE(SENDER,P3,TM,TM1), TC) =
    to O : FIND-VALUE(SENDER,P3,TM monus TC, TM1 monus TC) .
eq delta(to O : FIND-VALUE-REPLY1(SENDER,P3,CS,TM,TM1), TC) =
    to O : FIND-VALUE-REPLY1(SENDER,P3,CS,TM monus TC, TM1 monus TC) .
eq delta(to O : FIND-VALUE-REPLY2(SENDER,I1,P3,TM,TM1), TC) =
    to O : FIND-VALUE-REPLY2(SENDER,I1,P3,TM monus TC, TM1 monus TC) .
eq delta(to O : FILE-FOUND(SENDER,I1), TC) = t
    o O : FILE-FOUND(SENDER,I1) .
```

Finally, time is defined for a configuration in the P2P-NETWORK module (Kademlia.maude file).

```
ceq delta(< peer(MC) : Peer | RT : RTC, Files : FT1, Publish : PF, SearchFiles : SF,
          SearchList : SL, Life : T1 , Reconnect : INF, NumTimesConnected : N >,TC)
= < peer(MC) : Peer | RT : delta(RTC,TC), Files : delta(FT1,TC), Publish : delta(PF,TC),
                 SearchFiles : delta(SF, TC), SearchList : delta(SL, TC), Life : T1 monus TC ,
                 Reconnect : INF, NumTimesConnected : N > if T1 > 0 .

ceq delta(< peer(MC) : Peer | RT : RTC, Files : FT1, Publish : PF, SearchFiles : SF,
      SearchList : SL,Life : INF, Reconnect : T1, NumTimesConnected : N >,TC)
```

```
=  < peer(MC) : Peer | RT : RTC, Files : FT1 , Publish : PF , SearchFiles : SF ,
          SearchList : SL , Life : INF , Reconnect : T1 monus TC, NumTimesConnected : N >
if T1 > 0 .

eq delta(< peer(MC) : Peer | RT : R1 + MCC + MM + L + K  , Files : FT1 , Publish : PF,
     SearchFiles : SF , SearchList : SL, Life : K1 , Reconnect : TM1, NumTimesConnected : N >, N')
= < peer(MC) : Peer | RT : delta(R1 + MCC + MM + L + K, N'), Files : delta(FT1, N'),
                        Publish : delta(PF, N'), SearchFiles : delta(SF, N'),
                        SearchList : delta(SL, N'), Life : K1 monus N',
                        Reconnect : TM1 monus N', NumTimesConnected : N > .
```

The `mte` operation is defined for the different configurations in the `P2P-NETWORK` module (`Kademlia.maude` file).

```
  var MSG : Msg .
  eq mte(MSG) = 0 .
  eq mte(< peer(MC) : Peer | RT : R1 + MCC + MM + L + K, Files : FT1, Publish : PF,
                             SearchFiles : SF, SearchList : SL, Life : T1,
                             Reconnect : INF, NumTimesConnected : N >) =
                                                          minTime(SL) [owise] .

  eq mte(< peer(MC) : Peer | RT : RTC, Files : FT1 , Life : INF, Reconnect : T2 >) = T2 .
  eq mte(< Random : RandomNGen | seed : N3 >) = INF .

  ceq mte(< peer(MC) : Peer | RT : R1 + MCC + MM + L + K, Files : FT1, Publish : PF,
                              SearchFiles : SF, SearchList : SL, Life : T1 ,
                              Reconnect : INF, NumTimesConnected : N >) =
           min(minTime(FT1),min(minTime(PF),min(minTime(SF),min(T1,K))))
   if (FT1 == empty-table or minTime(FT1) =/= INF) /\
      (PF == empty-table or minTime(PF) =/= INF) /\
      (SF == empty-table or minTime(SF) =/= INF) .

  eq mte(< peer(MC) : Peer | RT : R1 + MCC + MM + L + K, Files : FT1, Publish : PF,
                             SearchFiles : SF, SearchList : SL, Life : K1,
                             Reconnect : INF, NumTimesConnected : N >) = INF [owise] .

  eq mte(< peer(MC) : Peer | RT : RTC, Files : FT1, Life : INF, Reconnect : K2 >) = K2 .
```

4.1 Time constants

We define some constants for the system in the `TIME-CONSTANTS` module (`TimeFiles` file) to control the pass of time.

```
  --- Time an RPC will be alive
  op RPCRemove : -> Nat .
  --- Time a peer will be in the search list if it does not answer
  op SearchListRemove : -> Nat .
  --- Time to republish a file. The peer has its ID
  op RePublishID : -> Nat .
  --- Time to republish a file. The peer is publishing the file
  op RePublishFile : -> Nat .
  --- Maximum life a peer is created with
  op MaxLife : -> Nat .
  --- Time to research a file when it is not founded
  op ReSearch : -> Nat .
```

5 Network processes

We present processes for searching a file, publishing a file, and republishing a file that has been published by other node. We also present the processes for treating the `PING` RPC that are sent

by the routing table to insert new contacts when the table is full.

5.1 Looking for file

The searching process starts automatically when there are IDs in the `SearchFiles` attribute of some peer, with expiration time equal to 1. We call the peer that starts a searching process the initiator. In this version we only permit one search per node at a time as there is only one `SearchList` in the attributes of the peer. The search list should be empty to start a new looking-up process. The life time of the initiator, should be greater than zero; otherwise, the node is supposed to be disconnected. The expiration time of the file should be 1 since the zero value indicates that the search has finished, successfully or not, and a greater value indicates that the file has already been searched for, but it was not found and now is waiting to repeat the search. Time does not pass for a file with expiration time equal 1, simulating that it will wait forever for the look-up process.

1. The first step consists of creating a temporary search list with the contacts of the peer routing table that are the closest to the ID of the searched file.

```
  var SENDER : MyContact .   var ID : BitString160 . var S : String .
  var TM : TimeInf .   var NzT : NzTime .
  var R : RoutingTable{vContact-BitString} .  var MCC : MyContact+ .
  var MM : MayBeMessage .  var L : ContactList{vContact-BitString} .
  var SF : TSearchFile .

rl [lookfor-file1] :
   < peer(SENDER) : Peer |  RT : R + MCC + MM + L + TM ,
        SearchFiles : < ID & (S ; 1) > # SF , SearchList : temp-empty , Life : NzT >
=>
   < peer(SENDER) : Peer | RT : R + MCC + MM + L + TM ,
        SearchFiles : < I1 & (S ; INF) > # SF ,
        SearchList : create-search-list(closest-nodes(ID,R,FilesInSearchTable), ID) ,
        Life : NzT > .
```

   When the process starts, the expiration time of the searched file is set to `INF` to indicate that the process is initiated. The temporary search list is filled with the `FilesInSearchTable` *closest* nodes the initiator has in its routing table. `FilesInSearchTable` is a global constant set in the `ROUTING-CONSTANTS` module that is defined in the `Kademlia.maude` file. The `closest-nodes` operation, defined in the `ROUTING-TABLE` module (`kademliaRT.maude` file), returns the `FilesInSearchTable` closest nodes to the key `ID` in the routing table `R`. `FilesInSearchTable` is a global constant defined in the `ROUTING-CONSTANTS` module (`kademlia.maude` file) that defines the maximum number of files in the `SearchFiles` table.The list is created with the operation `create-search-list`, defined in the `TEMPORARY-LIST` module (`KademliaFiles.maude` file), which inserts the nodes returned by the `closest-nodes` operation ordered by its distance to the key.

2. The process continues by sending `FIND-VALUE` RPCs to the `ParallelSearchRPC` first nodes of the list. `ParallelSearchRPC` defines the maximum number of RPCs that are sent in parallel. It is a global constant defined in the `ROUTING-CONSTANTS` module (`kademlia.maude` file).

```
  var ID : BitString160 . var SENDER : MyContact .  var S : String .
  var SF : TSearchFile . var SL : TemporaryList{vContact-BitString} . var NzT : NzTime .
  var Tr : MyContact .

  ceq [lookfor-file21] :
     < peer(SENDER) : Peer | SearchFiles : < ID & (S ; INF) > # SF , SearchList : SL,
                             Life : NzT >
  =  < peer(SENDER) : Peer | SearchFiles : < ID & (S ; INF) > # SF,
                             SearchList : set-flag-process(Tr,SearchListRemove,SL),
```

```
                              Life : NzT >
     to peer(Tr) : FIND-VALUE(SENDER, ID, RPCRemove, 1)
  if not all-sent(SL) /\ Tr := first-not-send(SL) /\
     messages-in-process(SL) < ParallelSearchRPC /\
     number-nodes-reply(SL) < kSearched .
```

Once the RPC is sent, a flag is activated in the search list that marks this node as *in process*. The RPC is sent if the initiator is active (`NzT > 0`); if there are still nodes in the search list to which no RPC has been sent; if the number of messages is less than the `ParallelSearchRPC` constant; and if there are less than `kSearched` RPCs that have not replied yet, where `KSearched` is a constant defined in the `ROUTING-CONSTANTS` module (`kademlia.maude` file). Notice that we have to ask as many nodes as possible, because there can be nodes not so close to the objective than others but that have in their routing tables information of the closest ones.

When the flag of the contact of the temporary search list is changed the time to remove the contact from the list is also updated to the constant `SearchListRemove` defined in the `TIME-CONSTANTS` module (`TimeFiles.maude` file). The RPC is sent with a time to remove set to the constant `RPCRemoved` (module `TIME-CONSTANTS`).

3. The receiver may find the value, rule `find-value2`, or it may return the closest nodes to the ID it knows about, rule `find-value1`. Each `FIND-VALUE` RPC has a life time, which is set to the global constant, `RPCRemove` defined in the `TIME-CONSTANTS` module (`TimeFiles.maude` file), when it is created. If the message life time is set to zero, the message is supposed not to be attended, and it is removed from the system, rule `find-value3`.

```
var P3 : BitString160 . vars SENDER RECEIVER : MyContact .
vars TM TM1 : TimeInf .   var NzT : NzTime .
var R : RoutingTable{vContact-BitString} .  var MCC : MyContact+ .
var MM : MayBeMessage .  var L : ContactList{vContact-BitString} .
var FT1 : TFileTable .

crl [find-value1] :
   (to peer(RECEIVER) : FIND-VALUE(SENDER, P3, TM1, 0))
   < peer(RECEIVER) : Peer | RT : R + MCC + MM + L + TM, Files : FT1, Life : NzT >
=> < peer(RECEIVER) : Peer | RT : R + MCC + MM + L SENDER + TM, Files : FT1, Life : NzT >
   (to peer(SENDER) : FIND-VALUE-REPLY1(RECEIVER, P3, closest-nodes(P3,R,bucketDim),
          RPCRemove, 1))
if not (P3 in FT1) /\ TM1 > 0 .

crl [find-value2] :
   (to peer(RECEIVER) : FIND-VALUE(SENDER, P3, TM1,0))
   < peer(RECEIVER) : Peer | RT : R + MCC + MM + L + TM , Files : FT1 , Life : NzT >
=> < peer(RECEIVER) : Peer | RT : R + MCC + MM + L SENDER + TM, Files : FT1, Life : NzT >
   (to peer(SENDER) : FIND-VALUE-REPLY2(RECEIVER, P3, first?(find(P3,FT1)), RPCRemove, 1))
if (P3 in FT1) /\ TM1 > 0 .

eq [find-value3] : to 0 : FIND-VALUE(SENDER, P3, 0, TM1) = none .
```

Notice that each time a peer receives an RPC it changes its routing table by moving the `SENDER` of the RPC to the tail of the bucket by adding it to the contact list of the routing configuration. If the peer does not have the file ID in its `Files` table, that is the peer is not responsible of publishing the file, it will return as many peers ID as the bucket dimension constant indicates by means of the `FIND-VALUE-REPLY1` RPC. If the peer knows the file ID location, it returns it by means of the `FIND-VALUE-REPLY2` RPC.

4. If the initiator receives a `FIND-VALUE-REPLY2` RPC with the peer that publish the file, the process ends.

```
vars P3 ID : BitString160 . vars SENDER RECEIVER : MyContact .
vars TM TM1 : TimeInf .   var NzT : NzTime .
var R : RoutingTable{vContact-BitString} .  var MCC : MyContact+ .
var MM : MayBeMessage .  var L : ContactList{vContact-BitString} .
var SF : TSearchFile . var SL : TemporaryList{vContact-BitString} .
var S : String .

crl [lookfor-file3] :
   (to peer(RECEIVER) : FIND-VALUE-REPLY2(SENDER,ID,P3,TM1,0))
   < peer(RECEIVER) : Peer | RT : R + MCC + MM + L + TM ,
             SearchFiles : < ID & (S ; INF) > # SF ,SearchList : SL , Life : NzT >
=>
   < peer(RECEIVER) : Peer | RT : R + MCC + MM + L SENDER + TM , SearchFiles : SF ,
              SearchList : temp-empty , Life : NzT >
   to peer(RECEIVER) : FILE-FOUND(SENDER,ID)
if TM1 > 0 .
```

Here again the routing table is updated with the contact that sends the message, which is added to the contact list of the routing configuration.

5. If it receives the list of the closest nodes, it changes its search list, adding the nodes ordered by the distance to the objective. Only nodes closer than the one which proposes them are added. The initiator also updates its routing table, as it is always done when an RPC is received. When the full list is treated, a flag is activated to mark this node as done in the search list.

```
var P3 : BitString160 . vars SENDER RECEIVER : MyContact .
vars TM TM1 : TimeInf .   var NzT : NzTime .
var R : RoutingTable{vContact-BitString} .  var MCC : MyContact+ .
var MM : MayBeMessage .  var L : ContactList{vContact-BitString} .
var SL : TemporaryList{vContact-BitString} .
var CS : Set{vCONTACT}{vContact-BitString} .

crl [lookfor-file40] :
   (to peer(RECEIVER) : FIND-VALUE-REPLY1(SENDER,P3,CS,TM1,0))
   < peer(RECEIVER) : Peer | RT : R + MCC + MM + L + TM ,
                    SearchList : SL , Life : NzT >
=>
   < peer(RECEIVER) : Peer | RT : R + MCC + MM + L SENDER + TM ,
                     SearchList : insertList(CS,SL,SENDER,RECEIVER,P3) , Life : NzT >
if TM1 > 0 .
```

6. If the `FIND-VALUE` RPC is not attended because the receiver has left the network, the node remains in the search list blocking other searches. When this happens the node should be removed from the search list. To detect these cases, each node in the search list has a time to reply. When this time is set to 0 the node is removed from the list.

```
var I1 : BitString160 . var SENDER : MyContact .
var NzT : NzTime .     var S1 : String .
var SF : TSearchFile . var SL : TemporaryList{vContact-BitString} .

ceq [lookfor-file5] :
   < peer(SENDER) : Peer | SearchFiles : < I1 & (S1 ; INF) > # SF ,
                SearchList : SL , Life : NzT >
=
    < peer(SENDER) : Peer | SearchFiles : < I1 & (S1 ; INF) > # SF ,
                SearchList : remove-time0(SL) , Life : NzT >
if messages-time0(SL) > 0 .
```

7. If all the peers in the temporary search list have replied or failed to reply and the file has not been found, we keep the file in the `SearchFile` table and give it a time, given by the `ReSearch` constant (`TIME-CONSTANTS` module, `TimeFiles.maude` file) to repeat the looking-up process.

```
var ID : BitString160 . var SENDER : MyContact .  var NzT : NzTime .
var SF : TSearchFile . var SL : TemporaryList{vContact-BitString} .
var S : String .

ceq [lookfor-file6] :
    < peer(SENDER) : Peer | SearchFiles : < ID & (S ; INF) > # SF ,
            SearchList : SL , Life : NzT >
=   < peer(SENDER) : Peer | SearchFiles : < ID & (S ; ReSearch) > # SF ,
            SearchList : temp-empty , Life : NzT >
if first-k-done(SL,kSearched) .
```

5.2 Publishing a file

Publication is performed automatically. Even more, to ensure the persistence of the information, nodes periodically republish files. In [8] not only the node that shares the file republishes it, but also all the nodes which store the file ID. The process is done each hour but, to avoid replication, when a node receives a `STORE` RPC it will not republish the file in the next hour. As said in [8], since replication intervals are not exactly synchronized, only one node will republish the file every hour, making the process more efficient.

A file is published on the k nodes which have the *closest* ID to the file ID since the other nodes will look for the file there. The publish process starts automatically when the time to republish a file is set to one. It can be a node's shared file kept in the *publish files* table or a known file shared by other node. The first task is creating the temporary search list.

```
var R : RoutingTable{vContact-BitString} .     var MCC : MyContact+ .
var MM : MayBeMessage .      var L : ContactList{vContact-BitString} .
var TM : TimeInf .     var I1 : BitString160 .   var S : String .
var PF : TPublishFile .   var NzT : NzTime . var SENDER : MyContact .

rl [publish11] :
   < peer(SENDER) : Peer | RT : R + MCC + MM + L + TM , Publish : < I1 & (S @ 1) > # PF ,
            SearchList : temp-empty , Life : NzT >
=> < peer(SENDER) : Peer | RT : R + MCC + MM + L + TM , Publish : < I1 & (S @ INF) > # PF ,
            SearchList : create-search-list(closest-nodes(I1,R,FilesInSearchTable), I1) ,
            Life : NzT > .
```

In the following we only explain the process that treats the shared file process; the one for the known files is similar. First the initiator should find the k *closest* nodes to the file ID. The initiator sends `FIND-NODE` RPCs to the closest nodes of the temporary search list.

```
var I1 : BitString160 .   var S : String .
var PF : TPublishFile .   var NzT : NzTime . var SENDER : MyContact .
var SL : TemporaryList{vContact-BitString} .     var Tr : MyContact .

ceq [publish21] :
   < peer(SENDER) : Peer | Publish : < I1 & (S @ INF) > # PF , SearchList : SL , Life : NzT >
=  < peer(SENDER) : Peer | Publish : < I1 & (S @ INF) > # PF ,
                           SearchList : set-flag-process(Tr, SearchListRemove, SL) ,
                           Life : NzT >
   to peer(Tr) : FIND-NODE(SENDER, I1, RPCRemove, 1)
if  not all-sent(SL) /\  Tr := first-not-send(SL) /\
    messages-in-process(SL) < ParallelSearchRPC /\
    number-nodes-reply(SL) < kSearched .
```

On receiving the `FIND-NODE-REPLY` RPC response, the initiator incorporates the set of contacts to its temporary list.

```
var I1 : BitString160 .   var S : String .
var PF : TPublishFile .   var NzT : NzTime . vars SENDER RECEIVER : MyContact .
var SL : TemporaryList{vContact-BitString} .   var TM1 : TimeInf .
var CS : Set{vCONTACT}{vContact-BitString} .

ceq [publish30] :
   (to peer(RECEIVER) : FIND-NODE-REPLY(SENDER,I1,CS,TM1,0))
   < peer(RECEIVER) : Peer | Publish : < I1 & (S @ INF) > # PF , SearchList : SL ,
          Life : NzT >
=  < peer(RECEIVER) : Peer | Publish : < I1 & (S @ INF) > # PF ,
          SearchList : insertList(CS, SL, SENDER , RECEIVER , I1) , Life : NzT >
if TM1 > 0 .
```

Delete elements of the search list that do not respond the find-node message in time.

```
var I1 : BitString160 .   var S : String .
var PF : TPublishFile .   var NzT : NzTime . var SENDER : MyContact .
var SL : TemporaryList{vContact-BitString} .

ceq [publish4] :
   < peer(SENDER) : Peer | Publish : < I1 & (S @ INF) > # PF , SearchList : SL ,
            Life : NzT >
=  < peer(SENDER) : Peer | Publish : < I1 & (S @ INF) > # PF ,
            SearchList : remove-time0(SL) , Life : NzT >
if messages-time0(SL) > 0 .
```

Then, a `STORE` message is sent to the first `RedundantPublish` nodes of the list, that are supposed to be the closest to the file's ID. When the `RedundantPublish` `STORE` messages have been sent the time to republish the file is set to `RePublishID`. Both constants are defined in the `ROUTING-CONSTANTS` module (`KademliaRT.maude` file).

```
var I1 : BitString160 .   var S1 : String .
var PF : TPublishFile .   var NzT : NzTime . vars SENDER RECEIVER : MyContact .
var SL : TemporaryList{vContact-BitString} .   var TM1 : TimeInf .
var CS : Set{vCONTACT}{vContact-BitString} .

ceq [publish51] :
  < peer(SENDER) : Peer | Publish : < I1 & (S1 @ INF) > # PF , SearchList : SL , Life : NzT >
=
  < peer(SENDER) : Peer | Publish : < I1 & (S1 @ INF) > # PF ,
                   SearchList : set-flag-store(Tr, SearchListRemove, SL) , Life : NzT >
  (to peer(Tr) : STORE(SENDER,< I1 & (get-ID(SENDER) ;; RePublishID) >, RPCRemove, 1))
if first-k-done(SL, RedundantPublish) /\
     number-messages-store(SL) + number-messages-store-reply(SL) < RedundantPublish /\
     prepared-stored(SL) /\ Tr := first-not-stored(SL) .

ceq [publish52] :
   (to peer(SENDER) : STORE-REPLY(RECEIVER,I1,TM1, 0))
   < peer(SENDER) : Peer | Publish : < I1 & (S1 @ INF) > # PF , SearchList : SL , Life : NzT >
=
   < peer(SENDER) : Peer | Publish : < I1 & (S1 @ INF) > # PF ,
                   SearchList : set-flag-store-reply(RECEIVER, SearchListRemove, SL),
                   Life : NzT >
if TM1 > 0  /\ number-messages-store-reply(SL) < RedundantPublish .

ceq [publish53] :
   < peer(SENDER) : Peer | Publish : < I1 & (S1 @ INF) > # PF , SearchList : SL , Life : NzT >
=  < peer(SENDER) : Peer | Publish : < I1 & (S1 @ RePublishID) > # PF ,
                  SearchList : temp-empty , Life : NzT >
if (number-messages-store-reply(SL) == RedundantPublish or all-store(SL)) .
```

5.3 Republishing a file

This process publishes a file that is shared by another peer, but kept in this one. The process is similar to the previous one, but changing the `Publish` attribute by the `Files` one.

```
var R : RoutingTable{vContact-BitString} .     var MCC : MyContact+ .
var MM : MayBeMessage .      var L : ContactList{vContact-BitString} .
var TM : TimeInf .     vars I1 I2 : BitString160 .   var S : String .
var FT : TFileTable .    var NzT : NzTime . var SENDER : MyContact .
var Tr : MyContact .     var SL : TemporaryList{vContact-BitString} .

rl [republish11] :
   < peer(SENDER) : Peer | RT : R + MCC + MM + L + TM , Files : < I1 & (I2 ;; 1) > # FT ,
             SearchList : temp-empty , Life : NzT >
=> < peer(SENDER) : Peer | RT : R + MCC + MM + L + TM , Files : < I1 & (I2 ;; INF) > # FT ,
             SearchList : create-search-list(closest-nodes(I1,R,FilesInSearchTable), I1) ,
             Life : NzT > .

ceq [republish21] :
   < peer(SENDER) : Peer | Files : < I1 & (I2 ;; INF) > # FT , SearchList : SL ,
              Life : NzT >
= < peer(SENDER) : Peer | Files : < I1 & (I2 ;; INF) > # FT ,
                SearchList : set-flag-process(Tr, SearchListRemove, SL) , Life : NzT >
   (to peer(Tr) : FIND-NODE(SENDER, I1, RPCRemove, 1))
if  not all-sent(SL) /\ Tr := first-not-send(SL) /\
      messages-in-process(SL) < ParallelSearchRPC /\
      number-nodes-reply(SL) < kSearched .

ceq [republish30] :
   (to peer(RECEIVER) : FIND-NODE-REPLY(SENDER,I1,CS,TM1,0))
   < peer(RECEIVER) : Peer | Files : < I1 & (I2 ;; INF) > # FT  , SearchList : SL ,
             Life : NzT >
=  < peer(RECEIVER) : Peer | Files : < I1 & (I2 ;; INF) > # FT ,
                SearchList : insertList(CS, SL, SENDER , RECEIVER , I1) , Life : NzT >
if TM1 > 0 .

ceq [republish35] :
   < peer(RECEIVER) : Peer | Files : < I1 & (I2 ;; K) > # FT , Life : NzT >
   (to peer(RECEIVER) : FIND-NODE-REPLY(SENDER,I1,CS,TM1,0))
=
   < peer(RECEIVER) : Peer | Files : < I1 & (I2 ;; K) > # FT , Life : NzT >
if K =/= INF .

ceq [republish4] :
   < peer(SENDER) : Peer | Files : < I1 & (I2 ;; INF) > # FT , SearchList : SL , Life : NzT >
=
   < peer(SENDER) : Peer | Files : < I1 & (I2 ;; INF) > # FT ,
                SearchList : remove-time0(SL) , Life : NzT >
if messages-time0(SL) > 0 .

ceq [republish51] :
  < peer(SENDER) : Peer | Files : < I1 & (I2 ;; INF) > # FT , SearchList : SL , Life : NzT >
=
  < peer(SENDER) : Peer | Files : < I1 & (I2 ;; INF) > # FT ,
               SearchList : set-flag-store(Tr, SearchListRemove, SL) , Life : NzT >
  (to peer(Tr) : STORE(SENDER,< I1 & (I2 ;; RePublishID) >, RPCRemove, 1))
if first-k-done(SL, RedundantPublish) /\
    number-messages-store(SL) < RedundantPublish /\
     prepared-stored(SL) /\ Tr := first-not-stored(SL) .

ceq [republish52] :
   (to peer(SENDER) : STORE-REPLY(RECEIVER,I1,TM1, 0))
```

```
   < peer(SENDER) : Peer | Files : < I1 & (I2 ;; INF) > # FT , SearchList : SL , Life : NzT >
=  < peer(SENDER) : Peer | Files : < I1 & (I2 ;; INF) > # FT,
                  SearchList : set-flag-store-reply(RECEIVER, SearchListRemove, SL),
                  Life : NzT >
if TM1 > 0  /\ number-messages-store-reply(SL) < RedundantPublish .

ceq [republish53] :
   < peer(SENDER) : Peer | Files : < I1 & (I2 ;; INF) > # FT , SearchList : SL , Life : NzT >
=
   < peer(SENDER) : Peer | Files : < I1 & (I2 ;; RePublishID) > # FT ,
               SearchList : temp-empty , Life : NzT >
if (number-messages-store-reply(SL) == RedundantPublish or all-store(SL)) .
```

5.4 Treatment of the routing table PING RPC

The following equations and rules define the behavior of the routing table configuration when it sends PING messages to check if a peer is still alive and receives the PING-REPLY messages from the alive peers.

- **The routing table configuration sends a PING message**. The system sends the message and sets the time the routing table configuration will wait for the reply.

```
vars SENDER RECEIVER : MyContact .   var R : RoutingTable{vContact-BitString} .
var Tr : MyContact .   vars NzT NzT1 NzT2 : NzTime .
var L : ContactList{vContact-BitString} .

eq [RT-send] :
   < peer(SENDER) : Peer | RT : R + Tr + (to peer(RECEIVER) : PING(SENDER,NzT1,NzT2))
                                    + L + INF , Life : NzT >
=  < peer(SENDER) : Peer | RT : R + Tr + (to peer(RECEIVER) : PING(SENDER,NzT1,NzT2))
                                    + L + NzT1 , Life : NzT  >
   to peer(RECEIVER) : PING(SENDER,NzT1,NzT2) .
```

- **The peer is alive and has sent a PING-REPLY message**. The peer is not removed from the routing table by means of the add-entry2 operation defined in the ROUTING-TABLE module (KademliaRT.maude file). The add-entry2 operation flag is set to true to indicate that the reply has been received, the peer should not be removed and the new contact that generates this PING message because the bucket was full should not be added to the routing table.

```
vars SENDER RECEIVER : MyContact .   var R : RoutingTable{vContact-BitString} .
var Tr : MyContact .   vars NzT NzT1 : NzTime .   vars TM3 TM4 : TimeInf .
var T : Time .    var L : ContactList{vContact-BitString} .

eq [ping-reply1] :
   (to peer(RECEIVER) : PING-REPLY(SENDER,NzT1,0))
   < peer(RECEIVER) : Peer | RT : R + Tr + (to peer(SENDER) : PING(RECEIVER,TM3,TM4))
                                    + L + T , Life : NzT >
=  < peer(RECEIVER) : Peer | RT : add-entry2(Tr,R,RECEIVER,L,true) , Life : NzT > .
```

- **The peer is not alive and has not sent a PING-REPLY message**. The peer should be removed from the routing table. The add-entry2 operation defined in the ROUTING-TABLE module (KademliaRT.maude file) is used with the flag set to false to indicate that the contact should be removed from the routing table and the new contact that generates this PING message because the bucket was full should be added to the routing table.

```
vars SENDER RECEIVER : MyContact .   var R : RoutingTable{vContact-BitString} .
var Tr : MyContact .   vars NzT NzT1 NzT2 : NzTime .
var L : ContactList{vContact-BitString} .
```

```
eq [ping-reply3] :
< peer(SENDER) : Peer | RT : R + Tr + (to peer(RECEIVER) : PING(SENDER,NzT1,NzT2))
                              + L + 0, Life : NzT >
= < peer(SENDER) : Peer | RT : add-entry2(Tr,R,SENDER,L,false), Life : NzT > .
```

- **A PING-REPLY message is received, without having sending a PING message**. The contact is added to the routing table configuration contact list to be added later on to the routing table, since is a new acknowledge contact.

```
vars SENDER RECEIVER : MyContact .   var R : RoutingTable{vContact-BitString} .
var Tr : MyContact .     var MCC : MyContact+ .    var MM : MayBeMessage .
vars NzT NzT1 : NzTime .     vars TM : TimeInf .
var L : ContactList{vContact-BitString} .

crl [RT-receive1] :
       (to peer(RECEIVER) : PING-REPLY(SENDER,NzT1,0))
       < peer(RECEIVER) : Peer | RT : R + MCC + MM + L + TM , Life : NzT >
=> < peer(RECEIVER) : Peer | RT : R + MCC + MM + L SENDER + TM , Life : NzT >
  if TM > 0 /\ not is-ping(RECEIVER,SENDER,MM) .
```

- **A peer receives a PING message and sends a reply.** The peer adds the new contact to its routing table configuration contact list, in order to check later if it is already in its routing table .

```
vars SENDER RECEIVER : MyContact .   var R : RoutingTable{vContact-BitString} .
var Tr : MyContact .     var MCC : MyContact+ .    var MM : MayBeMessage .
vars NzT NzT1 : NzTime .     vars TM : TimeInf .
var L : ContactList{vContact-BitString} .

rl [ping1] :
    (to peer(RECEIVER) : PING(SENDER,NzT1,0))
    < peer(RECEIVER) : Peer | RT : R + MCC + MM + L + TM , Life : NzT >
=>
    < peer(RECEIVER) : Peer | RT : R + MCC + MM + L SENDER + TM , Life : NzT >
    to peer(SENDER) : PING-REPLY(RECEIVER, RPCRemove, 1) .
```

- **Add a contact of the routing table contact list to a routing table.** The operation is done when the routing table is not waiting any message reply. It uses the add-entry operation defined in the ROUTING-TABLE module (KademliaRT.maude file), which checks if there is space in the routing table for the new contact.

```
vars SENDER RECEIVER : MyContact .   var R : RoutingTable{vContact-BitString} .
var NzT : NzTime .   var L : ContactList{vContact-BitString} .

eq [ping2] :
  < peer(RECEIVER) : Peer | RT : R + noneContact + noneMessage + SENDER L + INF ,
                                  Life : NzT >
= < peer(RECEIVER) : Peer | RT : add-entry(SENDER,R,RECEIVER,L) , Life : NzT > .
```

- **Remove messages from a routing table configuration**. The messages are removed when their waiting time is zero.

```
  eq [ping3] : to 0 : PING(SENDER,0,NzT) = noneMessage  .
  eq [ping-reply4] : to 0 : PING-REPLY(SENDER,0,NzT) = noneMessage .
```

We use an auxiliary operation that detects if their is a PING message in the routing table configuration.

```
vars SENDER RECEIVER : MyContact .    vars NzT1 NzT2 : NzTime .
var MM : MayBeMessage .

op is-ping : MyContact MyContact MayBeMessage -> Bool .
eq is-ping(SENDER,RECEIVER,(to peer(RECEIVER) : PING(SENDER,NzT1,NzT2))) = true .
eq is-ping(SENDER, RECEIVER,MM) = false [owise] .
```

6 Open issues

We have shown a model of a P2P network that uses a Kademlia DHT for searching files in the formal language Maude. The model will permit us to execute the network specification, analyze its behaviour and prove properties about it.

But there are still some open issues in the model. There are more network processes, like the one that automatically connects a node to the network, that need to be refined. There are also some eMule facilities that we have not studied yet, like the modification of the routing table to keep more contacts in it or the `type` and `expire time` attributes used to keep the routing table up-to-date. It also allows publishing keywords and notes related to files. There are some protections eMule implements to protect itself against possible attacks, like the protection of hot nodes, that need a deep study. It will also be useful to compare the eMule implementation with the aMule and BitTorrent ones.

We should refine the notion of time adjusting the time it takes each action and the intervals in which the automatic actions are taken in order to make the system as realistic as possible.

The simulation will require: a process to create random peers that could be connected and disconnected from the network; stochastic processes to simulate the behaviour of the peers; and a system that automatically searches for files.

Finally, we have to define the properties we want to prove in the system and use the appropriate tools to prove them. The basic property a P2P file sharing network must meet is that: under all circumstances, the data stored in a hash table must be properly returned when asked for. Different circumstances may affect the seaching process: peers joining and leaving the network; publishing new files; searching for other files; .... Real Time Maude provides some techniques for proving this type of dynamic properties [12]. It admits a reachibility analysis from an initial state with a pattern behaviour up to a certain time bound. It also provides a *temporal logic model checking* that may be very useful if we can find an appropiate abstraction of the model that limits the number of states [13].

References

[1] aMule homepage `http://www.amule.org`

[2] R. Bakhshi and D. Gurov. Verification of peer-to-peer algorithms: A case study. In *Combined Proceedings of the 2nd 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*, *ENTCS* 181, pages 35–47. Elsevier, 2007.

[3] J. Borgström, U. Nestmann, L. O. Alima, and D. Gurov. Verifying a structured peer-to-peer overlay network: The static case. In C. Priami and P. Quaglia, editors, *Proceedings of the International Workshop on Global Computing 2004, GC 2004*, *LNCS* 3267, pages 251–266. Springer, 2004.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, *LNCS* 4350. Springer, 2007.

[5] Crosby S. and Wallach D. An Analysis of BitTorrent's Two Kademlia-Based DHTs Technical Report `TR-07-04`, Department of Computer Science, Rice University, Houston, TX, USA., 2007.

[6] H. Breitkreuz. The eMule project. `http://www.emule-project.net`.

[7] S. Haridi. EU-project PEPITO `IST-2001-33234`, 2002. Project funded by EU IST FET Global Computing (GC). `http://www.sics.se/pepito/`.

[8] 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 1st International Workshop on Peer-to-Peer Systems, IPTPS 2001*, *LNCS* 2429, pages 53–65. Springer, 2002.

[9] Mühl G. Large-Scale Content-Based Publish/Subscribe Systems. Master Thesis. Darmstädter Dissertationen D17. Technischen Universität Darmstadt. 2002.

[10] Mysicka, D. 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.

[11] D. Mysicka. eMule attacks and measurements. Master's thesis, Swiss Federal Institute of Technology (ETH) Zurich, 2007.

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

[13] Miguel Palomino Tarjuelo, Refexión, abstracción y simulación en la lógica de reescritura. PhD thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain, Mar. 2005.

[14] I. Pita and M. Fernández-Camacho. Formal specification of the Kademlia and the Kad routing tables in Maude. In N. Martí-Oliet and M. Palomino, editors, *Proceedings of 21st International Workshop on Recent Trends in Algebraic Development Techniques, WADT 2012*, *LNCS* 7841, pages 231–247. Springer, 2013.

[15] I. Pita and A. Riesco. Specifying and Analyzing the Kademlia Protocol in Maude, In M. Leucker and C. Rueda and F. D. Valenciaeditors, *Proceedings of Theoretical Aspects of Computing - ICTAC 2015 -12th International Colloquium Cali, Colombia, October 29-31, 2015*, *LNCS* 9399, pages 524–541. Springer, 2015.

[16] 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.

[17] 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*, *LNCS* 2218, pages 329–350. Springer, 2001.

[18] Saroiu S, Gummadi P., and Gribble S. A Measurement Study of Peer-to-Peer File Sharing Systems. Technical Report `UW-CSE-01-06-02`, Department of Computer Science and Engineering, University of Washington, july 2001.

[19] Sit E. and Morris R. Security Considerations for Peer-to-Peer Distributed Hash Tables. *In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts, March 2002.*, LNCS 2429, pages 261-269. Springer, 2002. *In Proceedings of Middleware*, Heidelberg. 2001.

[20] 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.

[21] Verdejo A., Pita I. and Mart-Oliet N. Specification and Verification of the Tree Identify Protocol of IEEE 1394 in Rewriting Logic. *Formal Aspects of Computing.* Volume 14, number 3, pages 228-246. Springer, 2003. *In Proceedings of SIGCOMM*, 2001.

[22] Wang P., Tyra J., Chan-Tin E., Malchow T., Foo Kune D., Hopper N., and Kim Y. Attacking the Kad Network. *In Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm'08).* 2008.