Specifying and Analyzing the Kademlia Protocol in Maude

Isabel Pita(✉) and Adrián Riesco

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

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 O(log(n)) steps, where n is the number of nodes in the network.

This paper presents 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 by 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.

Keywords: Kademlia · Distributed specification · Formal analysis · Maude · Real-Time Maude

1 Introduction

Kademlia based distributed hash tables (DHTs) [11] are an essential factor in the implementation of P2P networks since the Kad DHT was incorporated in the eMule client [5]. Among the large number of DHTs studied through theoretical simulations and analysis, such as Chord [25], CAN [21], or Pastry [24], Kademlia is the one that has been chosen for implementation of file sharing systems over large networks due to its relative simplicity. Some of its advantages are that there is only one routing algorithm from start to finish; it prevents a number of attacks by preferring long-standing nodes over new ones in the routing tables; and it allows nodes to learn about the network simply by participating in it.

Research supported by MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04) and Comunidad de Madrid program *N-GREENS Software-CM* (S2013/ICE-2731).

DOI: 10.1007/978-3-319-25150-9_30

DHTs are mainly used for file sharing applications and decentralized storage systems, due to its lack of security. On the one hand, the large number of users involved in the systems and the absence of a central authority certifying the trust of the participants suggests that the system must be able to operate even if some of them are malicious. On the other hand, the dynamics of the system, mainly the arrival and departure of nodes in P2P networks, and the continuous upload of new data, requires a precise information which is challenging to acquire. Most of the existing studies evaluate these problems experimentally. There is a lack of formal descriptions, even though they have obtained good results in the analysis of other distributed networks and protocols.

This paper presents a distributed specification in Maude [6], a formal specification language based on rewriting logic, of the behavior of a P2P network that uses the Kademlia DHT. Rewriting logic [13] is 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 for the implementation of real distributed applications. Real-Time Maude [19] 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 strengthens 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) [22], embedded systems [17], and the AER/NCA active network protocol [16]. Moreover, analysis of real-time systems using Maude sockets, and thus requiring a special treatment for them, has been studied [1,26]. While the algebraic representation of the distribution used in these works follows, as well as our work, the approach presented in [22], the way used to relate logical and physical time allows for a more precise and formal analysis than the one used here, allowing the system to synchronize only when needed.

Our distributed specification of the Kademlia protocol has been implemented on top of the routing protocol described in [22] and uses an external Java clock. Since we formally specify the semantics of the protocol, we obtain a correct *by design* application. Moreover, this distributed system can be simulated and analyzed in Maude if a "centralized" version is provided. This version is obtained by using: (i) an algebraic specification of the sockets provided by Maude; (ii) an abstraction of the underlying routing protocol, which allows the analysis tools to focus on the properties; and (iii) Real-time Maude, as explained above. 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. Actually, we present different levels of abstraction, which allows us to focus on the properties we want to prove while discarding the unnecessary implementation details.

Our specification is, to the best of our knowledge, the first formal description of a Kademlia DHT. The use of formal methods to describe the behavior of the Kademlia DHT may help to understand the informal description of the protocol and the algorithm given in [11], and to identify areas that are not covered in the description and are being resolved in different ways in different implementations. In particular, the Maude language gives us the opportunity of executing the distributed specification taking into account the time aspects of the protocol in order to detect weak points in the protocol that would be interesting to study. Then using the centralized model that mirrors the distributed one, we can analyze all possible executions of the system, either by searching in the execution tree (which is in fact represented as a graph for efficiency reasons) or by using model checking techniques.

The rest of the paper is structured as follows: Sect. 2 presents the Kademlia protocol and how to specify generic distributed systems in Maude, as well as some related work. 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 Sect. 5 describes how to simulate and analyze it. Finally, Sect. 6 concludes and presents some future work.

2 Preliminaries and Related Work

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

2.1 Maude

In Maude [6] 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. We can distinguish between Core Maude [6, PartI], which is implemented in C++ and provides the basic features, and Full Maude [6, PartII], an extension of Maude implemented in Maude itself and used as basis for further extensions, as we explain below.

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

In object-oriented specifications, *classes* are declared with the syntax `class` $C \mid a_1{:}S_1, \ldots, a_n : S_n$, where C is the class name, a_i is an attribute identifier, and S_i is the sort of the values this attribute can have, for $1 \leq i \leq n$. An *object* is represented as a term `<` O `:` C `|` a_1 `:` v_1`,` $\ldots$`,` a_n `:` v_n `>` where O is the object's name, belonging to a set `Oid` of object identifiers, and the v_i's are the current values of its attributes. *Messages* are defined by the user for each application (introduced with syntax `msg`).

In Maude, the state of a concurrent, object-oriented system is called a *configuration*. It 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 [6] 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 [22], 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!` 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.

[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.

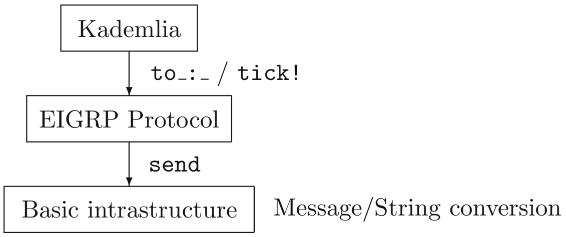

Fig. 1. Layers for distributed applications

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 identifier *close* to the file identifier, where the notion of distance between two identifiers 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, where n is the number of nodes in the network.

In Kademlia, every node keeps the following contact information: IP address, UDP port, and node identifier, for nodes of distance between 2^i and 2^{i+1} from itself, for $i = 0, \ldots, n$ and n the identifier length. In the Kademlia paper [11] 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 identifier. 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 node is discarded. This policy gives preference to old contacts, since the longer a node has been up, the more likely it is to remain up another hour and also prevents attacks by preferring long-standing nodes.

k-buckets are organized in a binary tree called the routing table. Each k-bucket is identified by the common prefix of the identifiers 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 identifier space, and together the k-buckets cover the entire identifier space with no overlap. Figure 2 shows a routing table for node 00000000 and a k-bucket of length 5. Identifiers have 8 bits.

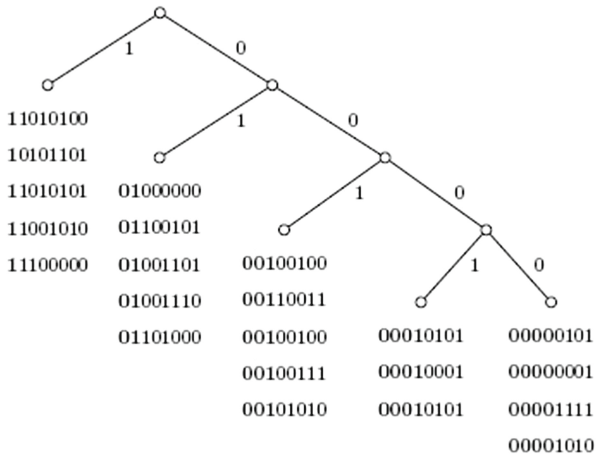

Fig. 2. A routing table example for node 00000000

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 identifier together with the contact of the node that shares the file to publish it to other nodes.
- `FIND-NODE` takes an identifier as argument and the recipient returns the contacts of the k nodes it knows that are closest to the target identifier.
- `FIND-VALUE` takes an identifier 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 dynamics of looking for a value and publishing a shared file from the Kademlia paper [11].

Looking for a Value. To find a file identifier, a node starts by performing a lookup to find the k nodes with the closest identifiers to the file identifier. First, the node sends a `FIND-VALUE` RPC to the α nodes it knows with an identifier closer to the file identifier, 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 not considered. 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. Those nodes that have information about the whereabouts of a file publish it more frequently than the node sharing 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.

2.3 Related Work

One of the first proposals about using formal methods for analyzing the DHT behavior is due to Borgströn et al., who prove in [4] the correctness of the lookup operation of the DHT-based DKS system, developed in the context of the EU-project [9], for a static model of the network using value-passing CCS. Besides, Bakhshi and Gurov give in [3] a formal verification of Chord's stabilization algorithm using the π-calculus. Lately Lu, Merz, and Weidenbach [10,12] have modeled Pastry's core routing algorithms in the specification language TLA$^+$. The model has been validated using the TLC model checker and they have proved the *CorrectDelivery* safety property stating that there can be only one node responsible for any key at any time using the interactive theorem prover TLAPS of TLA. A different approach is used by P. Zave in [27] to analyze correctness of the Chord DHT protocol. She uses the Alloy language and checks properties with the Alloy analyzer. Properties are expressed as invariants of the system and proved by an exhaustive enumeration of instances over a bounded domain. The analysis revealed some flaws in the original description of the algorithm, which allowed the author to propose some improvements.

Regarding the Kademlia DHT there is a previous work of the first author [20] focused on the Kademlia and the Kad routing tables. The paper highlights the main differences between the Kademlia proposal [11] and its first real implementation in the eMule network. Both routing tables were specified in the Maude formal specification language. The network specification presented in this paper uses the Kademlia routing table. The specification is designed in a modular way to support other *Kademlia style* routing tables, like the one from Kad. This will allow us to compare the behavior of different systems only changing the routing table specification.

3 Protocol Specification

We present in this section the main details of the distributed implementation of the Kademlia protocol. The Kademlia network is modeled as a Maude configuration of objects and messages, where the objects represent the network peers and the messages represent the protocol RPCs.

3.1 Peers

Peers in our specification are objects of class `Peer`, defined as follows:

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

which indicates that the class `Peer` has the attributes `RT`, of sort `RoutingTable`; `Files`, of sort `TFileTable`; `Publish`, of sort `TPublishFile`; `SearchFiles`, of sort `TSearchFile`; and `SearchList`, of sort `TemporaryList` These attributes are defined as follows:

– `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 identifier, 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 identifier, the file 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 identifier 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 identifier, it stores them in this file, and starts sending them messages.

Following is an example of an object of class `Peer`. We identify the peers by 6-bit quantities, represented by its decimal value to improve readability in the examples presented in this paper. This size provides us with enough nodes for our example network. However the specification may use any n-bit quantity or the complete Kademlia contact information, since it is parameterized with respect to the peers identification.

```
 < peer(c(48)): Peer |
   RT : (empty-bucket ! c(14)! c(0)! c(16))!! (empty-bucket ! c(33))!!
        (empty-bucket ! c(60)! c(58)! c(56))!!
        (empty-bucket ! c(50)) +
      c(8)+ PING(c(48),c(14),5,1)+ c(60)c(50) + 4,
   Files : < 32 & c(48);; 19 > # < 38 & c(0);; 4 > # < 54 & c(0);; 8 >,
   Publish : < 22 &"File4"@ 25 > # < 32 &"File5"@ 26 >,
   SearchFiles : < 12 &"File7"; 1 >,
   SearchList : temp-empty >
```

The routing table has four buckets, the first one with the contacts which have its first bit set to `0` (values between 0 and 31), the second with the contacts with its first two bits set to `10` (values between 32 and 47), and so on. The bucket dimension, which is also a parameter of the specification, is set to 3 in our example. We observe that the first and the third buckets are full. In the snapshot shown in the example, the peer has had knowledge of a new contact c(8), but since it should be located in the first bucket, which is full, the peer has sent a PING message to the first contact in the bucket, to verify whether it is still alive. The peer will wait four more time units for the reply before it decides that the contact is not alive and removes the PING message. Meanwhile it has had knowledge of two more contacts c(60) and c(50), which are waiting to be processed. See [20] for a detailed explanation of the routing table specification.

The peer keeps information about three files which has been published by other peers in the Files attribute. These are file 32, which has been published by peer c(48), and files 38 and 54, published by the peer c(0). These files are kept in this peer because it is one of the closest nodes to the file identification. We can choose the redundancy parameter in our specification, in the example it is set to three. The time parameter that appears at the end of each file represents the time remaining for republishing the files to keep them up to date. Each time a peer receives a STORE message for a node that it is already keeping it, updates this time value to the time chosen in the specification for republishing. In this way Kademlia prevents all the redundant peers that keep a file from republishing it at the same time.

The Publish attribute presents two files upload to the network by the peer. The first parameter is the file identifier, next is the file name, and the last parameter is the time remaining for republishing the files to keep them alive.

We observe in the SearchFiles attribute the identifier of a file that the peer wants to search for. Again the last parameter is the time remaining for the process to take place. The searched files are removed from the list when the search process succeed, if it fails the search process is repeated after some time.

Finally, the searchList attribute is a list of contacts used in the publish and look-for processes to find the closest nodes to the file identification. In our example, it is empty since the peer is not performing any of these tasks.

3.2 RPCs

There is a Maude message for each RPC defined in the Kademlia protocol. For example, the FIND-VALUE RPC and its two possible replys are defined as follows:

```
op FIND-VALUE : MyContact BitString -> TravelingContents [ctor] .
op FIND-VALUE-REPLY1 : MyContact BitString
        Set{vCONTACT}{vContact-BitString} -> TravelingContents [ctor] .
op FIND-VALUE-REPLY2 : MyContact BitString MyContact [ctor] .
```

Note that terms of this form will be used to form messages with the operator to_:_ described in Sect. 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 set of the k nodes the peer knows that are the closest ones to the target, where k is the bucket dimension or the contact of the owner of the file.

For example, a message from node `c(14)` to node `c(48)` requesting for information about file `54` has the form:

```
to peer(c(48)) : FIND-VALUE(c(14),54) .
```

this message will be in the Maude configuration of nodes and messages.

The reply message sended by node `c(48)` to node `c(14)` if the node does not have information about file `54` in its `Files` attribute list, which is not the case in our example, will be:

```
to peer(c(14)) : FIND-VALUE-REPLY1(c(48),54,c(56),c(58),c(50)) .
```

since the closest nodes to file `54` in the routing table of node `c(48)` are the nodes `c(56)`, `c(58)` and `c(50)`. The order in which these contacts are returned is not important, since they will be ordered by their distance to the given file in the node that ask for them.

Since node `c(48)` is one of the closest nodes in the network to file `54` in our example, and it has this file in its `File` attribute list, the message that it will return contains the contact of the owner of the file, which is node `c(0)`.

```
to peer(c(14)) : FIND-VALUE-REPLY2(c(48),54,c(0)) .
```

3.3 Process Specification in Maude

The specification of the different processes follows their definition. For example, the searching process starts automatically when there are identifiers 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 number of closest nodes used to initialize the auxiliary list is a parameter of the specification. The original Kademlia paper [11] indicates that *it is a system wide concurrency parameter, such as 3.* Notice that in the implementation the file is ordered by the distance of the contact to the file identification. In our example, when node `c(48)` starts searching file `12` we have:

```
 < peer(c(48)): Peer |
   RT : (empty-bucket ! c(14)! c(0)! c(16)) !! (empty-bucket ! c(33)) !!
        (empty-bucket ! c(60)! c(58)! c(56))!! (empty-bucket ! c(50)) +
      c(8)+ PING(c(48),c(14),5,1)+ c(60)c(50) + 4,
   Files : < 32 & c(48) ;;19 > # < 38 & c(0) ;;4 > # < 54 & c(0) ;;8 >,
   Publish : < 22 &"File4"@ 25 > # < 32 &"File5"@ 26 >,
   SearchFiles : < 12 &"File7"; INF >,
   SearchList : < c(14),2,20,0 > < c(16),4,20,0 > < c(0),12,20,0 >
```

where the first value of the nodes in the `SearchList` is the contact, the second value is the distance from the contact to the searched file, the third is the time that the node will remain in the list if no response is received from a sended

RPC, and the fourth value is a flag that indicates if the contact has already sent the FIND-VALUE RPC, has received the response, or has sent a STORE message and is waiting a response.

The searching process continues by sending FIND-VALUE RPCs to the first nodes in the list to find *closer* nodes to the file identifier. The process is controlled by the rewrite rule:

```
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 .
```

which states that 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. Once the RPC is sent, a flag is activated in the search list that marks this node as *in process* with set-flag.

Following our example, when peer c(48) sends the first RPCs, the configuration will have among other peers and messages the following:

```
 to peer(c(14)) : FIND-VALUE(c(48),12)
 to peer(c(16)) : FIND-VALUE(c(48),12)
 to peer(c(0)) : FIND-VALUE(c(48),12)
  < peer(c(48)) : Peer |
    RT :(empty-bucket ! c(14) ! c(0) ! c(16)) !! (empty-bucket ! c(33))!!
        (empty-bucket ! c(60) ! c(58) ! c(56)) !!
        (empty-bucket ! c(50)) +
       c(8) + PING(c(48), c(14),5,1) + c(60) c(50) + 4,
    Files : < 32 & c(48);; 19 > # < 38 & c(0);; 4 > # < 54 & c(0);; 8 >,
    Publish : < 22 &"File4"@ 25 > # < 32 &"File5"@ 26 >,
    SearchFiles : < 12 &"File7" ; INF >,
    SearchList : < c(14), 2, 20, 1 > < c(16), 4, 20, 1 >
                 < c(0), 12, 20, 1 >
```

The receivers of the FIND-VALUE messages may find the file the searcher is looking for in its 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 identifier 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 is a *ghost* message that remains in the configuration to show the files that have been searched and found, hence easing the description of some properties, that just check whether this message appears in the configuration. In the second case, the receiver sends a FIND-VALUE-REPLY1 message to the searcher including the closest nodes to the file identifier 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 identifier 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. The searching process continues by sending new `FIND-VALUE` messages to the new nodes in the `SearchList` that have not been asked yet, and are closer to the searched file identifier than the nodes that have already answer the RPC. If the process does not find the file in any of the contacted nodes, it does not remove the file from the `SearchFile` table and initializes its time for a new search.

4 Centralized Simulation

We use Real-Time Maude [19] to analyze our system. Real-Time Maude is an extension of Maude that allows to perform time-bound analyses such as breadth-first search or model checking. However, Real-Time Maude does not support distributed applications, so in order to use it we need to “centralize” our configuration. We discuss below how to achieve this and how to improve (or *abstract*) this representation. In this case, we distinguish between “architecture abstractions,” which simplify the state by removing the transitions not related to the properties we want to verify, and “formal abstractions,” which refers to established techniques that allow to improve the proofs by different means. It is worth noting that this transformation does not introduce an important overhead on the complexity of the specification: while the distributed implementation of the protocol has around 3100 lines of code, the centralized one has 3300 lines of code, approximately.

4.1 First Architecture Abstraction

As explained above, 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. This centralized specification must fulfill the following requirements:

– The underlying architecture must be simulated. This simulation includes not only redirecting the messages, but also possible delays and errors.
– Nodes can connect and disconnect during the process.

In order to solve the first issue, we provide a class `Process` with a single attribute `conf` that keeps the configurations in different locations[2] separated from each other:

```
class Process | conf : Configuration .
```

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

Besides "separating" the processes, we must 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 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 : O, listA : dl(to O' : TC, 0) DML >
   < O : Process | conf : CONF >
=> < S : Socket | listA : DML >
   < O : Process | conf : (to O' : TC CONF) > .
```

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.

4.2 Second Architecture Abstraction

Note that the abstraction in the previous section provides an exact correspondence between the distributed system and the centralized one. However, it is possible that most of the properties are either independent of the underlying architecture, independent of the disconnections from the peer, or both. For this reason, we can define more refined abstractions that omit some of these aspects. To abstract the architecture we just use a multiset of peers and messages, so messages sent by a peer reach the addressee immediately; to abstract the connections and disconnections we just remove the `Life` and `Reconnect` attributes introduced above and the associated rules, hence preventing the nodes from unwanted disconnections. In this way we obtain two main advantages: (i) the analysis is optimized, since the number of reachable states is greatly reduced; and (ii) it is easier to understand the results and trace back the causes.

4.3 Formal Abstractions

Beyond simplifying the state with the abstractions above, we can also apply other techniques to improve our proofs. The state space reduction technique in [8] allows us to turn rules (which generates transitions and hence new states during the search and model checking processes) into equations given they fulfill some properties: the specification thus obtained is still a correct executable Maude specification (that is, it is terminating, confluent, and coherent; see [6] for details) and the property is *invisible* for the rules transformed into equations. This invisibility concept informally requires the rules to preserve the satisfiability of the atomic predicates involved in the formulas being proved, and is also the basis for our own abstractions. Another interesting way of reducing the state space can be found in [18].

Regarding infinite systems, an important abstraction can be found in [14]. This abstraction turns an infinite-state system into a finite one by collapsing states by means of equations. This kind of abstraction was not necessary in our case, since our system becomes finite by setting a bound in the execution time.

5 Analysis

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 Maude 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.

Note that, before starting the analysis, we need to relate "real-time," as defined by our external Java clock in the distributed specification, and the "real-time" defined by Real-Time Maude. Our distributed specification contains a number of timeouts, defined by natural numbers, and we ask to the Java server to wait this number of seconds. We just mimic this strategy in the centralized specification, using natural numbers (or a constant `INF` standing for infinite time) and we ask Real-Time Maude to wait the maximal possible amount.

We have verified our system with networks form 6 to 20 nodes. We abstract the concrete connections and assume 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. The analysis of networks with hundreds of nodes using a model checker requires the use of some of the abstraction techniques explained in Sect. 4.3 and it is left as future work.

We can simulate how different attacks may affect a network. For example, in the *node insertion* attack, an attacking peer intercepts a search requests for a file, which are answered with bogus information [15]. The attacking peer creates its own identifier 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 identifier, with the following command:

```
Maude> (find earliest init =>* {< O : Process | conf :
                     (to O' : 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 identifier is the closest to the file identifier.

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 {< O : Process | conf : (< O' : Peer | Publish :
  < I1 & (S1 @ TC4) > # PF > CONF) > CONF'} |= PublishAFile(I1) = true .

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

  op FindAFile : MyContact Nat -> Prop [ctor] .
  eq {< O : 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 `54`, that is searched by `peer(c(33))`, we can use the following command to check that the property holds:

```
Maude> (mc init' |=t PublishAFile(54) /\ SearchAFile(c(33),54) =>
                    <> FindAFile(c(33),54) 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,c(0))) U PeerOffline(c(0))
           in time < 40 .)
Result ModelCheckResult : counterexample(...)
```

In a network where `peer(c(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,c(0)) \/ PeerOffline(c(0)))
           in time < 40 .)
Result Bool : true
```

6 Conclusion 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 takes care of the time. 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 implementation of the protocol. This centralized implementation of the protocol just mapped real-time to natural numbers. Although this “time sampling” is usual, the relation between physical time and logic time can be refined further. For example, the paper [1], which describes a theory for the orchestration of service-oriented solutions, or [26], which presents a theory for medical devices, provide a much more refined relation, taking into account small deviations due to hardware.

As future work we plan to use the narrowing techniques implemented in Maude [7] to analyze the Kademlia DHT protocol. In this way, we could apply the analyses described in recent works (see e.g. [2,23]) to our system and check whether an error state is reachable from a generic state (a state with variables).

Another line of research is to compare the performance of the routing tables under different parameters, like the bucket dimension or the concurrency parameters used in the node lookup procedure. The comparison can also be done for the different variants of the routing tables taking advantage of the fact that the specification is parametric on the routing table. We also plan to study more complex properties that could apply under other scenarios.

References

1. AlTurki, M., Meseguer, J.: Executable rewriting logic semantics of Orc and formal analysis of Orc programs. J. Logic. Algebraic Meth. Program. **84**(4), 505–533 (2015)
2. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: van Raamsdonk, F. (ed.) 24th International Conference on Rewriting Techniques and Applications, RTA 2013, LIPIcs 21, pp. 81–96. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)

3. Bakhshi, R., Gurov, D.: 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, vol. 181, pp. 35–47. Elsevier (2007)
4. Borgström, J., Nestmann, U., Onana, L., Gurov, D.: Verifying a structured peer-to-peer overlay network: the static case. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 250–265. Springer, Heidelberg (2005)
5. Breitkreuz, H.: The eMule project. http://www.emule-project.net
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual, version 2.6. http://maude.cs.uiuc.edu/maude2-manual
8. Farzan, A., Meseguer, J.: State space reduction of rewrite theories using invisible transitions. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 142–157. Springer, Heidelberg (2006)
9. Haridi, S.: EU-project PEPITO `IST-2001-33234`. Project funded by EU IST FET Global Computing (GC) (2002). http://www.sics.se/pepito/
10. Lu, T.: Formal Verification of the Pastry Protocol. Doctoral dissertation, Universität des Saarlandes, December 2013
11. Maymounkov, P., Mazières, D.: Kademlia: a peer-to-peer information system based on the XOR metric. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 53–65. Springer, Heidelberg (2002)
12. Lu, T., Merz, S., Weidenbach, C.: Towards verification of the pastry protocol using TLA+. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 244–258. Springer, Heidelberg (2011)
13. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theo. Comput. Sci. **96**(1), 73–155 (1992)
14. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theo. Comput. Sci. **403**(23), 239–264 (2008)
15. Mysicka, D.: eMule attacks and measurements. Master's thesis, Swiss Federal Institute of Technology (ETH) Zurich (2007)
16. Ölveczky, P., Meseguer, J., Talcott, C.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Form. Meth. Syst. Des. **29**, 253–293 (2006)
17. Ölveczky, P.C.: Formal model engineering for embedded systems using Real-Time Maude. In: Durán, F., Rusu, V., (eds.) Proceedings of the 2nd International Workshop on Algebraic Methods in Model-based Software Engineering, AMMSE 2011, EPTCS, vol. 56, pp. 3–13 (2011)
18. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. In: Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, ENTCS, vol. 176(4), pp. 5–27 (2007)
19. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. High. Ord. Symbolic Comput. **20**, 161–196 (2007)
20. Pita, I., Fernández-Camacho, M.I.: Formal specification of the Kademlia and the Kad routing tables in Maude. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 231–247. Springer, Heidelberg (2013)

21. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: ACM SIGCOMM Computer Communication Review - Proceedings of the 2001 SIGCOMM Conference, vol. 31, pp. 161–172, October 2001
22. Riesco, A., Verdejo, A.: Implementing and analyzing in Maude the enhanced interior gateway routing protocol. In: Roşu, G. (ed.) Proceedings of the 7th International Workshop on Rewriting Logic and its Applications, WRLA 2008. ENTCS, vol. 238(3), pp. 249–266. Elsevier (2009)
23. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. In: Escobar, S. (ed.) WRLA 2014. LNCS, vol. 8663, pp. 247–262. Springer, Heidelberg (2014)
24. Rowstron, A., Druschel, P.: Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
25. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Comput. Commun. Rev. **31**, 149–160 (2001)
26. Sun, M., Meseguer, J.: Distributed real-time emulation of formally-defined patterns for safe medical device control. In: Ölveczky, P.C. (ed.) Proceedings of the 1st International Workshop on Rewriting Techniques for Real-Time Systems, RTRTS 2010, EPTCS, vol. 36, pp. 158–177 (2010)
27. Zave, P.: Using lightweight modeling to understand Chord. SIGCOMM Comput. Commun. Rev. **42**(2), 49–57 (2012)