

Specifying and Analyzing the Kademlia Protocol in Maude[★]

Isabel Pita and Adrián Riesco

Facultad de Informática, Universidad Complutense de Madrid, Spain
ipandreu@sip.ucm.es, ariesco@fdi.ucm.es

Abstract. Kademlia is a peer-to-peer distributed hash table (DHT) currently used in the P2P eDonkey file sharing network. Kademlia 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 the 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 (or almost *closest* node, if the information is replicated) 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.

This paper presents the distributed specification of the behavior of a P2P network that uses the Kademlia DHT in the formal specification language Maude. We use sockets to connect different Maude instances and create a P2P network where the Kademlia protocol can be used. This protocol is executed on top of a previously developed routing protocol that provides real-time by connecting Maude to an external Java server and allows peers to enter and leave the network dynamically. Then, we show how to represent this distributed system in one single term in order to simulate and analyze the system using *Real-Time Maude*.

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

1 Introduction

Distributed Hash tables (DHTs) are becoming an essential factor in the implementation of P2P networks since the Kad DHT was incorporated in the eMule client [3] of the eDonkey file sharing network. Previously a large number of DHTs were studied through theoretical simulations and analysis, such as Chord [21], CAN [17], Pastry [20], and also Kademlia [9], in which the Kad DHT is based.

P2P networks are mainly used for file sharing applications, due to its lack of security. The large number of users involved in the networks and the absence of a central authority that certifies the trust of the participating nodes imply that the system must be able to operate even though some participants are malicious.

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

Several attacks to P2P networks have been studied in the literature [22,4], in particular, the problem of ensuring efficient and correct peer discovery despite adversarial interference like the Sybil attack or the Eclipse attack. However, the majority of these studies examine the problems, drawing examples from existing systems, or experimentally evaluate the attacks over the networks. Formal methods, which have had a great success in the analysis of distributed networks and protocols, have almost not contributed to P2P networks, nor to the DHT implementations. One of the first formal proposals is due to Borgström et al., who prove in [2] 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. Moreover, Bakhshi and Gurov, give in [1] a formal verification of Chord's stabilization algorithm using the π -calculus. Lately Lu, Merz, and Weidenbach [8] have modeled Pastry's core routing algorithms in the specification language TLA^+ and have proved properties of correctness and consistency using its model checker.

This paper presents a distributed specification in Maude [5], a formal specification language based on rewriting logic, of the behavior of a P2P network that uses the Kademlia DHT. Rewriting logic [10] was proposed in the early nineties as a unified model for concurrency in which several well-known models of concurrent and distributed systems can be represented. The specification language Maude supports both equational and rewriting logic computations. It can be used to specify in a natural way a wide range of software models and systems and, since (most of) the specifications are directly executable, Maude can be used to prototype those systems. Moreover, the Maude system includes a series of tools for formally analyzing the specifications. Since version 2.2, Maude supports communication with external objects by means of TCP sockets, which allows the implementation of real distributed applications. Real-Time Maude [15] is a natural extension of the Maude language that supports the specification and analysis of real-time systems, including object-oriented distributed ones. It supports a wide spectrum of formal methods, including: executable specification, symbolic simulation, breadth-first search for failures of safety properties in infinite-state systems, and linear temporal logic model checking of time-bounded LTL formulas. Real-Time Maude has strengthened that analyzing power by allowing to specify sometimes crucial timing aspects. It has been used, for example, to specify the Enhanced Interior Gateway Routing Protocol (EIGRP) [19], embedded systems [14], and the AER/NCA active network protocol [12]. Moreover, analysis of real-time systems using Maude sockets, and thus requiring a special treatment for them, has been studied [?,?]. While the algebraic representation of the distribution used in these works follows, as well as our work, the approach presented in [19], the way used to relate logical and physical time allows a more precise and formal analysis than the one used here, allowing the system to synchronize only when needed. We consider this approach an interesting subject of future work.

As part of an ongoing work to develop distributed applications in Maude, we specified a distributed version of the mobile agents language Mobile Maude and

algorithmic skeletons in [18]. Both specifications used Maude sockets to connect different Maude instances. These systems were executed on different topologies, being available star, ring, and centralized ring architectures. These topologies were improved in a later work when we developed the Enhanced Interior Gateway Routing Protocol (EIGRP) in Maude [19]. This protocol allowed us to build dynamic (nodes can be added anytime) and reconfigurable (when a node leaves the network alternative paths are found) topologies, on top of which we can execute any distributed Maude specification. This work used Maude sockets to, in addition to connecting different Maude instances, connect Maude to a Java server that communicates the time elapsed to the Maude specification. In this way, it was possible to use Real-Time in Maude for analyzing the system.

In this paper we present, from the point of view of the implementation of distributed applications in Maude, the first system implemented on top of the routing protocol described in [19]. Our implementation of the Kademlia protocol uses real-time by sharing the “tick” messages sent from the Java server and allows peers to be added and removed dynamically, while the underlying protocol takes care of redirecting the messages. In this way we show that “real” distributed applications can be implemented in Maude in an incremental and easy way. From the point of view of the analysis, this distributed system can be simulated and analyzed in Maude by using an algebraic specification of the sockets provided by Maude; an abstraction of the underlying routing protocol, which allows the analysis tools to focus on the properties; and by using Real-Time Maude. That is, we abstract some implementation details but leave the protocol implementation unmodified, which allows us to use the centralized protocol to prove properties that must also hold in the distributed version. The analyses that can be performed on the protocol include the simulation of the system to study, for example, how its properties change when its parameters, like the redundancy constant, are modified; examine the reaction of the system to different attacks; and check properties such as that any published file can be found or that files remain accessible even if their publishing peers become offline.

The use of formal methods to describe the behaviour of the Kademlia DHT may help to understand the informal description in [9]. In particular, the Maude language, that we proposed, gives us the opportunity of executing the distributed specification taking into account the time aspects of the protocol. It also allows us to analyze all possible executions of the system, using the centralized model that mirrors the distributed one, either by searching in the execution tree or by using model checking techniques.

The rest of the paper is structured as follows: Section 2 presents how to specify generic distributed systems in Maude, and the Kademlia protocol. Section 3 describes the distributed specification in Maude of this protocol. Section 4 shows how the distributed system can be represented in one single term, while Section 5 describes how to simulate and analyze it. Finally, Section 6 concludes and presents some future work.

2 Preliminaries

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

2.1 Maude

In Maude [5] the state of a system is formally specified as an algebraic data type by means of an equational specification. In this kind of specification we can define new types (by means of keyword `sort(s)`); subtype relations between types (`subsort`); operators (`op`) for building values of these types; and equations (`eq`) that identify terms built with these operators.

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

In object-oriented specifications, *classes* are declared with the syntax `class C | a1:S1, ..., an:Sn`, where C is the class name, a_i is an attribute identifier, and S_i is the sort of the values this attribute can have. An *object* is represented as a term $\langle O : C \mid a_1 : v_1, \dots, a_n : v_n \rangle$ where O is the object's name, belonging to a set `Obj` of object identifiers, and the v_i 's are the current values of its attributes. *Messages* are defined by the user for each application (introduced with syntax `msg`).

In a concurrent object-oriented system the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting. The rewrite rules specify the behavior associated with the messages. By convention, the only object attributes made explicit in a rule are those relevant for that rule. We use Full Maude's object-oriented notation and conventions [5] throughout the whole paper; however, only the centralized specification is specified in Full Maude (which is required by Real-Time Maude), while the actual implementation of the distributed protocol is in Core Maude because Full Maude does not support external objects. The complete Maude code can be found at <http://maude.sip.ucm.es/kademlia>.

In [19], we described a methodology to implement distributed applications in such a way that the distributed behavior remains transparent to the user by using a routing protocol, the Enhanced Interior Gateway Routing Protocol (EIGRP). Figure 1 presents the architecture proposed in that paper, where the lower layer provides mechanisms to translate Maude messages from and to String (Maude sockets can only transmit Strings); to do so, the user must instantiate a theory requiring a (meta-represented) module with the syntax of all the transmitted messages. The intermediate layer, EIGRP, provides a message of the form `to_:_`, with the first argument an object identifier (the addressee of the message) and the second one a term of sort `TravelingContents`, that must be defined in each specific application. We have slightly modified this layer to share the `tick!`

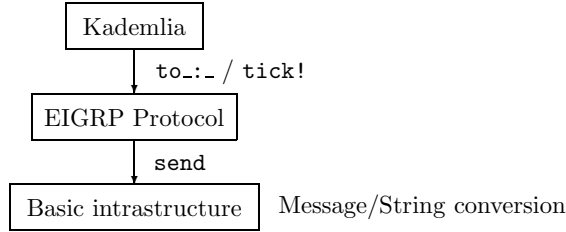


Fig. 1. Layers for distributed applications

message obtained from the Java server in charge of dealing with time.¹ This layer provides a fault-tolerant and dynamic architecture where nodes may join and leave at any moment, and where nodes are always reached by using the shortest path, thus allowing us to implement realistic systems. Finally, the upper layer is the application one, which in our case corresponds to Kademlia. It relies on the lower layers to deliver the messages and focus on its specific tasks, just like the real Kademlia protocol.

2.2 Kademlia

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

Each node stores contact information about others. In Kademlia, every node keeps a list of: IP address, UDP port and node ID, for nodes of distance between 2^i and 2^{i+1} from itself, for $i = 0, \dots, n$ and n the ID length. In the Kademlia paper [9] these lists, called k -buckets, have at most k elements, where k is chosen such that any given k nodes are very unlikely to fail within an hour of each other. k -buckets are kept sorted by the time they were 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 at the head of the list responds, it is moved to the tail, and the new

¹ In the standard implementation, **tick!** messages are introduced into the configuration each second. However, the time can be customized to get these messages in the time span defined by the user.

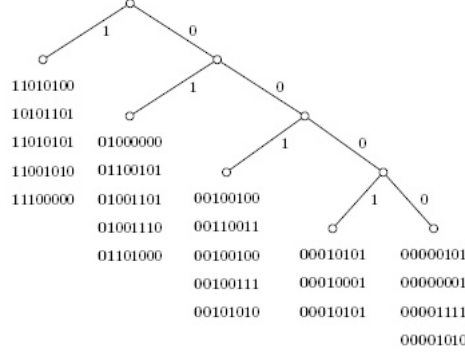


Fig. 2. A routing table example for node 00000000

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. [6] 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. Figure 2 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** checks whether a node is online.
- **STORE** instructs a node to store a file ID together with the contact of the node that shares the file to publish it to other nodes.
- **FIND-NODE** takes an ID as argument and the recipient returns the contacts of the k nodes it knows that are 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 that are 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 [9].

Looking for a value. To find a file ID, a node starts by performing a lookup to find the k nodes with the 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 one already seen, the initiator resends the **FIND-VALUE** to all of the k closest nodes it has not queried yet. The process terminates when any node returns the value or when the peer that started the query has obtained the responses from its k closest nodes.

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 from time to time. Nodes that know about a file publish it more frequently than the node that shares it.

To share 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 k closest nodes, it sends them a **STORE** RPC.

3 Distributed Implementation

We present in this section the main details of the distributed implementation of the Kademlia protocol. A more detailed explanation of the data types, classes, and messages in the specification may be found in [16].

The Kademlia network is modeled as a Maude configuration of objects and messages. Peers in our specification are objects of class **Peer**, defined as follows:

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

where the object identifier is the node ID, which we have simplified from its original version and is expressed with the operator **peer**, that takes a natural number, which is the decimal representation of the reverse of the node n -bit ID, as argument. The attributes related to the Kademlia network are:

- **RT** is a list that keeps the information of the routing table.
- **Files** is a table that keeps the information of the files the peer is responsible for publishing. It includes the file ID, the identification of the peer that shares the file, a time for republishing the file and keep it alive, and a time to remove the file from the table.
- **Publish** is a table that keeps the information of the files shared by the peer. The information includes the file ID, the file's location in the peer and a time for republishing the file. This time is greater than the time for republishing of the **Files** table, and prevents the information in the **Files** table from being removed.
- **SearchFiles** is a table that keeps the files a peer is looking for. The information includes the file ID, and a waiting time to proceed with the search. This time is used when the file is not found and it should be researched later.

- **SearchList** is an auxiliary list used in the search and publish processes to keep the information of the nodes that have been already contacted by the searcher/publisher and the state in which the searching/publishing process is. As the searcher/publisher finds out new *closer* nodes to the file ID, it stores them in this file, and starts sending them messages.

The messages represent the RPCs. There is a message for each RPC defined in the Kademlia protocol. For example, the **FIND-NODE** message and its reply are defined as follows:

```
op FIND-NODE : Nat Nat -> TravelingContents [ctor] .
op FIND-NODE-REPLY : Nat Nat NatList -> TravelingContents [ctor] .
```

Note that terms of this form will be used to form messages with the operator **to_:** described in Section 2.1, where the first parameter is the identifier of the addressee. The first parameter of these operators identifies the peer sending the message, while the second one represents the key the sender is looking for. The reply has also an additional parameter that keeps a list of the k nodes the peer knows that are the closest ones to the target, where k is the bucket dimension.

The specification of the different processes follows their definition. For example, the searching process starts automatically when there are IDs in the **SearchFiles** attribute of some connected peer with time for searching equal to one. A greater value indicates that the file has already been searched for, it was not found, and now it is waiting for repeating the search. When the search starts, the auxiliary list **SearchList** is filled with the *closest* nodes the searcher has in its routing table, and the time of this file in the **searchFiles** table is set to INF. It will remain with this value until the search process ends.

The process continues by sending **FIND-VALUE** RPCs to the first nodes in the list to find *closer* nodes to the file ID. The RPC is only sent if the number of parallel messages is less than the given constant, **ParallelSearchRPC**, the peer in charge of the search has not received response yet from a certain number of peers given by the **kSearched** constant, and there are nodes in the search list that have not been contacted yet. Notice that we have to ask as many nodes as possible, because there can be nodes not so close to the objective as others but that have in their routing tables information of the closest ones. Once the RPC is sent, a flag is activated in the search list that marks this node as *in process* with **set-flag**:

```
crl [lookfor-file21] :
  < peer(SENDER) : Peer | SearchFiles : < I1 & (S1 ; INF) > # SF,
    SearchList : SL >
=> < peer(SENDER) : Peer | SearchFiles : < I1 & (S1 ; INF) > # SF,
    SearchList : set-flag(Tr,SrchListRmve,SL) >
  to peer(Tr) : FIND-VALUE(SENDER, I1)
if not all-sended(SL) /\ Tr := first-not-send(SL) /\
  messages-in-process(SL) < ParallelSearchRPC /\
  number-nodes-reply(SL) < kSearched .
```

The receiver may find the file the searcher is looking for in his table or it may return the closest nodes it knows about. In the first case, it sends a **FIND-VALUE-REPLY2** message to the searcher including the node ID of the peer that shares the file. When the searcher receives this reply the process finishes by sending a **FILE-FOUND** message and the file is removed from its searching table. The **FILE-FOUND** message remains in the configuration to show the files that have been searched and found, easing the proof of properties. In the second case, the receiver sends a **FIND-VALUE-REPLY1** message to the searcher including the closest nodes to the file ID it knows about. When the searcher receives this message 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. When the full list is traversed, a flag is activated to mark this node as done in the search list. Additionally, the searcher routing table is updated with the **move-to-tail** operation that puts the ID of the message sender first in the list, so that it will not be removed from the routing table, as it is the last peer the searcher knows it is alive:

```

r1 [lookfor-file3] :
  to peer(REC) : FIND-VALUE-REPLY2(SENDER, I1, P3)
  < peer(REC) : Peer | RT : R1, SearchList : SL,
    SearchFiles : < I1 & (S1 ; INF) > # SF >
=> < peer(REC) : Peer | RT : move-to-tail(SENDER, REC,R1),
    SearchList : mt-list, SearchFiles : SF >
  to peer(REC) : FILE-FOUND(SENDER,I1) .

r1 [lookfor-file40] :
  to peer(REC) : FIND-VALUE-REPLY1(SENDER, P3, L)
  < peer(REC) : Peer | RT : R1, SearchList : SL,
    SearchFiles : < I1 & (S1 ; INF) > # SF >
=> < peer(REC) : Peer | RT : move-to-tail(SENDER, REC,R1),
    SearchFiles : < I1 & (S1 ; INF) > # SF,
    SearchList : insert(L,SL,SENDER,REC,P3) > .

```

Another important rule in the system consists of updating the peer when a **tick!** message arrives. In order to treat similarly the distributed and centralized versions, we define a **delta** function [13] on peers that updates the time-related attributes:

```

r1 [tick!]:
  tick! < 0 : Peer | >
=> delta(< 0 : Peer | >, 1) .

```

where **delta** just applies auxiliary **delta** functions to each of the attributes. These auxiliary functions just traverse the tables and lists, decreasing the time stored on the appropriate fields:

```

eq delta(< 0 : Peer | Files : FT1, Publish : PF, SearchFiles : SF,
    SearchList : SL >, TC) =
  < 0 : Peer | Files : delta(FT1, TC), Publish : delta(PF, TC),

```

```

SearchFiles : delta(SF, TC),
SearchList : delta(SL, TC) > .

```

Executable examples of the distributed protocol can be found at <http://maude.sip.ucm.es/kademlia>.

4 Centralized Simulation

We use Real-Time Maude [13,15] to specify our timed system. It declares modules defining the natural numbers as the time values of sort `Time`, with operations like `plus`, `<=`, `monus`, and a supersort `TimeInf`, which contains the constant `INF` representing ∞ [13]. To ensure that time advances uniformly in all the parts of a state, a new sort `GlobalSystem` is used, with constructor `{_}` : `System` \rightarrow `GlobalSystem`.

In Real-Time Maude an object-oriented system is represented as a term of sort `Configuration` (a subsort of `System`) and, since it has a rich structure, it is useful to have an explicit operation `delta`, that defines the effect of time elapse on each object and message in a configuration. An operation `mte` giving the maximum time elapse permissible to ensure timeliness of time-critical actions, and defined separately for each object and message, is also useful. Then, time elapse is modeled by the tick rule

```

crl [tick] : { SYSTEM }
=> { delta(SYSTEM, T) } in time T
if T <= mte(SYSTEM) [nonexec] .

```

Real-Time Maude deals with in principle non-executable tick rules by offering a choice of different “time sampling” strategies, so that instead of covering the whole time domain, only some moments are visited. We have selected the sampling strategy that advances time by the maximal possible amount.

Thus, in order to use the analysis features provided by Real-Time Maude, we need to represent the distributed configuration described in the previous section as a single term and define the appropriate `delta` and `mte` functions. We achieve the former by following a similar approach to the one we followed in [18,19]. We provide a class `Process` with a single attribute `conf` that keeps the configurations in different locations² separated from each other:

```

class Process | conf : Configuration .

```

The `mte` function applied to objects of this class returns the `mte` of the configuration, while `delta` is also applied to the objects in this attribute. We also provide an algebraic specification of the built-in sockets. In our case, we use an object of class `Socket` for each two connected locations in the distributed (real) protocol. This class has attributes `sideA` and `sideB`, indicating the two

² We will use the word *location* to denote the different Maude instances appearing in the distributed system.

sides of the socket; **delay**, which stores the delay associated to this socket; and **listA** and **listB**, the lists of **DelayedMsg** (pairs of messages and time) sent to **sideA** and **sideB**, respectively:

```
class Socket | sideA : Oid, sideB : Oid, delay : Time,
               listA : List{DelayedMsg}, listB : List{DelayedMsg} .
```

In this way, we can simulate the delay due to the network and specify the architecture with only four rules, two for moving messages into the socket and two more for putting the messages into the target configuration, depending of the side of the socket. For example, the rule moving a message from the list to the side of the socket indicated by **sideA** is specified as follows, where it is important to note that the time of the element being moved has reached 0:

```
rl [receive1] :
  < S : Socket | sideA : 0, listA : dl(to 0' : TC, 0) DML >
  < 0 : Process | conf : CONF >
=> < S : Socket | listA : DML >
   < 0 : Process | conf : (to 0' : TC CONF) > .
```

The **mte** function applied to sockets returns the minimum time required by the messages in the head of the lists, while **delta** updates the messages in the lists. That is, we consider that this abstract architecture is initially connected (there exists an object of class **Socket**) for each pair of locations connected, directly or indirectly, in the real system, and it never fails, that is, connections are never broken.³ In this way we prevent the search and model-checking commands from using the rules for connecting the locations, for redirecting messages, and for transforming the messages from/into **String**, which introduces a lot of non-determinism (although the order in the creation of the connections and in the redirection of messages is not important, these tools must traverse all the possible paths to check the given properties).

In order to simulate errors and disconnections in the peers we have added two attributes to the **Peer** class: **Life** and **Reconnect**, containing values of sort **TimeInf**. Basically, when the **Life** attribute reaches the value 0, it is set to **INF**, the peer cannot receive nor send messages, and the **Reconnect** attribute is set to a random value. Similarly, when **Reconnect** reaches 0, it is set to **INF**, **Life** is set to a random time, and the peer works again. Once this class has been modified, we only need to (i) define the **mte** function on peers and messages; (ii) modify the **delta** function on peers to take the **Life** and **Reconnect** values into account, and define it over messages; (iii) add a condition in each rule saying that the value in **Life** is greater than 0 and different from **INF**; and (iv) add the rules for disconnecting and reconnecting the peer. For messages, **mte** and **delta** are easily defined; **mte** returns 0 for all of them except for **FILE-FOUND**, that is

³ Note that errors in the connections will be simulated by making the peers to fail, as we will see later. It is anyway easily simulated in sockets e.g. by adding an attribute **numMsgs** indicating the number of messages that can be sent through the socket before it fails; see [19] for details.

just used for information purposes and thus it is not processed,⁴ while `delta` does not modify them:

```
eq mte(to 0 : FILE-FOUND(SENDER,I1)) = INF .
eq mte(MSG) = 0 [otherwise] .
eq delta(MSG, T) = MSG .
```

As explained above, the `delta` function for peers is very similar to the one presented in the previous section, just including (and updating) the new attributes, and thus we focus on `mte`. When the peer is connected, we return the minimum between all the other time-related attributes, while when it is disconnected (`K2` is a variable of sort `Nat`) then nothing else “works” and `mte` returns the time left for reconnecting:

```
eq mte(< 0 : Peer | Files : FT1, Publish : PF, SearchFiles : SF,
      Life : K1, Reconnect : INF >) =
  min(minTime(FT1), min(minTime(PF), min(minTime(SF), K1))) .
eq mte(< 0 : Peer | Reconnect : K2 >) = K2 .
```

where the functions `minTime` computes the minimum time used in the tables.

5 Protocol Properties

We can use now Real-Time Maude in two different ways: to execute the centralized specification and to verify different properties. The former is achieved by using the commands `trew` and `tfrew`, that execute the system (the second one applies the rules in a *fair* way) given a bound in the time; with `find earliest` and `find latest`, that allow the user to check the paths that lead to the first and last (in terms of time) state fulfilling a given property; and with `tsearch`, that checks whether a given state is reachable in the given time. The latter is accomplished by using the `tsearch` command to check that an invariant holds; by looking for the negation of the invariant we can examine whether there is a reachable state that violates it. The specification can also be analyzed by using timed model checking with the command `mc`, that allows the user to state linear temporal logic formulas with a bound in the time.

We will use in this section two network topologies as case studies, of 6 and 20 nodes. We abstract the concrete connections and suppose total network connectivity. The life time of each node is randomly chosen, although we use an upper bound life constant to control the ratio of alive nodes. We change the peers that share and search files, as well as the number and time of published and searched files.

We can simulate how different attacks may affect a network. For example, in the *node insertion* attack, an attacking peer catch search requests for a file, which are answered with bogus information [11]. The attacking peer creates its own

⁴ We assume here that messages are attended as soon as they are received. For this reason, a rule is in charge of deleting messages addressed to inactive peers.

ID such that it matches the hash value of the file. Then the search requests are routed to the attacking peer, that may return its own file instead of routing the search to the original one. Since the Kademlia network sends the request not only to the closest peer the searcher may find the original file. The `find earliest` command can be used to study different network parameters and check whether this attack is effective. We study if a file may be found in a node that is not the closest one to the file ID, with the following Real-Time Maude command:

```
Maude> (find earliest init =>* {< 0 : Process | conf :
      (to 0' : FILE-FOUND(SENDER, N2) CONF) > CONF'} .)
```

Note that, since the `FILE-FOUND` message returns in its first parameter the peer that is publishing the file, we only need to check whether the peer ID is the closest to the file ID.

From the model-checking point of view, there are several properties that can be proved over this protocol. The basic property all P2P networks should fulfill is that if a peer looks for a file that is published somewhere, the peer eventually finds it. We define three propositions (of sort `Prop`, imported from the `TIMED-MODEL-CHECKER` module defined in Real-Time Maude) over the configuration expressing that a peer publishes a file; a peer is looking for that file; and the peer that searches the file finds it. Note that, as in the command above, all the properties are defined taking into account that the configurations are wrapped into objects of class `Process`, that may contain other objects and messages on the `conf` attribute (hence the `CONF` variable used there) and that other processes may also appear in the initial configuration (hence the `CONF'` variable used at the `Process` level):

```
op PublishAFile : Nat -> Prop [ctor] .
eq {< 0 : Process | conf : (< 0' : Peer | Publish :
  < I1 & (S1 @ TC4) > # PF > CONF) > CONF'} |= PublishAFile(I1) = true .

op SearchAFile : Nat Nat -> Prop [ctor] .
eq {< 0 : Process | conf : (< peer(N) : Peer | SearchFiles :
  < I1 & (S1 ; TC3) > # SF > CONF) > CONF'} |= SearchAFile(N,I1) = true .

op FindAFile : Nat Nat -> Prop [ctor] .
eq {< 0 : Process | conf : (to peer(Searcher) : FILE-FOUND(I2,I1)
  CONF) > CONF'} |= FindAFile(Searcher,I1) = true .
```

Assuming an initial configuration where a peer publishes the file 200, that is searched by `peer(22)`, we can use the following command to check that the property holds:

```
Maude> (mc init' |=t PublishAFile(200) /\ SearchAFile(22,200) =>
      <> FindAFile(22,200) in time < 20 .)
Result Bool : true
```

Another basic property is that once a file is published it remains published in some peers unless the publisher is disconnected. We can define the properties

`FilePublished`, stating that a peer publishes a file, and `PeerOffline`, indicating that a peer is offline, similarly to the properties above and use the following command to check the property:

```
Maude> (mc init |=t (<> [] (FilePublished(53,0)) U PeerOffline(0))
      in time < 40 .)
Result ModelCheckResult : counterexample(...)
```

In a network where `peer(0)` has published file 53. Notice that the model checker finds a counterexample. The reason is that all the peers that share the file may be offline at the same time. The property should be reformulated, stating that if the file is published it will always be published again or the publisher will be disconnected:

```
Maude> (mc init |=t ([] <> (FilePublished(53,0) \/ PeerOffline(0))
      in time < 40 .)
Result Bool : true
```

See <http://maude.sip.ucm.es/kademlia> to obtain the source code of the centralized specification, the complete initial terms used in the commands above, and the different properties proved.

6 Conclusions and Ongoing Work

We have presented in this paper a distributed implementation of the Kademlia protocol in Maude. This distributed system uses sockets to connect different Maude instances and, moreover, to connect each one of these instances to a Java server that provides `tick` messages notifying when a given amount of time has elapsed. It can be used to share files (only text files in the current specification) using this protocol, allowing peers to connect and disconnect in a dynamic way, adding and searching for new files. Moreover, we also provide a centralized specification of the system, which abstracts most of the details of the underlying architecture to focus on the Kademlia protocol. This centralized specification allows us to simulate and analyze the system using Real-Time Maude to represent the real time implemented in Java in the distributed version. We are currently working to improve the analysis by defining a more precise relation between physical and logical time, following the approach in [?,?].

There are some open issues from the model point of view. We want to incorporate more network processes, like the one that automatically connects a node to the network. 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 to allow the publication of keywords and notes related to files. There are also some protections eMule implements to protect itself against possible attacks, like the protection of hot nodes, that requires a deeper study. It will also be useful to compare the eMule implementation with the aMule and BitTorrent ones. Finally, we want to study and prove more complex properties over the protocol.

References

1. R. Bakhshi and D. Gurov. Verification of peer-to-peer algorithms: A case study. In *Combined Proceedings of the Second International Workshop on Coordination and Organization, CoOrg 2006, and the Second International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, MTCoord 2006*, volume 181 of *Electronic Notes in Theoretical Computer Science*, pages 35–47. Elsevier, 2007.
2. 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*, volume 3267 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2004.
3. H. Breitkreuz. The eMule project. <http://www.emule-project.net>.
4. E. Chan-Tin, P. Wang, J. Tyra, T. Malchow, D. F. Kune, N. Hopper, and Y. Kim. Attacking the Kad network—real world evaluation and high fidelity simulation using DVN. *Wiley Security and Communication Networks*, 2009.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
6. S. S. P. Gummadi and S. Gribble. 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.
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. T. Lu, S. Merz, and C. Weidenbach. Model checking the Pastry routing protocol. In J. Bendisposto, M. Leuschel, and M. Roggenbach, editors, *10th International Workshop Automatic Verification of Critical Systems, AVOCS 2010*, pages 19–21. Universität Düsseldorf, 2010.
9. P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In P. Druschel, M. F. Kaashoek, and A. I. T. Rowstron, editors, *Revised Papers from the First International Workshop on Peer-to-Peer Systems, IPTPS 2001*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2002.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
11. D. Mysicka. eMule attacks and measurements. Master’s thesis, Swiss Federal Institute of Technology (ETH) Zurich, 2007.
12. P. Ölveczky, J. Meseguer, and C. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29:253–293, 2006.
13. P. C. Ölveczky. *Real-Time Maude 2.3 Manual*, 2007. <http://heim.ifi.uio.no/~peterol/RealTimeMaude>.
14. P. C. Ölveczky. Formal model engineering for embedded systems using Real-Time Maude. In F. Durán and V. Rusu, editors, *Proceedings of the 2nd International Workshop on Algebraic Methods in Model-based Software Engineering, AMMSE 2011*, volume 56 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–13, 2011.
15. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20:161–196, 2007.

16. I. Pita. A formal specification of the Kademlia distributed hash table. In V. M. Gulías, J. Silva, and A. Villanueva, editors, *Proceedings of the 10 Spanish Workshop on Programming Languages, PROLE 2010*, pages 223–234. Ibergarceta Publicaciones, 2010. <http://www.maude.sip.ucm.es/kademlia>. Informal publication—Work in progress.
17. 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.
18. A. Riesco. Distributed and mobile applications in Maude. Master’s thesis, Departamento de Sistemas Informáticos y Computación, Facultad de Informática de la Universidad Complutense de Madrid, June 2007.
19. A. Riesco and A. Verdejo. Implementing and analyzing in Maude the enhanced interior gateway routing protocol. In G. Roşu, editor, *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008*, volume 238(3) of *Electronic Notes in Theoretical Computer Science*, pages 249–266. Elsevier, 2009.
20. A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In R. Guerraoui, editor, *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.
21. 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.
22. G. Urdaneta, G. Pierre, and M. van Steen. A survey of DHT security techniques. *ACM Computing Surveys*, 43(2), January 2011.