A formal specification of the Kademlia distributed hash table

Isabel Pita
Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid
ipandreu@sip.ucm.es

Resumen

Kademlia is a peer-to-peer distributed hash table (DHT) currently used in the P2P eDonkey file sharing network. The most popular clients used to connect to the Kad network are eMule, aMule and BitTorrent. As other DHT, Kademlia look-up algorithm takes $\log n$ steps, which can be reduced to $\log_{2^b} n$ by increasing the node's routing table size. It also offers a number of desirable features not offered by any previous DHT, which makes it the only DHT used in real networks. These features result from the use of a notion of *distance* between objects introduced by Kademlia. Both nodes and shared files are represented by n-bit keys, and their relation depend on the *distance* between their keys. In this sense, 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 (or almost *closest* node, if the information is replicated) to the file key.

This paper explains the specification of the behaviour of a P2P network that uses the Kademlia DHT in the formal specification language Maude. We use the initial description of the Kademlia DHT and fill some open issues with the eMule real implementation. We allow peers to connect to the network and leave it by simulating time using the *Real Time Maude* facilities.

1. Introduction

Peer-to-peer (P2P) systems have seen a great growth in the last few 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 over the last years, such as Chord [18], CAN [14] or Pastry [15]. 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 one.

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 [17, 20, 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. Borgströn 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, in [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 (v0.50a). 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 [19] and it offers simple an elegant time simulation resources.

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 $i = 1 \dots n$ 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. [16] 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

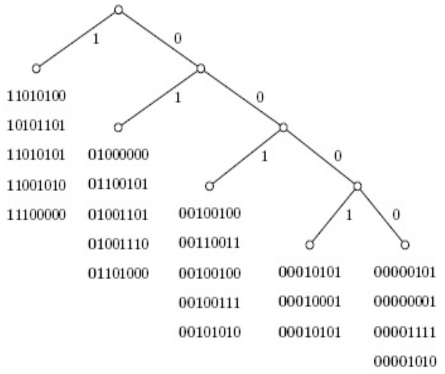

Figura 1: A routing table example for node 00000000

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. Figure 1 shows a routing table for node 00000000 and a *k*-bucket of length 5. IDs have 8 bits.

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.

```
class Peer | RT : RoutingTable ,
             Files : TFileTable ,
             Publish : TPublishFile ,
             SearchFiles : TSearchFile ,
             SearchList : TemporaryList ,
             Life : TimeInf ,
             Reconnect : TimeInf .
```

where the object identification consists of: the peer IP address; its UDP port; and its node ID. It is defined of sort `Triple`.

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 shared peer files. 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 look for many files.
- 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.

The messages represent the RPCs. There is a message for each RPC defined in the Kademlia protocol. The first two parameters of the message are the peer that sends the message and the peer that receives it. 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 PING RPC syntax is:

```
msg PING :
  Triple Triple TimeInf TimeInf -> Msg .
msg PING-REPLY :
  Triple Triple TimeInf TimeInf -> Msg .
```

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

```
msg STORE :
  Triple Triple TFileTable TimeInf TimeInf
  -> Msg .
msg STORE-REPLY :
  Triple Triple TimeInf TimeInf -> Msg .
```

The FIND-NODE message has an aditional parameter that represents the key the sender is looking for. The reply has an 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.

```
msg FIND-NODE :
  Triple Triple BitString TimeInf TimeInf
  -> Msg .
msg FIND-NODE-REPLY :
  Triple Triple List{Triple} TimeInf
  TimeInf -> Msg .
```

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.

```
msg FIND-VALUE :
  Triple Triple BitString TimeInf TimeInf
  -> Msg .
msg FIND-VALUE-REPLY1 :
  Triple Triple List{Triple} TimeInf
  TimeInf -> Msg .
msg FIND-VALUE-REPLY2 :
  Triple Triple BitString TimeInf TimeInf
  -> Msg .
```

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 haven't considered it yet 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.

The empty routing table is represented by an empty bucket that covers all the ID space.

```
subsort Bucket < RoutingTable .
op _||_ :
  Bucket RoutingTable -> RoutingTable [ctor] .
```

The information about nodes stored in the routing table includes the IP address of the node, the UDP port and the node ID. In the following we call them *contacts.*

k-buckets are a list of contacts, which can be empty. The order in which identifiers are allocated in the list is important, since the most recent identifiers are removed first, in this sense its behaviour is similar to a queue. k-buckets are defined as follows:

```
subsort Triple < Bucket .
op empty-bucket : -> Bucket [ctor] .
op _|_ : Bucket Bucket -> Bucket
         [assoc id: empty-bucket ctor] .
```

The k-bucket number of elements is set by the constant:

```
op bucketDim : -> Nat .
```

The routing table offers the following operations:

- `op move-to-tail :`
  ```
  Triple Triple RoutingTable
  -> RoutingTable .
  ```
  Moves a contact to the tail of its k-bucket.
- `op add-entry :`
  ```
  Triple Triple RoutingTable
  -> RoutingTable .
  ```
  Adds an entry to a routing table.
- `op free-bucket :`
  ```
  Triple BitString RoutingTable -> Bool .
  ```
  Checks if a k-bucket is full.
- `op closest-nodes :`
  ```
  BitString RoutingTable Nat
  -> List{Triple} .
  ```
  Returns the list of the n closest contacts to a given node in the routing table.

3.2. Shared files

There are two 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] .
```

Operations on this abstract data type (ADT) include the typical operations for tables plus an operation to handle the time. The generic definition of the table 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] .
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 _monus_ : Table{X,Y} Time
            -> Table{X,Y} .
```

```
op key? : InfoTable{X,Y} -> X$Elt .
op value? : InfoTable{X,Y} -> Y$Elt .
```

The concrete table used to represent the files a peer publishes is:

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

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 __ : Triple TimeInf -> InfoFileTable
                                 [ctor] .
```

The table is

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

To publish a file, a node has to find the k nodes with the *closest* key to the file ID. As the information in the node's routing table may not include the *closest* nodes, it should search for them. Now, we follow the eMule implementation of the process. The node looks in the routing table for contacts that are as near as possible to the file key and keeps them, ordered by distance to the file key, in a temporary list. The information for each contact in the temporary list is its key, and the time passed since the RPC was sent. 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.

```
sort Node-Time .
sort TemporaryList .
subsort Node-Time < TemporaryList .
--- 1 param : Node ID (key, IP port and UDP)
--- 2 param : Distance to the search key
--- 3 param : Time since the RPC was send
--- 4 param : Flag
---    0 indicates the RPC was not sent,
---    1 the RPC was sent,
---    2 the RPC has responded,
---    3 the store message is sent
op <____> : Triple Nat TimeInf Nat
            -> Node-Time [ctor] .
op empty-list : -> TemporaryList [ctor] .
op insert : Node-Time TemporaryList
            -> TemporaryList [ctor] .
```

3.3. Searched files

One of the tasks 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 .
--- Time for expiration:
--- 0 has already been searched and found.
---   > 0 < 50 is ready to be searched.
---   > 50 the file is waiting.
op _;_ : String TimeInf -> InfoSearchFile
                                   [ctor] .
```

The table is

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

4. Modeling time

Simulating the behaviour 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* 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. For connected peers (`Life` > 0), it changes the time to republish a file (attributes `Files` and `Publish`), the time left to obtain a response in the temporary search list (attribute `SearchList`) and the time left to disconnect the peer (attribute `Life`). For disconnected peers, only the time to reconnect is changed.

  ```
  eq delta
    (< P1 : Peer | RT : R1 ,
        Files : FT1 , Publish : PF,
        SearchFiles : SF , SearchList : SL,
        Life : K1 , Reconnect : INF >,TM)
    =
    < P1 : Peer | RT : R1 ,
        Files : FT1 monus TM ,
        Publish : PF monus TM ,
        SearchFiles : SF monus TM,
        SearchList : SL monus TM,
        Life : K1 monus TM ,
        Reconnect : INF > .
  ```

Respect to messages, only the time to attend the message is changed. If time is set to zero the message is removed from the system.

```
eq delta(
  PING(SENDER,RECEIVER,TM1,1),TM) =
  PING(SENDER,RECEIVER,TM1 monus TM, 0) .
```

5. Network processes

We present two processes:

5.1. Looking for a file

The searching process starts automatically when there are IDs in the `SearchFiles` attribute of some peer that we will call the initiator. In this version we permit only one search per node at a time. The life time of the initiator, `K1`, should be greater than zero; otherwise, the node is supposed to be disconnected. The expiration time, `TM1`, should be greater than zero since the zero value indicates that the search has finished or no peer has found the file. It should also be less that n, set to 50 at this time, since a greater value indicates that the file has been already searched for but it was not found and now is waiting to repeat the search.

```
crl [lookfor-file1] :
 < SENDER : Peer |  RT : R1 ,
   SearchFiles : < I1 (S1 ; TM1) > SF ,
   SearchList : empty-list , Life : K1 >
=>
 < SENDER : Peer | RT : R1 ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : create-search-list(
             closest-nodes(I1,R1,10), I1) ,
   Life : K1 >
if K1 > 0 /\ K1 =/= INF /\ TM1 > 0 /\
   TM1 < 50 .
```

The expiration time of the search file is set to `INF` to indicate that the process is initiated. The search list is filled with the *closest* nodes the initiator has in its routing table. The `closest-nodes` operation returns the n closest nodes to the key `I1` in the routing table `R1`. We set the number of initial nodes to ten due to the size of our testing network. The list is created with the operation `create-search-list`, which inserts the nodes ordered by its distance to the key.

The process continues by sending `FIND-VALUE` RPCs to the first nodes of the list to find *closer* nodes to the file ID. We may have up to three active RPCs at the same time.

```
crl [lookfor-file21] :
 < SENDER : Peer |  RT : R1 ,
  SearchFiles : < I1 (S1 ; INF) > SF ,
  SearchList : SL , Life : K1 >
=>
 < SENDER : Peer | RT : R1 ,
  SearchFiles : < I1 (S1 ; INF) > SF ,
  SearchList : set-flag-process(Tr,SL) ,
  Life : K1 >
 FIND-VALUE(SENDER,Tr, I1, K1, 1)
if  K1 > 0 /\ K1 =/= INF /\
    not all-done(SL) /\
    Tr := first-not-sent(SL) /\
    messages-in-process(SL) < 3 .
```

Once the RPC is sent, a flag is activated in the search list that marks this node as *in process*. The RPC is only sent if the initiator is active and if there are still nodes in the search list to which no RPC has been sent. 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.

The receiver may find the value, or it may return the closest nodes it knows about. If the message is not attended, it is removed from the system.

```
crl [find-value1] :
  < RECEIVER : Peer | RT : R2 ,
   Files : FT2 , Life : K2 >
  FIND-VALUE(SENDER, RECEIVER, P3, TM,0)
=>
  < RECEIVER : Peer |
   RT : move-to-tail(SENDER,RECEIVER,R2) ,
   Files : FT2 , Life : K2 >
  FIND-VALUE-REPLY1(RECEIVER,SENDER,
   closest-nodes(P3,R2,bucketDim), K2,1)
if K2 > 0 /\ K2 =/= INF /\
   not (P3 in FT2) .

crl [find-value2] :
  < RECEIVER : Peer | RT : R2 ,
   Files : FT2 , Life : K2 >
  FIND-VALUE(
     SENDER, RECEIVER, P3, TM,0)
=>
  < RECEIVER : Peer |
   RT : move-to-tail(SENDER,RECEIVER,R2) ,
   Files : FT2 , Life : K2 >
  FIND-VALUE-REPLY2(RECEIVER,
    SENDER,ID?(first?(find(P3,FT2))), K2,0)
if K2 > 0 /\ K2 =/= INF /\ (P3 in FT2) .

rl [find-value3] :
  FIND-VALUE(
     SENDER, RECEIVER, P3, 0, 0)
=>
  none .
```

If the initiator receives the node that shares the file, the process ends.

```
crl [lookfor-file3] :
  < RECEIVER : Peer | RT : R2 ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : SL , Life : K2 >
  FIND-VALUE-REPLY2(
     SENDER,RECEIVER,P3,TM1,TM2)
=>
  < RECEIVER : Peer |
   RT : move-to-tail(SENDER,RECEIVER,R2) ,
   SearchFiles : < I1 (S1 ; 0) > SF ,
   SearchList : empty-list , Life : K2 >
 FIND-VALUE-REPLY2(
    SENDER,RECEIVER,P3,TM1,TM2)
if K2 > 0 /\ K2 =/= INF .
```

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.

```
crl [lookfor-file41] :
  < RECEIVER : Peer | RT : R2 ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : SL , Life : K2 >
  FIND-VALUE-REPLY1(
     SENDER,RECEIVER,Tr L,TM1,TM2)
=>
  < RECEIVER : Peer |
   RT : move-to-tail(SENDER,RECEIVER,R2) ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : insertOrd(
      < Tr distance(ID?(Tr),I1) 100 0 >,SL) ,
   Life : K2 >
  FIND-VALUE-REPLY1(
     SENDER,RECEIVER,L,TM1,TM2)
if K2 > 0 /\ K2 =/= INF /\
   distance(ID?(Tr),I1) <
     distance(ID?(SENDER),I1) /\
   SL =/= empty-list .

crl [lookfor-file42] :
  < RECEIVER : Peer | RT : R2 ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : SL , Life : K2 >
  FIND-VALUE-REPLY1(
     SENDER,RECEIVER,Tr L,TM1,TM2)
=>
  < RECEIVER : Peer |
   RT : move-to-tail(SENDER,RECEIVER,R2) ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : SL , Life : K2 >
  FIND-VALUE-REPLY1(
     SENDER,RECEIVER,L,TM1,TM2)
if K2 > 0 /\ K2 =/= INF /\
   distance(ID?(Tr),I1) >=
     distance(ID?(SENDER),I1) /\
   SL =/= empty-list .
---
crl [lookfor-file43] :
  < RECEIVER : Peer | RT : R2 ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : SL , Life : K2 >
  FIND-VALUE-REPLY1(
     SENDER,RECEIVER,nil,TM1,TM2)
=>
  < RECEIVER : Peer |
   RT : move-to-tail(SENDER,RECEIVER,R2) ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : set-flag-done(SENDER,SL) ,
   Life : K2 >
if K2 > 0 /\ K2 =/= INF /\
   SL =/= empty-list .
```

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.

```
crl [lookfor-file5] :
  < SENDER : Peer | RT : R1 ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : SL , Life : K1 >
=>
  < SENDER : Peer | RT : R1 ,
   SearchFiles : < I1 (S1 ; INF) > SF ,
   SearchList : remove-time0(SL) ,
   Life : K1 >
if K1 > 0 /\ K1 =/= INF /\
   messages-time0(SL) > 0 .
```

5.1.1. Publishing a file

Publish 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 zero. It can be a node's shared file kept in the *publish files* table or a known file shared by other node.

```
crl [publish11] :
  < SENDER : Peer | RT : R1 ,
   Publish : < I1 (S1 0) > PF ,
   SearchList : empty-list ,
   Life : K1 >
=>
  < SENDER : Peer | RT : R1 ,
   Publish : < I1 (S1 INF) > PF ,
   SearchList : create-search-list(
      closest-nodes(I1,R1,10), I1) ,
   Life : K1 >
if K1 > 0 /\ K1 =/= INF .
```

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 creates the temporary list and sends `FIND-NODE` RPCs to the closest nodes. Since the process is similar to the one explained for looking for a file we only present the rules once all nodes have replied or they have been removed from the list because their response time has expired. Then, a `STORE` message is sent to the first three nodes of the list, that are supposed to be the closest to the file's ID. When the three `STORE` messages have been sent the time to republish the file is set to k.

```
crl [publish51] :
  < SENDER : Peer | RT : R1 ,
   Publish : < I1 (S1 INF) > PF ,
   SearchList : SL , Life : K1 >
=>
  STORE(SENDER,first-not-stored(SL),
        < I1 (SENDER 100) >,K1,1)
  < SENDER : Peer | RT : R1 ,
   Publish : < I1 (S1 INF) > PF ,
   SearchList : set-flag-store(
     first-not-stored(SL),SL) ,
   Life : K1 >
if K1 > 0 /\ K1 =/= INF /\
   all-done(SL) /\
   number-messages-store(SL) < 3 .

crl [publish52] :
  < SENDER : Peer | RT : R1 ,
   Publish : < I1 (S1 INF) > PF ,
   SearchList : SL , Life : K1 >
=>
  < SENDER : Peer | RT : R1 ,
   Publish : < I1 (S1 100) > PF ,
   SearchList : empty-list ,
   Life : K1 >
if K1 > 0 /\ K1 =/= INF /\
   (number-messages-store(SL) == 3 or
    length(SL) == number-messages-store(SL)) .
```

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].

Referencias

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

[2] Bakhshi, R., and Gurov, D. Verification of Peer-to-peer Algoritms: A case Study. ENTCS 181, pages 35—47. Elsevier, 2007.

[3] Borgström J., Nestmann U., Onana L. and Gurov D. Verifying a Structured Peer-to-peer Overlay Network: The Static Case *In Proceedings of Global Computing 2004*, LNCS 3267, pages 251-266. Springer 2004

[4] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Tal-

cott, C. 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] eMule http://www.emule-project.net.

[7] EU-project PEPITO: IST-2001-33234. Homepage: http://www.sics.se/pepito/

[8] Maymounkov, P., and David Mazieres, D. Kademlia: A peer-to-peer Information System Based on the XOR Metric. *In Proceedings of the 1st International Workshop on Peer-to Peer Systems (IPTPS02)*, 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] Mysicka D. eMule Attacks and Measurements. Master Thesis. Dept. of Computer Science, Distributed Computing group, (ETH) Zurich. 2007.

[12] Ölveczky, P., and Meseguer, J., Semantics and pragmatics of Real-Time Maude, *Higher Order Symbol. Comput.*, volume 20, number 1-2, pages 161–196. Kluwer Academic Publishers. 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] Ratnasamy S, Francis P, Handley M, Karp R, and Shenker S. A Scalable Content-Addressable Network. *In Proceedings of SIGCOMM*, 2001.

[15] Rowstron A, and Druschel P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001. *Proceedings In Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (2001)*, pp. 329-350. 2001.

[16] 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.

[17] 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.

[18] Stoica I, Morris R, Karger D, Kaashoek M, and Balakrishnan H. Chord: A scalable peer-to-peer lookup service for Internet applications. IEEE/ACM Trans. Netw., volume 11, number 1, pages 17–32. 2003.

[19] Verdejo A., Pita I. and Martí-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.

[20] 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.