ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Semester Thesis

Reverse Engineering of eMule

An Analysis of the Implementation of Kademlia in eMule

David Mysicka
dmysicka@student.ethz.ch

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Stefan Schmid and Thomas Locher

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Summer 2006

Abstract

eMule is a popular peer-to-peer (p2p) client which builds upon the sever-based eDonkey2000 (ed2k) platform. In newer versions of eMule, lookups can also be performed over the serverless Kad-network, an implementation of a distributed hash table (DHT) called Kademlia. Since only little is known about the techniques and algorithms used by eMule, this thesis presents an analysis of the implementation of Kademlia in eMule. This includes the structure of the routing table, the connection phase with bootstrapping, searching and publishing in Kademlia, the behavior of the client if network address translation (NAT) or a firewall is used and an analysis of all message types used in Kademlia. The algorithms found are compared to the version described in the Kademlia paper. Additionally, a successful attack is presented, that removes an arbitrary keyword from the Kademlia network, using a modified version of the client.

Contents

1 Introduction

The eDonkey2000 (ed2k) network[1] is a popular server-based file sharing platform. As its original client has limited functionality and is not open source, some anonymous developers founded the eMule project[2]. They have started implementing an alternative client for the eDonkey2000 network, which is fully compatible with the original client. Many additional functionality was added to the client, like a credit system for fair downloading or the use of compression. But the most important feature compared to the original client, is the ability of using a second network to exchange information between clients, that does not need any server. This serverless, peer-to-peer (p2p) network is an implementation of a distributed hash table (DHT) called Kademlia[3]. It features fast lookup times, because of a parallel and asynchronous search algorithm. It uses XOR as metric for measuring distances between nodes.
Since only little is known about the techniques and algorithms used by eMule, this thesis presents an analysis of the implementation of Kademlia in eMule. This report begins in Section 2 with the results of the analysis of Kademlia's implementation. It is followed by the presentation of a possible attack of Kademlia, which was found during the analysis. It was successfully implemented and tested by modifying the eMule client. Then, in Section 4, the graphical user interface of eMule is explained using the results of the source code analysis. Appendix A contains a reference of all message types that are used to communicate in the Kademlia network. Appendix B provides a user guide of the modified eMule client that performs the attack presented.

2 eMule's Kademlia Reverse Engineered

This section of the report describes how Kademlia is implemented and used in eMule. It is based on a thorough analysis of the source code (eMule version 0.47a) and the results are confirmed by various tests monitoring the network traffic of eMule. This was done with Ethereal[4], a popular network protocol analyzer, whereas the source code which is written in C++ was examined with Microsofts Visual Studio 2005[5]. It is important to keep in mind that all the Kademlia messages that are sent between two eMule clients use the UDP protocol, whereas the rest of eMule's communication (e.g. with servers) is accomplished with TCP messages.
The section is organized in the following way: After explaining the basic elements in Kademlia in the first three subsections, the routing table is studied. Then, in Subsection 2.5, the connection procedure is examined, followed by a subsection about searching in the Kademlia network. After that, the publishing of information is explained. Then, the behavior of

eMule is analyzed, when it is behind a firewall or when network address translation is used. The section ends with an examination of the built-in estimation function for the number of users in the Kademlia network. Every time a part of the implementation is analyzed in eMule, a comparison is given to the theoretical version of Kademlia.

2.1 Client IDs & File Hashing

Like every other distributed hash table, Kademlia requires a well defined ID space. Given such an ID space, every client can pick some ID and every file can be hashed to a specific ID from that ID space.
eMule uses 128-bit IDs, instead of 160-bit as proposed in the original Kademlia paper. The main reason for this deviation was very likely the existing eDonkey 2000 protocol, which also uses 128-bit IDs to hash the shared files within its network. The idea was to use exactly the same file hashing algorithm in Kademlia as in eDonkey 2000 since eMule uses both networks at the same time. Therefore, the file hashing algorithm used for Kademlia is basically MD4 with some minor modifications.[1]
Prior to the integration of Kademlia into eMule, there already was a system to identify clients. It is used by the eMule credit system, which tries to give download priorities to those clients that had uploaded the largest amounts of data in the past. Therefore, every client randomly generates a 128-bit ID called the *client hash* in the very first session und reuses it in the following sessions to be able to collect credits at other clients, so that it will get a better position in their download queues. To distinguish the eMule client from different clients (e.g. MLDonkey or eDonkey) its 128-bit ID contains two 8-bit areas which have always the same value (byte 5 is always set to `0Eh` and byte 14 to `6Fh`). As a consequence, the client hash cannot be used as the Kademlia client ID, because then the IDs would not be distributed evenly in the 128-bit ID domain anymore. For that reason an additional ID was added. It is called the *Kad ID* and it is used as the ID of a client in Kademlia. The generation process is entirely random. Like the client hash, this ID is generated only in the first session and from then on reused in every following session, so that other clients can find a specific client even after a disconnect.
There exist even more IDs in eMule, but they are of minor importance for the Kademlia system. They all can be examined in the graphical user interface of eMule (see Section 4 for more details).

2.2 Sources and Keywords

In a peer-to-peer system every node is responsible for storing a certain amount of data, so that all nodes together hold the sum of all informa-

[1]Tests showed that the checksums differ from the standard implementation of MD4.

tion. The distributed hash table variant stores only those entries on a node that have the same ID as the node itself or that are near its ID.[2] In Kademlia there are several types of information to be stored at each node. To download an arbitrary file eMule needs to have *sources* for it. Sources are clients that share this specific file. eMule can then contact all the sources it has and request them to upload the file. For this purpose, each node has a map of sources which has the hash-ID of a file as key and a list of sources that share this file as value. This map contains all the files that have an ID near the nodes ID and for every file there are all its sources. It is called `mapSources` in the source code.

But no user can know the hash-ID of the file he or she wants to download, unless he uses a link from a web page.[3] That is why a second type of information needs to be stored, so that user can search for files given their filename or parts of it. Since Kademlia is a distributed hash table, the filename search must also be performed using IDs. One possible solution would be to hash the filename with MD4 and use the result as ID, like it is done with the file hashing. But this would require the user to know the exact filename. Already a change of only one character or a permutation of two words in the filename would lead to a totally different hash result and the user would not find the file. For that reason the file name is split into consecutive words and every word is hashed separately with MD4 to obtain an ID which can be stored at the corresponding node. As a consequence every node has a second map which takes the ID of a word as key and a list of related filenames as value. This means that this map contains all the words that hash to an ID near the nodes ID which is storing it and for every word there are all the filenames that include this word. As these words can be considered as *keywords* this map is named `mapKeyword` in the source code.

eMule allows to store a maximum of 60,000 filename entries in the keyword map. In addition there must not be more than 50,000 entries per keyword ID. The sources map allows to save at most 1,000 sources per file ID. The number of ID entries is not limited.

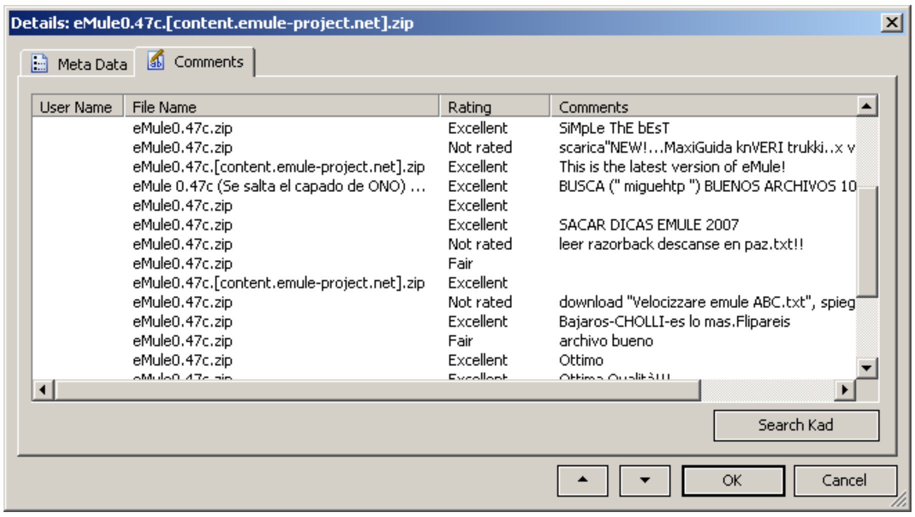
2.3 Notes

The eDonkey file-sharing network is neither controlled nor organized by any authority. Consequently there is no guarantee about the files shared within this network. They can have misleading file names or can be corrupt. Such files are often called *fakes*. One way to prevent users from downloading a fake file, is to use appropriate web pages that have links to valid files, since the author validated or even shares these files. In addition eMule implemented

[2]This is necessary because the ID space is much larger than the number of nodes. So every node must be responsible for a whole range of IDs near it.

[3]The eDonkey protocol supports linking to shared files via URL.

a *notes* system, which allows every user to rate and comment his shared files. These notes are spread via Kademlia, so that every user can review them before he downloads a file.



User Name	File Name	Rating	Comments
	eMule0.47c.zip	Excellent	SiMpLe ThE bEsT
	eMule0.47c.zip	Not rated	scarica"NEW!...MaxiGuida knVERI trukki..x v
	eMule0.47c.[content.emule-project.net].zip	Excellent	This is the latest version of eMule!
	eMule 0.47c (Se salta el capado de ONO) ...	Excellent	BUSCA (" miguehtp ") BUENOS ARCHIVOS 10
	eMule0.47c.zip	Excellent	
	eMule0.47c.zip	Excellent	SACAR DICAS EMULE 2007
	eMule0.47c.zip	Not rated	leer razorback descanse en paz.txt!!
	eMule0.47c.zip	Fair	
	eMule0.47c.[content.emule-project.net].zip	Excellent	
	eMule0.47c.zip	Not rated	download "Velocizzare emule ABC.txt", spieg
	eMule0.47c.zip	Excellent	Bajaros-CHOLLI-es lo mas.Flipareis
	eMule0.47c.zip	Fair	archivo bueno
	eMule0.47c.zip	Excellent	Ottimo

Figure 1: The notes dialog box

A note is bound to a specific file (or more precisely to a file hash ID) which it rates. It holds a comment field that allows the user to enter an arbitrary text and a rating box with the possibility to rate the file with one of six options (no rating, corrupt / fake, poor, fair, good, excellent). Figure 1 shows an example of a rated file in the eMule GUI with several comments, each from a different user. The notes system also helps to reveal misleading file names, because every comment entry also shows the filename that a user has given to this specific file. If the file names completely differ from each other, than this is a good indicator for a misleading file name.
To be able to spread the comments there is a third map on every client, called `mapNotes`, which stores the comments that belong to files whose IDs are close to the nodes ID. This map takes the file ID as key and a list of notes as value. A note consists of the filename, the rating number, the comment text and the name of the user who wrote the note. `mapNotes` has a limitation of 150 notes per file ID, but the number of file IDs and therefore the total number of notes is not limited.

2.4 The Routing Table

This subsection describes the routing table that is used in eMule's implementation of Kademlia for storing information about other nodes in the Kademlia system. It is essential to have a well organized and fast routing table, since every operation uses it to retrieve those nodes that it is going

to contact. Nodes that are stored in the routing table of a specific user are called the *contacts* of this user.
The subsection is divided into several parts. The first explains the structure of the routing table, the second shows how a new contact is added into the structure, followed by an analysis of how the contacts are kept up-to-date. After that, the forth part describes where new contacts come from and finally, the routing table is compared to the one described in the Kademlia paper.

2.4.1 Binary Tree with Zones and Bins

The routing table is implemented as a binary tree, which means that there is a root node on the top of the tree and that every node has two child nodes below it, unless it is a leaf node. The nodes are called *routing zones*. The actual data (namely the contacts) is only stored in the leaf nodes. Therefore, they have a structure named *routing bin*, which can hold up to ten contacts. They are kept in a list, using a least recently used strategy (LRU), which means that contacts are retrieved from the front of the list and inserted at the end. All other nodes of the binary tree (the inner ones) only have a reference to their two sub zones. An example of such a tree is shown in Figure 2.

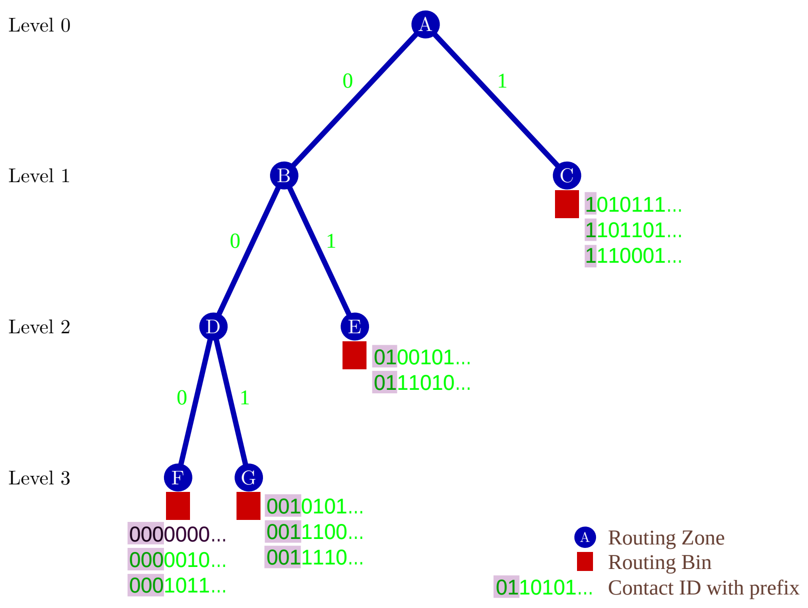


Figure 2: An example of eMule's routing tree with 7 zones and 4 bins.

Furthermore, every routing zone contains a reference to its parent zone, an

integer that states on which `Level` this zone resides (level 0 is the root of the tree, level 1 its two children, etc.) and an integer named `ZoneIndex` which will be explained in the following subsection. Contacts that are stored in routing bins consist of the 128-bit ID, the IP address, the UDP and TCP port and of a `type` and `expire time` (the last two are needed to keep the contacts up-to-date and are discussed in Section 2.4.3 later on).
In which bin a specific contact is placed in, depends on his *distance* to the client, which is defined as the XOR between the contact's ID and the client's ID. Now, every bit in the distance corresponds to a level in the tree, beginning with bit number 0 which corresponds to level 0. In other words the most significant bit corresponds to the root zone. If this bit is 0 the contact belongs to the left sub tree and if it is 1 it belongs to the right sub tree. The second bit (bit number 1) decides in which sub tree the contact belongs to in level 1, and so forth. With this scheme every contact can only be placed in exactly one bin. Remember that the bins are only at the leaves of the tree, so this scheme will give every contact a unique path to its bin. The other way around, a bin contains only contacts from a specific ID space, which is limited by a prefix of the length of the path from the root to that bin. Figure 2 illustrates several contacts and shows in which bin they belong. Also, the prefix of the IDs is highlighted.
Since the routing tree uses relative IDs, the distance to the client would be 0 and therefore it would be placed in the leftmost bin available. Of course, the node with the client's ID is not inserted into its own routing tree, because it is never going to contact itself. As a consequence, the further away a specific bin is from the leftmost bin, the larger is the distance of the contacts in it (regarding their IDs to the client's ID).

2.4.2 Adding a new Contact

If a new contact is going to be inserted into the routing tree, first the appropriate bin has to be selected according to the procedure described above. If the contact already exists in this bin, then the procedure is aborted. If not, then the contact is inserted at the end of the list. But every bin can only hold up to ten contacts. In case of a full bin it is split into two, by creating two new bins, which become the children of the current leaf node in the tree. Then all the contacts from the old bin are moved to the two new bins and the new contact that is being added to the routing tree, can now be inserted into the appropriate bin. The old bin is no longer located on a leaf node and is therefore removed.
But the tree cannot be expanded continuously without a limitation. The two integer values `Level` and `ZoneIndex` are present in every bin for exactly that reason. While the level was explained in the previous subsection, the zone index will be clarified now. It represents the horizontal distance in number of routing zones from the leftmost one on the same level. In the

example tree in Figure 2 nodes A, B, D and F have an index of 0 and nodes C, E and G have an index of 1. Now, the limitation for splitting a bin is defined by the following 3 rules:

- Up to level 3 splitting is always permitted. So, it is possible to have a complete binary tree of depth 4 with 16 bins, all on level 4.
- If the level is greater than 3, the limiting element is the zone index, which allows splitting only if the value is smaller than 5. Thus, in deeper levels only the first five bins on the left side can be split. That implies that there can be a maximum of 10 zones per level, from which the first 5 can be split further and the other 5 cannot be split anymore and are therefore bins.
- If the tree has reached level 127 splitting is not allowed any more.

The result from this limitations is a highly unbalanced tree that can only grow down on the left side. This can be seen in Figure 3, which shows the *fully* populated tree. But this of course is on purpose because it produces a logarithmic distribution of contacts regarding their IDs. The smaller the distance in the tree to ourself is, the more contacts there are. According to the Kademlia paper, only such a routing tree results in logarithmic searches. The routing tree is empty when eMule starts connecting to the Kademlia network. It is populated during the session, starting with only one bin in the root zone of the tree. Then, new contacts are added (they come from different sources, see Section 2.4.4 later on) and the bins are split according to the rules described earlier, until the limitations are reached. From the splitting rules follows that there could be a maximum of 6'310 contacts (0 bins on *levels 0 to 3*, 11 bins on *level 4*, 5 bins on *levels 5 to 126* and 10 bins on *level 127*; totals up to 631 bins and therefore 6'310 contacts). But in practice such a large number is never reached. Most of the time the total amount of contacts varies between 300 and 800.

2.4.3 Keeping Bins up-to-date

To be useful, the contacts in the routing tree have to be up-to-date. Since the nodes in Kademlia have a high rate of leaving and joining the system, the routing tree has to consider this. For that reason, there is a process which searches for new contacts to populate the tree and one that periodically tests if the existing contacts are still alive and deletes dead ones.
The first mentioned process checks once an hour for every bin if it needs new contacts. This is the case if the bin has less than 3 contacts in it or if splitting the bin is allowed. To find new contacts a random ID from the bins ID space is chosen and a search for this ID is started (searching will be explained in Section 2.6). It will find some new contacts that have an ID

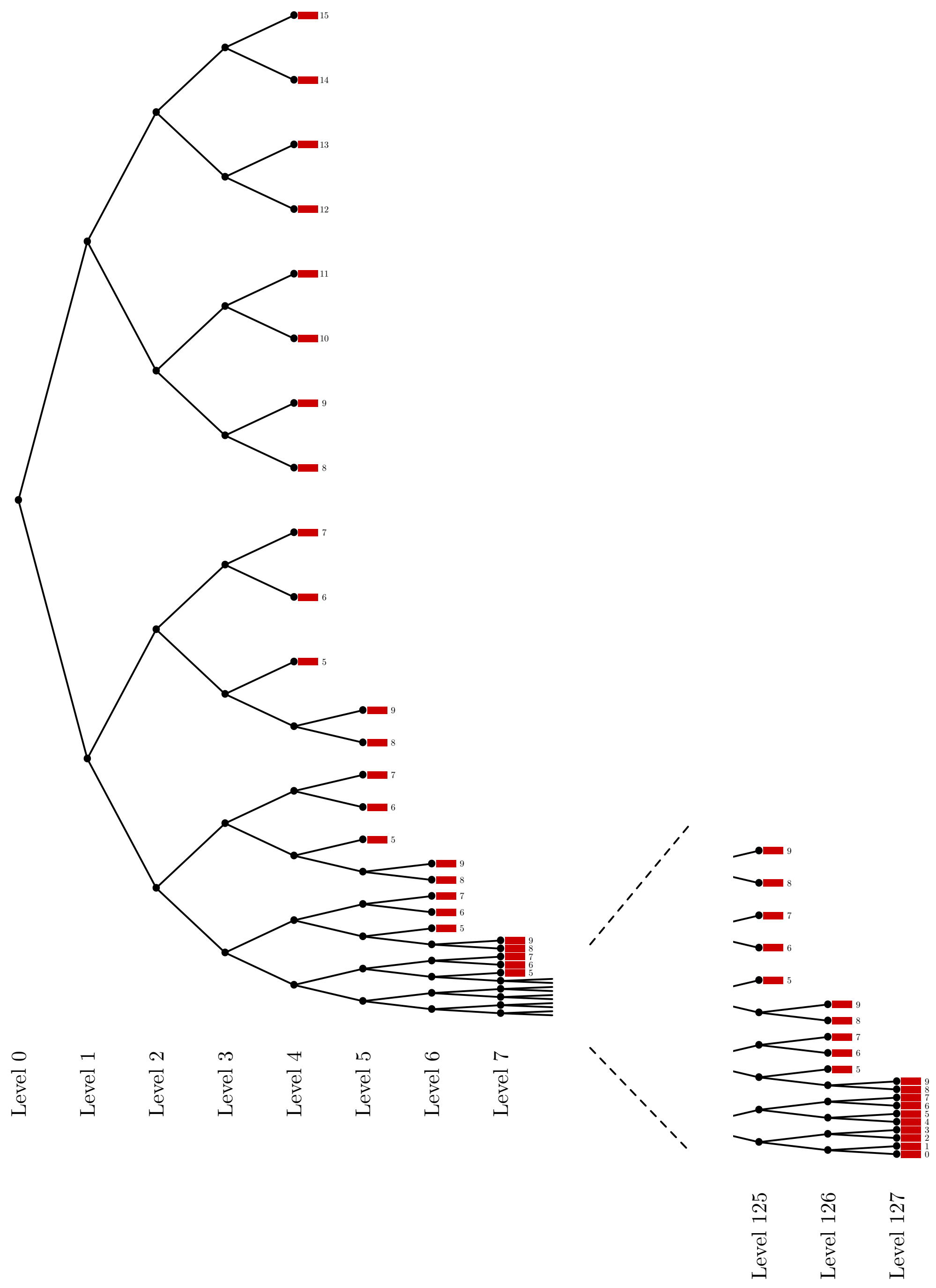


Figure 3: Sketch of a fully populated routing tree (only the top and the bottom of the tree are shown)

near the searched one and they will be inserted into the bin. If a new bin is created this process is immediately started for this bin.
The other process is responsible for deleting dead contacts. For that reason every contact has an `expire time` variable, which holds the time when a specific contact has to be checked again if it is still alive. It also has a variable `type` that indicates in 5 levels how long a contact is already online, where older contacts get a lower value. These older contacts are regarded as better ones because of the fact that the longer a contact has been online, the more likely it is to remain online, as stated in the Kademlia paper. Since the bins apply a least recently used strategy as described in Section 2.4.1, they implement this preference for old contacts. The following list illustrates the 5 different levels of the type variable including their icons, which can be seen in the graphical user interface of eMule (see Section 4 for more details):

- 🟢 **Type 0**: These contacts are known for more then two hours and are rated as the most reliable ones.
- 🔵 **Type 1**: This level is used for contacts that were added between one and two hours ago.
- 🟡 **Type 2**: Contacts with this level are known for less than an hour. They are not rated very reliable, but they have responded at least one time.
- 🟠 **Type 3**: A newly inserted contact obtains this level, until it has responded for the first time. After that it is rated type 2.
- 🔴 **Type 4**: Represents all contacts that failed to respond for at least one time. They can be considered unreachable and will be deleted from the routing tree.

Every bin is scanned by the process once a minute by retrieving the first contact and checking it. If its expire time is ok, the contact is re-inserted into the bin. But if the time has expired, it needs to be tested if it is still alive. This is done by sending a `HELLO_REQ`[4] message to that contact. Moreover its type is increased by one level and its time is set to expire in 2 minutes. This forces the process to check this contact again in 2 minutes unless it has answered by then. Additionally, the type increase leads to a devaluation of its rating. In the case that the contact is alive, it will send back a `HELLO_RES` message, indicating that it is still reachable in the Kademlia system. Receiving this message will place the contact at the end

[4]A list of all message types and their format can be found in Appendix A.

of the list and invoke a procedure that re-adjusts the type of this contact and sets a new expire time. This is done using the following table that lists three different cases that may occur, depending on the duration the contact is known for (uptime):

Uptime	New type level	New expire time
0-1 h	Type 2	1 h
1-2 h	Type 1	1,5 h
$>$ 2 h	Type 0	2 h

This shows that the longer a contact is known, the less it is contacted and checked. However, a newly inserted contact obtains type 3 and is checked immediately, since it might be already dead when it is inserted. As the expire time is set to 2 minutes when a contact is checked and the type is increased, a dead contact will end up in 2 to 8 minutes as type 4. The process will then delete such a contact as soon as its bin is re-checked. As the process is run on every bin every minute, the set of all contacts is processed in 10 minutes (but they are only contacted if their timer has expired). This is because of the fact that there are a maximum of 10 contacts per bin and every time only one contact is processed.
Finally there is a third process, that consolidates almost empty bins. This is only done with two adjacent bins that have the same parent routing zone, which means that they both were created because their parent zones bin was full and had to be split. The consolidation is done if they both together have less than 5 contacts. It is the reverse procedure of splitting a bin: A new bin is created in their parents routing zone, the contacts are moved there and the two adjacent bins are deleted. This consolidation process runs only every 45 minutes, but it checks the whole routing tree.

2.4.4 Where do new Contacts come from?

An interesting question is to analyze where do the contacts come from or, in other words, what has to happen so that a contact is added to the routing tree.
One part is the startup of eMule. Since Kademlia cannot work without any contacts and does not work well with only a few of them, they have to be present from the beginning. Therefore, they are permanently saved to a file, when eMule is closed and re-loaded from that file at startup. More details on that procedure and especially which contacts are chosen to be stored to disk will be explained in Section 2.5, later on. It will also discuss a bootstrapping procedure, which is used at the beginning, if there are only a few contacts. This procedure is also a source for contacts as it explicitly orders some new ones from a known contact and inserts them into the routing tree.

There are only two cases where contacts are inserted into the routing tree, without eMule having ordered them explicitly. The first one occurs when another node needs to check whether the client is still alive, so that it can decide if it keeps this client in its routing tree. As explained before the node will send the client a `HELLO_REQ` message, which will then reply with a `HELLO_RES` answer. But the client will also insert that node as a new contact into its routing tree. The second case arises if some other node is bootstrapping and asks the client for some contacts. It does this with a `BOOTSTRAP_REQ` message and the client answers with a `BOOTSTRAP_RES` message that includes several contacts it knows. But also in this case the client will insert that node as a new contact into its routing tree. Apart from this two cases, *no* other type of message such as a search request or a publish request (will be explained later) from another node will induce the client to insert that node into its routing tree. That implies, that attacks aiming to force an insertion of contacts into nodes have to use this two cases.

2.4.5 Comparison with Paper & Discussion

There are some minor differences when comparing eMule's implementation of the routing tree with the theoretical one described in the Kademlia paper. The latter has a limitation of only 160 bins (which they call buckets) and uses a tighter splitting rule, where it is only allowed to split a bin if its ID space includes the user's ID (which would be 0 in eMule since it uses a relative representation). Although they present an optimized version that allows more bins to be split (namely the two neighbors of the bin with the user's ID space), eMule's version allows even more bins to be split, which results in more possible contacts.
The theoretical algorithm is specified such that it will add the sender from every received message to the routing tree. eMule does this only with messages that check if a node is alive and that are for bootstrapping purposes. The reason for adding contacts from every message (especially the search messages) lies in the symmetric structure of Kademlia as stated in the paper. Besides, eMule does not implement a replacement cache which would hold new contacts that could not be inserted because of a full and unsplittable bin and would later on replace dead contacts in that bin. Instead, there are more bins, so new contacts can be inserted over a longer time. This solution is a little worse, because a replacement cache would lead to less contacts but there would be a bigger preference to older and therefore better contacts.
Both systems, the theoretical and the implemented one use almost the same algorithms for keeping the bins up-to-date. There are only differences in detecting dead contacts. The paper does not propose a timer which indicates when a contact should be re-checked as eMule does. They check for dead contacts only when inserting into full and unsplittable bins. Therefore, the probability of having useless contacts in the tree is lower in eMule's

implementation, since it re-checks every contact at least every 2 hours.

2.5 Connecting & Bootstrapping

As mentioned before, it is very important to have at least a few working contacts to be able to interact with the Kademlia network. Consequently they are also needed at the beginning of each session. eMule stores some of them to disk when it is closed to be able to provide them at the next startup. This subsection explains which contacts are stored to disk, how the startup procedure works and how eMule performs bootstrapping when no contacts are available. At last, the results are compared with the theoretical Kademlia version.

2.5.1 Storing Contacts

When eMule is closed a maximum of 200 contacts is saved to the file called `notes.dat`. The objective is to select 200 from all contacts, so that they are distributed over the whole ID space. This will help to quickly populate the routing tree at the next startup. The algorithm therefore works as follows: Starting in the root zone of the tree, it recursively first visits the left child zone and then the right one. If at any time a bin is reached in a zone (which means that a leaf is reached in the tree), the containing contacts are appended to the file, until the maximum of 200 is reached. But from level 5 on, the algorithm does not visit both child zones anymore, instead it randomly chooses either the left or the right one. This is done to achieve the distribution over the whole ID space, since without that only the contacts from the 20 leftmost bins would be stored.
As explained in Section 2.4.1, the right part of the routing tree is limited to level 4. This part can contain up to 10 bins. In the other part the algorithm chooses from level 5 on only one child zone, which stops the recursive multiplication there. So, it would be the same thing as if this part of the tree ended on level 5 and there were up to 11 bins on that side. This sums up to 21 bins, that are visited at the most. If they are all full, only the leftmost 20 will be visited, because of the recursive behavior which first goes down to the left child zone and then to the right one. Remember that the tree is a highly unbalanced one that only grows down on the left side, and therefore many of the contacts that are in those deep levels on the left side near our ID will not be saved.

2.5.2 Connection Phase

As soon as the *Connect* button is pressed in eMule, all contacts that were saved when it was closed last time, will be loaded into the routing tree. This will create the same 20 bins that the algorithm visited while storing the contacts at the end of the last session. Then, they will be checked by the

two processes which keep the bins up-to-date as described in Section 2.4.3. The process that searches for new contacts will generate a *random lookup* in all bins that are splittable. This applies only to the 10 bins on the right part of the tree, whereas the other process which deletes dead contacts will check all contacts in every bin. Thus, every minute a contact will be checked in every bin, so that after 10 minutes all contacts have been tested and either deleted if they did not respond or kept if they did respond. Actually it is somewhat more complicated, because if a contact does not respond, it will not be deleted until the next time the process scans its bin. Consequently, this will prolong the whole process of checking the contacts in a bin for one minute and if all contacts were dead, then it would take 20 minutes to check them.
Depending on how much time passed since eMule was last used, more or less contacts will be still alive. Many users do not use eMule regularly and the majority of all users does not have a fixed IP address either. Therefore, many of them will most likely not be reachable if we reconnect and try to contact them, as their addresses have changed. That is why so many contacts are stored for the next session, so that with high probability there will be some of them that are still alive and did not change their address. The case that all of them are unreachable will be discussed in the next subsection.
The range near our ID, or in other words, the bins in deep levels on the left side of the routing tree will be very weakly populated after the insertion of the saved contacts, as could be seen before when looking at the appropriate algorithm. But this range should also be populated from the beginning, because contacts near our ID are needed when other clients search for them and request us to send them these contacts. This is part of the searching procedure and will be clarified in Section 2.6. To populate that range a *self lookup* is initiated right after three minutes of being connected. It is a search for the own ID, which will look for contacts that have an ID near ours. The self lookup is then repeated every four hours.
Through the self lookup and mainly through the process that searches for new contacts in every bin, the routing tree will be filled more and more, until a balance is reached. This has to do with the fact that contacts leave the system and are removed from the tree and, on the other hand, new contacts join it and are added to the tree.

2.5.3 Bootstrapping

There are two cases were eMule does not have any usable contact and connecting to the Kademlia network is a little more difficult: Either the last session was a long time ago and all saved contacts are useless, because none of them is alive or if eMule is installed und started for the first time. In both cases a bootstrapping procedure has to be performed. It is sufficient to obtain one working contact, which will provide several contacts from its own

routing tree. These can then provide their contacts and so on. This is done by sending a `BOOTSTRAP_REQ` message to that first contact, which requests it to send 20 contacts from its routing tree. It retrieves those contacts with the same procedure that is used when saving them to disk at the end of a session (which was described in Section 2.5.1). After that it replies with a `BOOTSTRAP_RES` message which contains these contacts. We add them to our routing tree and the connection phase continues as if it would be a regular one where we have loaded the contacts from disk.
But it has to be noted that if only 20 instead of 200 contacts are retrieved, the algorithm does not deliver a set of contacts whose IDs are well distributed regarding the ID space. On the contrary, they come from only two bins and there IDs are therefore very close together. Moreover, they have IDs that are near to the contact that was used to start the bootstrapping procedure.
Now the question arises where to find such a working contact so that the bootstrapping procedure can start. In eMule there are two possibilities for that. If the user knows another client that is currently connected to the system, he can enter his IP Address and port number in the graphical user interface and push the *Bootstrap* button. But in the most cases the user does not know any IP addresses of other clients. Then he can use the second possibility, which makes use of the fact that almost every eMule client is connected to the Kademlia system *and* to a server via the eDonkey2000 protocol at the same time. eMule then can use the sources that the server provides for those files that are currently being downloaded by the client. It will contact them and request them to send some Kademlia contacts. For that, the client has to be connected to a server and has to have at least one file in the download queue, otherwise no sources will be needed from the server. Once eMule has received a source for a file, it will contact that source through its client-to-client protocol to obtain important information, like which parts of the file it offers to download. If that source answers and there are no contacts in the routing tree, eMule will send a `BOOTSTRAP_REQ` message to it and proceed as described above. Since the servers do have fixed IP addresses and there are lists of popular servers, it is possible to connect to Kademlia without having any contacts, by using the procedure described here.

2.5.4 Comparison with Paper & Discussion

The connection phase of eMule resembles to the one described in the Kademlia paper. The only difference is that eMule uses a lot more contacts to begin with, which gives a high probability that one will be still alive. Also, the theoretical version does not contain a bootstrapping procedure, where one contact sends its own contacts to the client which is running the bootstrapping. But an interesting idea was not implemented in eMule. If a new client joins the system, his nearest neighbors (regarding the ID) will notice this,

because the joining client will do a search for nodes with its own ID and find them. The paper recommends the nearest neighbor to transmit all the information he is storing (sources, keywords and notes) to the new client. If this is not done, only the original publishers of that information can send it to the new client, when they republish their information. As we will see in Section 2.7 the republish period is set to several hours, which could cause information to be lost in the system during this time if all the clients that store it leave the system.
There is one flaw that can be pointed out in eMule's implementation concerning the bootstrapping procedure. If a client is requested to reply with some of his contacts, the same algorithm is used as for storing contacts to disk at shutdown, but with a smaller number of them. It seems that the algorithm was not optimized for such a small number of contacts, as it only collects contacts with IDs that are close to each other and near by the client that runs the algorithm. But it would be much wiser to send contacts whose IDs are distributed over the whole ID space to a client that is joining the network and is trying to populate his routing tree.

2.6 Searching

Kademlia's main functionality is to provide information that other clients in the network hold. This is realized through searches. This subsection will discuss their implementations in eMule. First it is necessary to differentiate between several types of searches, which will do the following part. Then the search procedure is analyzed and explained in detail. After that, the type of search that looks for new sources is evaluated in detail in its own part. At the end a comparison to the search procedure in the paper is presented and discussed.

2.6.1 Types of Searches

There are many types of searches, as one can search for contacts, sources, keywords, notes and so on. But no matter which type of search, they all have a *target ID* that determines the point in the ID space where the search will find its results. In Section 2.7 we will see that every information is stored redundantly on several clients whose IDs are located around the ID of the information (which is obtained through hashing it). Therefore, when searching for a specific information the target ID of the search will be set to the ID of that information, unless when searching for a contact. Then of course the target ID will be the contact's ID itself.
To get an overview of the different types of searches, the following list presents them. Their icons and names can be seen in the graphical user interface of eMule (which will be discussed in Section 4).

Node Lookup: This type of search is used to find new contacts, so that the routing tree is populated or refreshed. It is only used by the *self lookup* and the *random lookup* procedure that were explained in Section 2.5.2.

Search Sources: If there are any files in the download queue, this type of search will find sources that can deliver parts of them.

Search Keywords: This type of search is activated when a user starts a search in the graphical user interface. It will find files whose file names match the keywords entered.

Search Notes: If the user is interested in notes for a specific file, he will use this type of search to find comments and ratings from other users for that file. This can be done for keyword search results, files in the download queue and files that the user is sharing.

Store File: This type of search is part of the publishing process that will be explained in Section 2.7 later on. It is used to publish a file shared by the user. Therefore, the ID of that file and the users IP address will be published.

Store Keyword: To publish the filename of a file the user is sharing, this type of search is used. This is done for every keyword in the filenames of all shared files.

Store Notes: If the user comments or rates a file he is sharing, this type of search will publish the note to the network.

The last three types in the list are used in the publishing process. The reason why they are listed here is because the process of publishing information in Kademlia is the same as when searching for information. The only difference is the request message that is sent to the nodes, which is either a store request or a retrieve request. But finding the relevant nodes is always accomplished the same way.
Another differentiation that can be observed in the list is that there are types of searches which the user has to activate (search keywords and search notes) and others which are started automatically through eMule's internal processes.

2.6.2 Detailed Search Procedure

Whenever a search is initiated, a corresponding object is created and inserted into the map called `mapSearches`, which contains all active searches at the current time. It has the search target ID as key and the search object as value. A *search manager* controls all active searches, by cycling through them and checking if something has to be done. If necessary he sends messages or processes responses. An interesting feature of eMule's graphical user interface is the visualization of the active searches, because it shows a lot of information and is very helpful when trying to understand how the searches work (cf. Section 4).

Every search is executed using two parallel processes. The first one searches for new nodes that are near the target ID of the search. Although there are already many contacts in the routing tree, the chance that it will contain the ones needed is very small, since there is an enormous number of possible IDs in a 128-bit range. The second process requests these nodes for the information that the search is looking for. As mentioned before, these two processes work in parallel, to optimize the search speed. As soon as the first node is found by the first process, the second process can already ask that node. The two processes will now be explained in more detail, beginning with the first one.

To find the nodes that are needed, the first process has to ask already available contacts for them. For that reason, it will look in the routing tree for contacts that are as near as possible to the target ID.[5] eMule retrieves the 50 nearest contacts from the routing tree and places them into a temporary map, which has the distance from a specific contact to the target ID as key, allowing it to retrieve the nearest one at any time. To obtain these 50 contacts, first all of the 10 contacts from the bin that is closest to the target ID are retrieved. Then, the second nearest bin is visited by going one level up to the parent routing zone and taking the other path down. This is repeated recursively until 50 contacts have been found. Contacts with a type of level 4 will not be considered. Figure 4 demonstrates the procedure on an example routing tree.

Now, the process begins to ask these 50 contacts, to obtain nodes from them that are closer to the target. eMule contacts 3 nodes at the time, which is similar to the implementation in the paper, that uses a "system-wide concurrency parameter α" and proposes to set it to 3. To contact a node a `KADEMLIA_REQ` message is sent containing the target ID and the number of nodes requested. The latter depends on the type of the search. For a node lookup 11 nodes are requested, for a source, keywords or notes search 2 are requested and for a file, keyword or notes store 4 are requested. If a node receives this message, it will retrieve the requested number of

[5]Remember that always the nearest ones have to be asked first to obtain a search that is logarithmic in time.

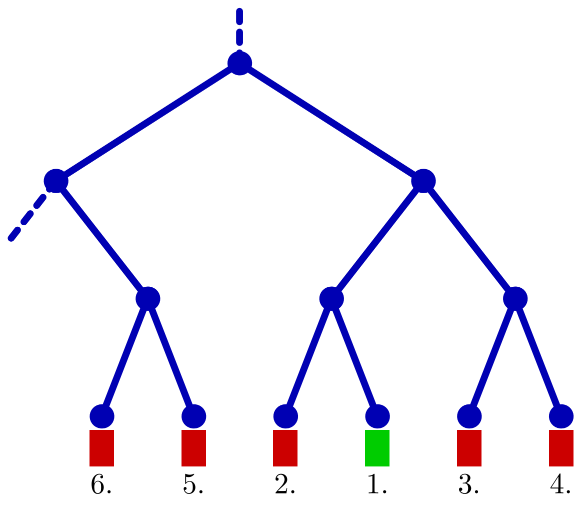


Figure 4: Bin visiting sequence of the procedure for retrieving the 50 closest nodes. The bin whose ID space matches the target ID is highlighted green.

contacts that are closest to the target ID from its routing tree. Thereby the same method is used as if contacts were retrieved to start a new search, as described above. Then, the requested node replies with a `KADEMLIA_RES` message including these contacts. When eMule receives this response, it adds all the included nodes as new contacts into the routing tree. They are also added into the temporary map of the current search so that they can be asked if needed. Then again, the 3 currently nearest nodes are contacted in parallel to obtain even closer nodes. But they are only contacted, if they are closer to the target ID than the node that provided them. This procedure continues as long as there are new nodes that are closer to the target.
When looking at the first process it becomes obvious that it is a passive one, since it only then asks nodes when another node from an earlier request has responded (except at the beginning where the 3 closest nodes are contacted to initiate the procedure). It can happen that at any time none of the 3 contacted nodes answer. Therefore, another process is needed, which observes every search and restarts it if necessary. This is done by the second process that was mentioned at the beginning of this subsection. It continuously loops through the currently active searches and checks those that did not receive any response for the last three seconds. There are several things that the process has to examine. First it goes through the temporary map, beginning with the node that is closest to the target ID. If there is a node that was not yet requested to send its closest nodes, it will be asked now and the process switches to the next search. If on the other hand the process finds a node that did not respond to the `KADEMLIA_REQ` message, it will be

removed and the process retrieves the next node from the map. As a result all nodes that do not answer within three seconds are removed, because the process has to wait that amount of time until it can check a stalled search. The most important task of the second process however, is to send the actual request of a specific search to nodes that have replied with a `KADEMLIA_RES` message. This may be a request for sources or notes, the results for keywords or one of the three store requests, according to the type of the search. This is done when going through the temporary map, where again the closest node is processed first. The following list describes for every type of search what exactly is sent:

- **Node Lookup**: Nothing has to be sent, because the actual request is to find new contacts and that is already done with the `KADEMLIA_REQ` message.
- **Search Sources**: A `SEARCH_REQ` message is sent containing the file hash ID, for which sources should be found.
- **Search Keywords**: Also here a `SEARCH_REQ` message is sent, but with the hash ID of the first keyword and a search expression if there are more than one keywords. To distinguish the keyword search from the sources search an additional flag is sent, which is either set to 0 or 1.
- **Search Notes**: A `SEARCH_NOTES_REQ` packet is sent that holds the target ID, which is in this case the file hash ID of the file for which notes are being requested.
- **Store File**: To publish that the actual client is a source for a specific file, a `PUBLISH_REQ` message is sent, containing the file hash ID, the file name and details of the client such as its IP address and port number.
- **Store Keyword**: If a keyword is being published, also a `PUBLISH_REQ` message is sent, which includes the keyword's hash ID and a list of the files that have it in their file names. If there are more than 50 file entries, the message is split up to 3 packets, each having 50 entries. Also here a distinction is necessary between a file store and a keyword store. This is done with a flag, too.
- **Store Notes**: To publish a note for a file, a `PUBLISH_NOTES_REQ` message is sent, containing the file hash ID, the file name, the rating and the comment.

When some node receives such a request, it has to process it and possibly send a reply. What has to be done in the case of a store request will be discussed later on in Section 2.7. If it is one of the three search requests, the node tries to lookup the requested information in the corresponding map (sources in `mapSources`, Keywords in `mapKeyword` and notes in `mapNotes`).

Then it will reply with a SEARCH_NOTES_RES message for a notes search or with a SEARCH_RES message for the other two searches, which contains the retrieved results. If the node does not find any matching results, it will *not* send any reply.
While a search is running, it receives more and more results from different nodes. As soon as there are enough results, the search can be stopped to prevent the system from wasting network resources. This task is also done by the second process, while cycling through the active searches. Therefore, every search object has a counter for the received results, which cumulates the items that have been received (not the number of responses from different clients). As the information is stored redundantly in Kademlia, there will be duplicates among all replies. The counter does not consider them. There is also a timer in every search object, that holds the time elapsed since the beginning of the search. It is used to stop searches that where active for a long time and were not stopped by the results counter, because they delivered not enough results. To suit the different search types, the values that stop a search were adapted. The following table shows these values for every type of search:

Search type	Results	Expire time (seconds)
Node Lookup	10	45
Search Sources	300	45
Search Keywords	300	45
Search Notes	50	45
Store File	10	140
Store Keyword	10	140
Store Notes	10	100

As soon as one of the two criteria is fulfilled, the search is stopped and removed from the map that contains all active searches.

2.6.3 Automatic Search for Sources

The type of search that retrieves new sources for the files a user is currently downloading, works completely in background without its notice. It is therefore very interesting to analyze this type of search, especially to find out when and how often it is launched.
As soon as eMule is connected to the Kademlia Network, a search for sources is started for every file that is in the download queue. If a new file is added to the queue, an immediate search is started too. But there is a limitation to 5 parallel searches for sources, so if there are more files in the queue, the other searches have to wait until one finishes. After a successful first search for a specific file, the next searches are repeated in increasing intervals, always waiting 1 hour more, until an interval of 7 hours is reached (i.e.

the intervals are 0, 1, 2, 3, 4, 5, 6, 7, 7, 7, . . . hours). But the searching for sources in Kademlia is omitted from the moment where a file has more then 50 sources. This is a fixed value, that cannot be changed by the user. However this does not necessarily imply, that files only have so few sources, because already the first search can deliver up to 300 sources (as described in the last subsection). Also, there are many other ways to obtain sources. Even if eMule is not connected to a server, sources are exchanged on a client-to-client base.

2.6.4 Comparison with Paper & Discussion

First it has the be mentioned that eMule stores different types of information in the Kademlia system, so there also have to be several types of searches. The paper distinguishes only between a node search and a value search. The procedure of a particular search in eMule uses the same strategy as described in the paper to obtain the logarithmic lookup speed. Nevertheless, there is a major difference in eMule's implementation, since it uses 2 parallel processes for a search, where the first one searches for closer nodes and the second one asks for the requested information. This actually almost *doubles the message cost* of a search, because every node which is involved is asked twice in contrast to the version from the paper. There, every node is only asked once, because of a better message format, that combines the two messages from eMule's version into a single one (either the requested information is returned or a list of closer nodes). Another dissimilarity between eMule and the paper is that after a successful search the latter proposes to send the requested information to the node that has the closest ID but does not have the information stored. A possible reason for this is that the information is larger than the paper is assuming and that one would have to deal with the downsides, for example with the described problem of overcaching.
There are some drawbacks with eMule's implementation of the search procedure. For example, if a node needs more than 3 seconds to reply to a request for closer nodes, it will not be considered by the second process. This means, that it will not be asked for the actual request of a specific search, although its answer to the first process, a list of closer nodes, will be used. Maybe it is wiser not the ask such nodes, because a delay of more than 3 seconds could indicate an overload. A bigger weakness is located in the automatic search for sources, that was explained in the previous subsection. After a file in the download queue exceeds the amount of 50 sources, there will not be started any further searches for that file. Since this value cannot be changed by the user, it conflicts with the fact that the limit of total sources per file can be changed in the graphical user interface. If a significantly higher value is chosen there, it will never be reached, when only Kademlia is used. This problem was confirmed by tests with popular files. Since more sources result in higher downloading speeds, it would be useful to let the limitation in the

search for sources depend on the global one that can be set by the user. Also a more frequent search could help in a higher amount of sources.

2.7 Publishing

Every node that wants to share one or more files in the Kademlia system, has to publish them first, so that other nodes can search for it and download it. Therefore, that node needs to publish that it is a source for every file it is sharing, in order to allow others to find that node and add it to their list of sources. Moreover the filenames of every shared file have to be spread. This is done by publishing the keywords of every file that is shared with the Network. If a user commented a file he is publishing, this note is published too, so other users are able to retrieve this comment before downloading the file. Thus, there are three different types of publishes. Every node is responsible for publishing his own files and on the other hand every node will receive publishes from others that it has to store in the three corresponding maps, to be able to respond to the search requests it receives.
The next part will explain how the general procedure of a publish is implemented, followed by a differentiation of the three publish types. Then, the protection mechanism of so called hot nodes in keyword publishing is presented. After that, the behavior of the publishing process at startup is analyzed, to see when which type of publish is initiated for the first time. Finally, the publish procedure is compared to the one described in the paper.

2.7.1 Procedure of a Publish

Although there are three different types of a publish, the procedure is almost the same for all of them. As seen in the section about searching, a publish is done through the search system, using the search manager. In a publish, the first process is identical to a search, but the second differs, since a publish request is sent, instead of a search request. As publishing is performed automatically without the users notice, there is a process that goes through the shared file list and the list of all keywords and checks whether it is necessary to publish a file, keyword or a note. If it is the case, a new search object is created and the first process is initiated as if it was search. Once a node has answered in the first process, it is contacted by the second one with a `PUBLISH_REQ` message (or a `PUBLISH_NOTES_REQ` in the case of a notes publish). A node which receives such a message, stores the containing information into one of its maps according to the type of the publish. But there is one limitation, which causes the information not to be stored. This is the case when the ID of the information is differing to much from the nodes ID, since a node should only hold information whose ID is close to its own ID, because a search will always only ask a node for such information. The limitation is implemented through checking the XOR distance between the two IDs. If it is larger than 16'777'216, the information is not stored on that node. In other words the last 24 bits of the 128-bit IDs may be different. In case of a successful storage of the information, the node replies the publisher with a `PUBLISH_RES` message (or a `PUBLISH_NOTES_RES` for notes). If the information could not be stored, nothing is sent.[6] To achieve a redundant storage of information, publishing is not stopped until the information was replicated to 10 nodes. This fact can also be seen from the table, which was showed in Section 2.6.2. If for some reason such a replication cannot be reached by a publish (remember that there is also a time limit for every publish or search), no special action is taken.

2.7.2 The Three Publish Types

After seeing how the procedure of a publish works in general, this subsection will describe the differences between a file, keyword and a note publish.

- The **file publish** uses a list that contains all the shared files. The process cycles through this list and looks for files whose *publish time* has expired. When such a file is found, eMule starts a publish that contains the hash ID of this file and the owners ID, IP address and port, to be able to retrieve that user as a source for this file. Then, the publish time is set to 5 hours, which means that the next republish for this specific file will fall due in 5 hours. There are only 3 file publishes allowed at the time. If there are more, the others have to wait.

- For the **keyword publish** a list of keywords is maintained. It contains every keyword from all the files that are shared. The process rotates this list and checks every keyword for its expiration. If it is the case the keywords hash ID is published, including a list of all files that contain this keyword in their file name. An entry in this list consists of the file hash ID, the filename, the size and some tags, which describe the file (e.g. file type, duration for movies, bitrate for sound files). The republish time is set to 24 hours. Only 2 keyword publishes can run simultaneously. When publishing keywords a problem of overloaded nodes can occur. This will be discussed in the next subsection.

- The **note publish** uses the same file list as the file publish to identify those files which were commented by the user and whose publish time regarding the note has expired. If a note has to be published, a message with the corresponding file hash ID, the filename, the rating, the comment and the ID of the node that shares this file and commented it, is sent to the 10 closest nodes. A republish is done every 24 hours, as it is done with keywords. There are no parallel note publishes allowed.

[6]This does not hold for keyword publishes, as will be explained later.

Whenever the user changes a file name or adds a comment, the corresponding publish is immediately executed for that file. Since receiving publishes will fill the three maps more and more, there also has to be a process, that cleans them by removing old information. Therefore, every information stored in the maps has a timestamp, holding the time when it was stored. Every 30 minutes a cleaning process scans the maps and removes expired information. The expire time is identical to the republish time (24 hours for keyword and note entries and 5 hours for source entries).

2.7.3 Protection of Hot Nodes

It can be assumed that the hash IDs of all the shared files in the network are evenly distributed over the whole ID space. This follows from the fact that the files are hashed with a complex hash function. As a result every node is responsible for almost the same amount of files and therefore they all have a similar number of sources in their maps. Of course, not every shared file has the same popularity, so the number of sources can vary. But it can be assumed that there is no such file with a super popularity, meaning that everyone in the network would share it, as people have different interests. This cannot be proven, but the web site *eMule Top 50*[6] that lists the 50 most shared files, supports this assumption. So there is no node in the system which receives to many publish messages with sources, because of the facts mentioned before. The same holds for notes, because there are at most as many notes as there are sources for a specific file (which would be the case if everyone that shares a file also commented it).

With keywords it is different, since there exist very popular keywords. There are two reasons for that. The first one is based on the type of files being shared in eMule. Own researches have shown that these are mostly audio or video files. Their filenames very often contain words like "CD1", "CD2", "Radio Version", "Mix", a track number between 01 and 10 or the release year. The second reason for very popular keywords is language based. There are popular words in every language, for example "a", "of" and "the" in English. Such *stop words* are normally ignored by search engines like Google, because they appear to often and do not return usable results when searching for them. Therefore, it is very likely that the node which has an ID close to a super popular keyword will get overloaded by the publish requests it is receiving from all the other nodes. eMule calls these nodes *hot nodes*.

To prevent hot nodes from getting overloaded, a protection mechanism is implemented, that measures the load of the keyword map of a specific node. As described in the general procedure of a publish, when some node sends a publish message to another node, the latter will respond with a `PUBLISH_RES` message, indicating that it was able to store the information. This response also contains a load percentage of its appropriate map. In the case of a keyword publish, this value is evaluated to check whether the node is a hot node or not. A value of 100 % indicates that all of the 50,000 possible entries for a specific keyword ID are full, so for example 2 % would correspond to 1,000 entries. Whenever a keyword publish is executed, the protection mechanism analyzes all the received results and generates a mean value. If this value is greater than 20 % (which corresponds to more than 10,000 entries on *every* node) the published keyword is considered to be super popular and its closest nodes are considered hot nodes. To protect these overloaded nodes, the keyword is not published any more for a specific period of time. It is placed into a list that contains all currently banned keywords including the time until they are not allowed to be published. This duration is computed with the following formula:

$$t[days] = 7 * load_{normalized}$$

where $load_{normalized}$ is the average load that has to be normalized to 1. So, the duration t where this keyword is not allowed to be published varies between 1.4 and 7 days. To ensure that a publish is not started for this keyword after eMule was restarted, the list has to be stored permanently on disk (actually this list is implemented as a map, stored in the file `load_index.dat` and called `mapLoad` in the source code).
The protection mechanism can only work when the publish procedure for the keyword publish is slightly changed from the one described before (see Section 2.7.1), where the requested node does not reply if it cannot store the publish information. While this is acceptable for the file and notes publish, in a keyword publish the requested node always has to answer. Especially, this is necessary when it is overloaded, so that the publishing node receives the high load value and can prevent the specific keyword from being published again. If the requested node did not answer, the publishing node would have to try its neighbors to achieve the desired replication of the information. So, they would also be bothered with the same publish messages although this keyword already is super popular and publishing it to more nodes is needless.

2.7.4 Behavior at Startup

At startup the most important thing is to find some contacts that are alive. Afterwards the files that the user is sharing can be published. The be sure that the Kademlia network is running, publishing is started after the self lookup has successfully completed (see Section 2.5.2). The self lookup is initiated 3 minutes after connection to the network and takes approximately 30 seconds. Hence, the first publish will be started after this amount of time being connected to Kademlia, which matches the time measurements done when testing eMule. At this point of time the first three files, two keywords and the first note will start being published simultaneously.

2.7.5 Comparison with Paper & Discussion

When comparing eMule's implementation of the publishing process with the one from the Kademlia paper, there is one rather important feature missing. The paper proposes to republish the information that a node stores every hour. This means that an additional publish should be done every hour not by the original owner of a specific file, but by the nodes that were asked to store the information about that file. This idea seems to be very useful, since nodes join and leave the network very often and the owner of a file does not know when the nodes that store information about its file leave the system. Through a regular republishing, the information would not be lost when these nodes leave and new nodes that are closer to the ID would also receive it. Unfortunately, such a mechanism is not implemented in eMule.
As a negative point of eMule's publish procedure, the implementation of the tolerance for accepting a publish can be mentioned. Being fixed to the maximum distance of 24 bits, this seems to be alright for very large networks with hundred thousands of participants like the Internet, but eMule will with high probability not work in a small workgroup. There, the tolerance must be much larger or else the closest node is likely to be outside of it.

2.8 NAT and Firewall

A common problem for internet applications are firewalls and network address translation systems. The latter are widely spread, since they are integrated in almost every DSL internet connection, to share the public IP address among all connected computers. Firewalls are used in companies to protect their clients. This subsection addresses these two obstructions and shows how eMule overcomes them. As the report does not focus on this problem and the analysis of the corresponding code was difficult, this section only gives a rough overview.

2.8.1 Obtaining the Public IP

If network address translation is used on the client side, the clients IP address will differ from the external one, that must be used to contact it. Therefore, a mechanism has to detect it, using other clients that send back the observed address as follows: When sending a `FIREWALLED_REQ`[7] message to any client that uses eMule, it will answer with the `FIREWALLED_RES`[7] message containing the IP address of the initial sender, that wants to find out its public address. If it receives the same address from two different clients, it will use this address as its public IP. The mechanism uses a counter that starts with 0 and is increased every time the clients receives a message with its address. The `FIREWALLED_REQ` messages are sent as long as this counter

[7]The names of these two messages are misleading.

is smaller than 4, which gives a reserve of an additional message to the two needed ones. These messages are sent to such contacts which have sent a `HELLO_REQ`, `KADEMLIA_REQ` or a `KADEMLIA_RES` message, which means that they were asking if the client is still alive, asking for new contacts or replying with new contacts.

2.8.2 The Buddy System

If a client using eMule is located behind a firewall, the problem arises that it cannot be contacted by other clients. The only exception is when it starts a new connection, other clients can reply using this connection. With this handicap a firewalled client is useless for storing information, since it will not receive any publishes or searches. This would not be that bad, because only a minority of all clients is behind a firewall. But the main problem is that such clients will not find out when they can start downloading from another client, since the latter cannot contact them. Consequently they will not be able to download anything. To overcome this problem, eMule uses a so called buddy system, where a firewalled client is helped by a normal client, his so called buddy. If somebody wants to contact the firewalled client, he uses its buddy, which can be reached by everyone, to establish a connection.
But first every client that is joining the Kademlia network has to find out, whether it is behind a firewall or not. It cannot just ask another client to reply with a message to test for the presence of a firewall, because the latter would allow all incoming messages from a client that was contacted just before (otherwise no communication would be possible). For that reason there is a mechanism implemented that sends a separate `FIREWALLED_ACK_RES` message after a connection to another client was established. This means that any client will receive such messages when connecting to other clients (if it is not behind a firewall). When joining the network, eMule assumes that it is *firewalled.* This state is changed to *not firewalled*, when at least 2 such messages were received.
If after 5 minutes the state is still firewalled, eMule decides that the client is really behind a firewall and starts to search a buddy that will help others connecting to this client. The buddy search is based on the normal Kademlia search described in Section 2.6 except that in the second phase `FINDBUDDY_REQ` messages are sent instead of the ones used to retrieve or store information. The target ID of such a search is the inverted ID of the client that searches for a buddy, which means that every 0 of the ID becomes a 1 and every 1 a 0. This ensures a even distribution of buddies and that any node will receive at most one request to be a buddy. If some node receives such a message, it checks whether it already is a buddy of another node. If this is not the case, the request is accepted and the node replies with a `FINDBUDDY_RES` message. Once the client that is looking for a buddy

receives such a reply, it will establish a TCP connection with the sender and set him as its buddy. This permanent connection between the buddy and the firewalled node allows the buddy to overcome the firewall and contact the node behind it at any time, where other nodes can contact it only via a reply (they must have received a message from the node behind the firewall before they can send anything to it). The firewalled node sends every 10 minutes a `BuddyPing` message to its buddy to check whether it is still alive. The buddy then replies with a `BuddyPong` message. If it fails to reply, the firewalled node starts to search for a new buddy.

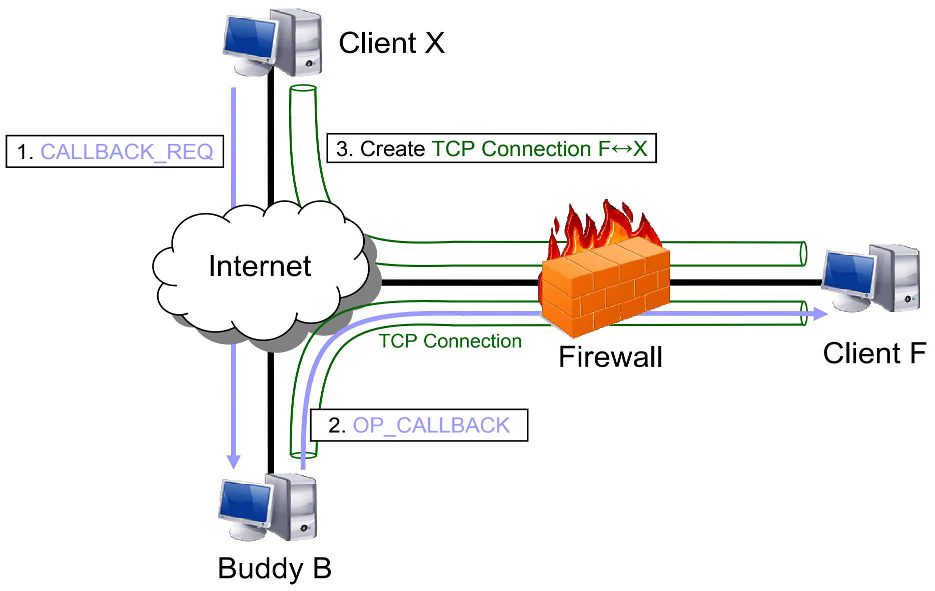


Figure 5: Establishing a connection to a firewalled client

There are two main reasons that a node behind a firewall has to be contacted. Either another node is requesting to download a file from it or somebody wants to inform this firewalled node that it is on top of its queue and can start downloading the requested file. Searching and publishing is not performed with firewalled clients. The connection from a client X to a firewalled client F with buddy B is established via callback as depicted in Figure 5. Client X knows that client F is not directly reachable and therefore sends a `CALLBACK_REQ` message to its buddy B (either X knows its IP address already or has to search for it by inverting the ID of F). B has an open connection to F, so it forwards the callback through it. The message is now sent over TCP, so the type of the message changes to `OP_CALLBACK`. F receives this message and extracts the IP address and port of client X from it. Now, F can establish a TCP connection to X and wait for the messages, that X wanted to send to F.

2.9 Estimating the Number of Users in Kademlia

eMule has a function to estimate the size of the Kademlia network. From this number it also derives the number of currently shared files in the whole network. At this time, these two values are only computed for the purpose of informing the user. But it seems like they will be spread to all contacts of the client in future releases. This could be very useful to adapt the routing algorithms in Kademlia to the actual size of the network. The number of users and the number of shared files is visualized in the status bar at the bottom of eMule's GUI (see Figure 7 for a screenshot of the GUI). The first value shows the estimate for the server network and the following value, which is in parenthesis, shows the estimate for the Kademlia network. The same holds for the number of shared files, which is displayed next to the user numbers.
The function that estimates the number of users is a complicated operation on the routing tree. For every routing zone in the tree a method is called, that computes the number for this part of the tree. At the end, the maximum of all values is taken. The method jumps 3 levels up in the tree and then applies the following formula:

$$users = 2^{level-2} * 10 * f_{modify} * 1.2$$

where *level* corresponds to the level of the routing zone in the tree. 10 stands for the maximum number of contacts in one routing bin and 1.2 is a correction factor for firewalled nodes, that are not visible. f_{modify} is supposed to be a factor, which reduces the number of users, when there are bins in that part of the tree, that are not fully occupied with contacts (it is computed as the total of all contacts in that part of the tree divided by 20). For routing zones that are located in one of the first 3 levels, the maximum number of possible contacts on that level is returned instead.
The number of files shared in the network is computed using the number of users. Therefore, the latter is multiplied with the average number of shared files per user on all known eDonkey servers. If no server is known, then the size of the keyword map (see Section 2.2) is used instead.
Both estimates seem to be problematic. eMule's computation of the number of users only depends on the size of the routing tree. So, if the frequency of finding new contacts is changed (which will result in either more or less contacts), this will also change the number of users. Using numbers from servers in the case of the computation of the shared files, is not correct since it is not proven that these numbers are equal in both networks. In addition Section 2.7.3 shows, that the size of the keyword map strongly varies depending on which ID a client has chosen. Some parts of the computation

are not even comprehensible. Because of all these reasons, it is doubtful whether the estimates are of any use.

3 Attacking the Kademlia Network

This section examines several attacks on the Kademlia network. First, the ones that are very difficult to realize are presented. Then an attack is suggested, which was successfully implemented and that is working.

3.1 Attacks that do not work

The *eclipse attack* tries to surround a client in the network, so that it will fail to communicate with other clients. This would require to replace all existing contacts in that clients routing table. But this is not possible in eMule, as new contacts cannot replace old ones. New ones are only added, if there is space available in the routing tree. The old ones will still exist and will only be deleted if they fail to respond.
One can imagine an attack where a client is flooded with publish messages, with the intention of removing the original information or bringing the client to crash. But it is not possible to replace the original information with corrupt data, as entries are only removed from the corresponding maps, if they are too old. Since other nodes which hold the original data, will also publish it from time to time, such an attack will fail.

3.2 Network Poisoning Attack

The network poisoning attack tries to corrupt the network by spreading defective information. There are many conceivable possibilities to poison the Kademlia network, but only a few are successful. The most effective attack that was found, restrains user from finding useful entries for a specific keyword, although they are present in the network. This means, that while the attack is running, a search for this keyword will not return the original results. Instead, entries are returned that were arbitrary chosen by the attacker. This network poisoning attack will now be discussed in detail in this section.
It is not possible to replace information that is stored on several nodes for a specific keyword, because these nodes will only *add* new information to their maps. So, if one would publish defective information, there would still be other clients that publish useful information, and the latter could also be retrieved. To avoid that, all the search requests for this specific keyword have to be routed to the attacker and not to the nodes that hold the original information. This is actually very simple, because the ID of every client is chosen randomly by the client itself. Therefore, the attacker just has to join the network with an ID that matches the hash ID of the keyword, which

is going to be poisoned. Then, he will be the closest node to this ID and will therefore receive all search requests for that keyword. The nodes that store the original information for that keyword, will always have a bigger distance to the keyword hash ID than the attacker[8] and will only be asked, if the latter did not deliver enough results. As a consequence, the attacker is able to control the information that is bound to the attacked keyword. He only has to provide enough results to the search requests and these have to match the keywords that the search requests contain.

3.2.1 Implementation

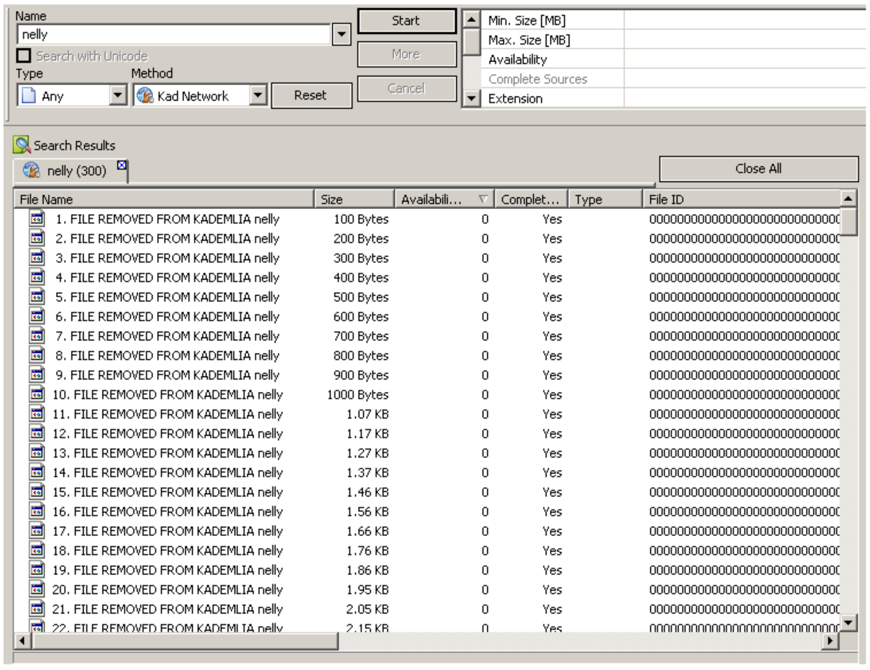
To prove the success of the network poisoning attack, it was implemented and tested. The user guide for the modified version is provided in Appendix B. There are two possibilities to implement this attack. Either a new application has to be built, which connects to the Kademlia network, spreads its ID to other clients and replies to search requests with poisoned data. The other possibility is to modify the original eMule application, so that it suits the requirements for this attack. As it seemed to be easier and faster to modify the existing source code, than to create a new application, the second possibility was chosen.
But first, the original source code of eMule (version 0.47a) had to be compiled. This was more problematic than one would expect. An older version of Microsofts Visual Studio is necessary to compile it without errors (version .NET 2002). eMule uses several external libraries, that have to be obtained separately and have to be in the right version. Even then, it was not possible to complete the compilation (there were errors in the linking part of the compilation, which indicate problems with the libraries). The solution to this problem was to obtain the sources of an already modified version of eMule (a so called *MOD*). There are a lot of such versions, which offer extra features, as the source code was developed under the *GNU General Public License* (GPL)[7]. The eMule MOD webpage[8] gives an overview of the most used versions. It was possible to successfully compile the version from "Sivka"[9] (version 17b1 alpha), because it also contains all needed libraries. Then the modified source code was replaced by the original version. Now everything was set, to perform the modification necessary for the attack.
To take over the ID of the attack keyword and set it as the ID of the client, the search tab in eMule's GUI was extended (see Figure 8 for a screenshot). It now contains a field that displays the hash ID of the keyword, that is being searched. The user can then set this ID as the client ID by pressing the corresponding button.
The second part of the modification concerns the function that processes the incoming `SEARCH_REQ` messages and sends back `SEARCH_RES` packets. The

[8]This can be assumed because of the extremely small probability that any client in the network has chosen exactly the ID of the keyword.

modified function does not look up the received keywords in the keyword map. Instead, it produces 300 result tuples that have the words “FILE REMOVED FROM KADEMLIA” in their filename. Such a large amount of results is necessary, to stop the search procedure on the client that was searching for this keyword (otherwise that client would ask another node and receive the original information). Additionally the filename must contain all the keywords to match the search request, so they are appended at the end of it. Every result tuple also contains a file size and a file hash ID (they differ in each tuple, to pretend to be different files).

3.2.2 Results

Tests have been made with different keywords. A small problem that arose, was the fact that the first keyword is not contained in the search request (see Appendix A for the message format). This is not necessary, because its hash ID is included in the message. But for the attack it is needed in text form, so it can be appended to the filename. Therefore, this keyword has to be stored in eMule. This is done every time, when the own ID is changed to a new attack ID in the search tab.



Name: nelly
☐ Search with Unicode
Type: Any
Method: Kad Network

Min. Size [MB]	
Max. Size [MB]	
Availability	
Complete Sources	
Extension	

nelly (300)

File Name	Size	Availabili...	Complet...	Type	File ID
1. FILE REMOVED FROM KADEMLIA nelly	100 Bytes	0	Yes		000000000000000000000000000
2. FILE REMOVED FROM KADEMLIA nelly	200 Bytes	0	Yes		000000000000000000000000000
3. FILE REMOVED FROM KADEMLIA nelly	300 Bytes	0	Yes		000000000000000000000000000
4. FILE REMOVED FROM KADEMLIA nelly	400 Bytes	0	Yes		000000000000000000000000000
5. FILE REMOVED FROM KADEMLIA nelly	500 Bytes	0	Yes		000000000000000000000000000
6. FILE REMOVED FROM KADEMLIA nelly	600 Bytes	0	Yes		000000000000000000000000000
7. FILE REMOVED FROM KADEMLIA nelly	700 Bytes	0	Yes		000000000000000000000000000
8. FILE REMOVED FROM KADEMLIA nelly	800 Bytes	0	Yes		000000000000000000000000000
9. FILE REMOVED FROM KADEMLIA nelly	900 Bytes	0	Yes		000000000000000000000000000
10. FILE REMOVED FROM KADEMLIA nelly	1000 Bytes	0	Yes		000000000000000000000000000
11. FILE REMOVED FROM KADEMLIA nelly	1.07 KB	0	Yes		000000000000000000000000000
12. FILE REMOVED FROM KADEMLIA nelly	1.17 KB	0	Yes		000000000000000000000000000
13. FILE REMOVED FROM KADEMLIA nelly	1.27 KB	0	Yes		000000000000000000000000000
14. FILE REMOVED FROM KADEMLIA nelly	1.37 KB	0	Yes		000000000000000000000000000
15. FILE REMOVED FROM KADEMLIA nelly	1.46 KB	0	Yes		000000000000000000000000000
16. FILE REMOVED FROM KADEMLIA nelly	1.56 KB	0	Yes		000000000000000000000000000
17. FILE REMOVED FROM KADEMLIA nelly	1.66 KB	0	Yes		000000000000000000000000000
18. FILE REMOVED FROM KADEMLIA nelly	1.76 KB	0	Yes		000000000000000000000000000
19. FILE REMOVED FROM KADEMLIA nelly	1.86 KB	0	Yes		000000000000000000000000000
20. FILE REMOVED FROM KADEMLIA nelly	1.95 KB	0	Yes		000000000000000000000000000
21. FILE REMOVED FROM KADEMLIA nelly	2.05 KB	0	Yes		000000000000000000000000000
22. FILE REMOVED FROM KADEMLIA nelly	2.15 KB	0	Yes		000000000000000000000000000

Figure 6: Results when searching for the attacked keyword

In addition, a second problem was found during testing. In most of the cases it happened, that a search request was not only sent to the attacker,

but also to other nodes which were further away from the target ID. This is because of the delay caused when transmitting the 300 search results. eMule sends another request before all of the results are received from the attacker, even though there are enough of them. This leads to the problem, that also original results are shown in the search result. The attack fails. To prevent this, the attacker also has to own the second, third, etc. closest IDs to the keyword ID. This means, that he has to run several modified clients, whereof one has the keyword ID as client ID and the others have IDs, such that their distance to the keyword ID is as small as possible.
The tests showed, that the modified client first has to spread its ID to other nodes, before the attack works. It was found that this is the case after approximately 15 minutes. Figure 6 shows how the search results look like, if the attack is active.

3.2.3 Protection

It is possible to prevent the network poisoning attack presented in this section. Therefore, the ID of a client must not be set randomly, as this way the attacker can choose the ID he needs to perform the attack. This can be achieved if the ID of a client is made dependent from its IP address and UDP port (the port is needed for clients using NAT). Then, the IDs of received messages could be compared with the senders IP addresses and illegal IDs could be ignored. One way to make the ID dependent from the IP address and the UDP port, is to hash these values.
There are two ways to obtain the original information belonging to a keyword, if a network poisoning attack is active on this specific keyword. Very often, the search request consists of more than one keyword. Then, exchanging the first keyword with another one in the search phrase will lead to a totally different target ID of the search, which may be free of any attacks. The second possibility is to use the server based search within eMule, if the search in Kademlia was not satisfying.

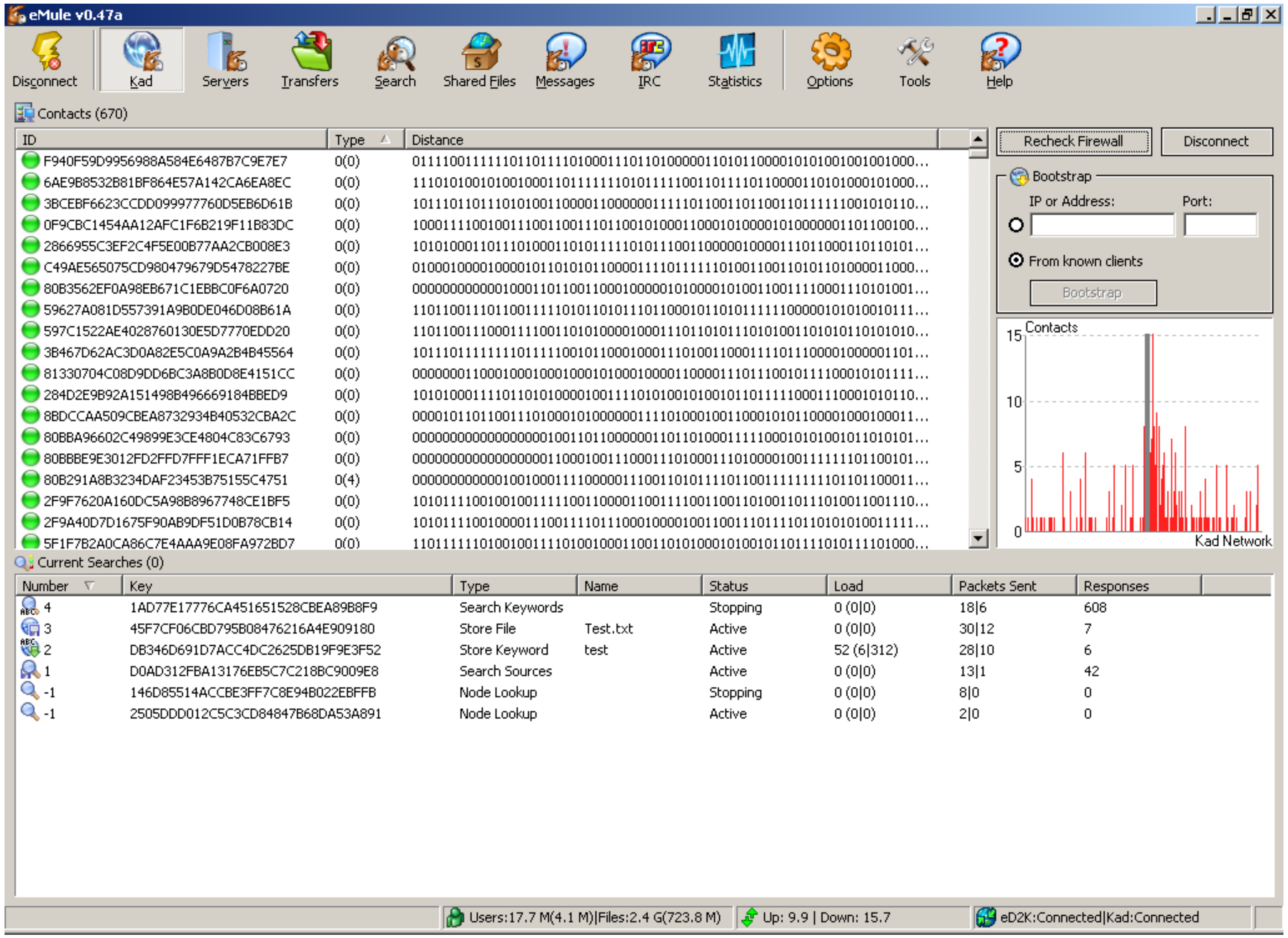
4 GUI

The graphical user interface of eMule is divided into several tabs, which visualize important information, like the status of the connected networks, the files being currently downloaded and many more. In addition, the user can interact with eMule and is able to search for new files, manage the downloads or change the settings of eMule. The GUI can be seen in Figure 7. The tab, that shows information about the Kademlia network is named “Kad”. It was activated in this figure and is therefore displayed underneath the tab buttons. The Kademlia tab consists of four sections: A list of contacts in the upper left area, bootstrap and connect buttons in the upper

right corner, a visualization of the contacts beneath and a list of active searches in the lower area. These sections will now be discussed in detail.
The list of contacts shows all contacts that are currently contained in the routing tree. Every entry is composed of an icon, the contacts ID, its type and the distance to our ID. The color of the icon corresponds to the type of the contact (see Section 2.4.3 for more details). The type field contains two numbers. The first one is the type of the contact, thus it shows how old and reliable that contact is. The second number, which is in parenthesis, shows the version of eMule the contact is running. The following table presents the possible entries:

Number	Version
0	unknown
1	0.46c
2	0.47a
3	0.47b
4	0.47c

It has to be remarked that this version is only sent via the new Kademlia 2.0 protocol, that is used from version 0.47c. Older versions (back to 0.46c) understand the new protocol, but do not use it except when they receive a request sent through the new protocol. As a consequence such clients will only display the version if it is 0.47c, like in the figure. If the number displayed is 0, the corresponding client does not support the new protocol. The distance field in the contact list shows the distance (binary representation) between the contacts ID and our ID, as it is used in the routing tree.
The *Recheck Firewall* button is useful when the user changed its firewall settings, so that the traffic is no more blocked and wants eMule to notice this (apart from that the firewall status is only checked once every hour). When this button is pressed the entries for the firewall status and the public IP are cleared. After that, the public IP is determined again and the mechanism for detecting a firewall, as described in Section 2.8.2, is started. The *Connect* (respectively *Disconnect*) button either starts or stops the Kademlia network. When the *Bootstrap* button is pushed, eMule first connects to the Kademlia network (as if the connect button was pushed), but only if it is not already connected. Then the bootstrapping procedure is initiated with the IP address and the port of another client, as described in Section 2.5.3. If instead the radio button "From known clients" is chosen, nothing will happen when the *Bootstrap* button is pressed, because this case was not implemented yet. Notice that the second possibility of bootstrapping, namely the one with the sources from a server, is executed independently of the *Bootstrap* button being pressed or not. This is because of the fact that the bootstrapping from sources is started automatically when there are no contacts available.



Contacts (670)

ID	Type	Distance
F940F59D9956988A584E6487B7C9E7E7	0(0)	0111100111111011011110100011101101000001101011000010101001001001000...
6AE9B8532B81BF864E57A142CA6EA8EC	0(0)	1110101001010010001101111111010111110011011110110000110101000101000...
3BCEBF6623CCDD099977760D5EB6D61B	0(0)	1011101101110101001100001100000011111011001101100110111111001010110...
0F9CBC1454AA12AFC1F6B219F11B83DC	0(0)	1000111100100111001100111011001010001100010100001010000001101100100...
2866955C3EF2C4F5E00B77AA2CB008E3	0(0)	1010100011011101000110101111101011100110000010000111011000110110101...
C49AE565075CD980479679D5478227BE	0(0)	0100010000100001011010101100001111011111101001100110101101000011000...
80B3562EF0A98EB671C1EBBC0F6A0720	0(0)	0000000000001000110110011000100000101000010100110011110001110101001...
59627A081D557391A9B0DE046D08B61A	0(0)	1101100111011001111101011010111011000101101011111100000101010010111...
597C1522AE4028760130E5D7770EDD20	0(0)	1101100111000111100110101000010001110110101110101001101010110101010...
3B467D62AC3D0A82E5C0A9A2B4B45564	0(0)	1011101111111101111100101100010001110100110001111011100001000001101...
81330704C08D9DD6BC3A8B0D8E4151CC	0(0)	0000000110001000100010001010001000011000011101110010111100010101111...
284D2E9B92A151498B496669184BBED9	0(0)	1010100011110110101000010011110101001010010110111110001110001010110...
8BDCCAA509CBEA8732934B40532CBA2C	0(0)	0000101101100111010001010000001111010001001100010101100001000100011...
80BBA96602C49899E3CE4804C83C6793	0(0)	0000000000000000001001101100000011011010001111100010101001011010101...
80BBBE9E3012FD2FFD7FFF1ECA71FFB7	0(0)	0000000000000000011000100111000111010001110100001001111111101100101...
80B291A8B3234DAF23453B75155C4751	0(4)	0000000000001001000111100000111001101011110110011111111101101100011...
2F9F7620A160DC5A98B8967748CE1BF5	0(0)	1010111100100100111110011000011001111001100110100110111010011001110...
2F9A40D7D1675F90AB9DF51D0B78CB14	0(0)	1010111100100001110011110111000100001001100111011110110101010011111...
5F1F7B2A0CA86C7E4AAA9E08FA972BD7	0(0)	1101111110100100111101001000110011010100010100101101111010111101000...

Current Searches (0)

Number	Key	Type	Name	Status	Load	Packets Sent	Responses
4	1AD77E17776CA451651528CBEA89B8F9	Search Keywords		Stopping	0 (0|0)	18|6	608
3	45F7CF06CBD795B08476216A4E909180	Store File	Test.txt	Active	0 (0|0)	30|12	7
2	DB346D691D7ACC4DC2625DB19F9E3F52	Store Keyword	test	Active	52 (6|312)	28|10	6
1	D0AD312FBA13176EB5C7C218BC9009E8	Search Sources		Active	0 (0|0)	13|1	42
-1	146D85514ACCBE3FF7C8E94B022EBFFB	Node Lookup		Stopping	0 (0|0)	8|0	0
-1	2505DDD012C5C3CD84847B68DA53A891	Node Lookup		Active	0 (0|0)	2|0	0

Users:17.7 M(4.1 M)|Files:2.4 G(723.8 M) | Up: 9.9 | Down: 15.7 | eD2K:Connected|Kad:Connected

Figure 7: The Kademlia tab in the graphical user interface of eMule

The visualization box shows how much populated the routing tree is and how the contacts are distributed in the ID space. The x-axis of the graph represents the whole ID space, going from the smallest ID on the left side to the largest one on the right side. A red bar on the y-axis shows how many contacts are located in a specific region of the ID space. A grey bar indicates the area with the greatest amount of contacts. This area also contains our ID, because the greatest amount of contacts stored in the routing tree is close to our ID.
The list of searches in the bottom half of the tab contains all currently active searches. In other words, this is a representation of the search managers list of searches. It can be very helpful to observe this list, when trying to understand how searches work in eMule, because it visualizes a lot of interesting information. The list has the following eight fields:

- The **Number field** contains an icon to quickly see of which type a specific search is. These icons were presented in Section 2.6. The icon is followed by the search number, which is used to identify a specific search. A counter assigns every new search a different value. The only exception are node lookup searches, that always have the value -1 there.

- The **Key field** holds the target ID, thus it shows which ID a search is looking for.

- The **Type field** shows the type of the search, to differentiate between the several types of searches (Node Lookup, Search Keywords, Search Sources, Search Notes, Store Notes, Store File, Store Keyword).

- The **Name field** is only used when a file or a keyword is published. It then contains the file name (in case of a file publish) or the keyword. For any other type of search this field is empty.

- The **Status field** informs the user whether a search is still “Active” or already “Stopping”. The latter means that the search produced enough results (or the time limit was reached) and will be removed from the search manager in a few seconds.

- With the **Load field** it is possible to see the popularity of a keyword that is being published. As described in Section 2.7.3 super popular keywords have to be revealed and no more published, in order to avoid overloading of the corresponding nodes. When a keyword publish is running, the requested nodes will return their load value. The first number in the Load field is the mean value (in percent) of all the returned load values. If this value is greater than 20, the related keyword is considered super popular and will not be published for a specific amount of time. In Figure 7 there is such a keyword publish, that has a high load. After the first number in this field, there are two additional numbers in parenthesis. The first one counts the number of nodes that responded with a load value and the second one shows the cumulated load over all these nodes. As a consequence, the mean load value is obtained when dividing the cumulated load by the number of nodes.

- The **Packets Sent field** contains two numbers, because the search procedure sends two different types of messages to other nodes. The first number shows the counter for the `KADEMLIA_REQ` messages. They are used in the first phase to obtain closer nodes to the target ID of the search. The second number counts the actual requests (`SEARCH_REQ`, `SEARCH_NOTES_REQ`, `PUBLISH_REQ` or `PUBLISH_NOTES_REQ`) that are sent in the second phase of a search.

- The **Responses field** shows how successful a search is. If the search is looking for sources, keywords or notes the value corresponds to the number of received sources, files (with filenames containing the keywords) or notes. If it is a publish, then the value is equivalent to the number of nodes that successfully stored the information (and replied with a `PUBLISH_RES` or `PUBLISH_NOTES_RES` message). If the search is a node lookup, the value corresponds to the number of received `KADEMLIA_RES` messages (which is not equal to the amount of new nodes received).

5 Outlook

The eMule project is under constant development. A log that lists all changes for every version can be found here[10]. The version that was analyzed in this thesis has still not reached the release status, as the number is under 1.0. The developers plan to improve the Kademlia protocol in future versions. It will then be called Kad 2.0, but it is not clear what improvements will be made. A part of the new protocol is already implemented in the source code that was analyzed. The only change visible there, is a better message format. In particular, more information is sent about a client, when it joins the network.
From version 0.47b eMule supports protocol *obfuscation*. Thereby, transmitted data will appear as random data on the first sight. This helps to prevent recognizing that eMule is used on a specific connection. But, obfuscation is not supported for Kademlia messages.

6 Conclusions

The analysis of eMule's source code showed the following three flaws: Almost all algorithms that are used to process the internal data structure, are naive versions and therefore CPU intensive (e.g. linear search is used very often). The second flaw can be found in the implementation of Kademlia's algorithms. Constant parameters are used, which were set to a fixed value, instead of allowing them to adapt to the rest of the application. One example is the automatic search for sources, which does not obtain the maximum sources per file value from eMule's global settings, as described in Section 2.6.4. The third flaw is the ability of attacking the Kademlia network by poisoning, as described in Section 3.2. This attack could be prevented, if IDs were used, that depend on the users IP address.
Despite all the flaws that exist in eMule, this file sharing client has proven to work very well in the last few years, as it is used by an enormous number of people. A positive aspect is the fact that eMule uses a theoretical basis (Kademlia) for the implementation of the peer-to-peer part, instead of having developed a proprietary version. As a consequence eMule's peer-to-peer functionality is solid and efficient. Moreover, eMule uses three redundant ways of obtaining sources for its downloads (which is the most essential function in a file sharing network), making downloading files very reliable.

A Kademlia Message Types

This section serves as a reference for all message types that are used to communicate between eMule clients in the context of the Kademlia protocol. Every message is sent via UDP, except the two last messages listed here (`BuddyPing` and `BuddyPong`), which are sent vie TCP. A message is either a request or a response. This can be seen from the suffix of the name of a message (the name ends with either `_REQ` or `_RES`).
The format of every message is presented in the following subsections. Figures are used to illustrate the content of the messages. They show all the fields that are enclosed in a specific message, starting at the left side. For each field, its name and size (in *Bytes*) is shown. The latter is written in squared brackets. Every Kademlia message starts with the same header of 1 Byte length. It contains the value 0xE4[9], which is referred as `OP_KADEMLIAHEADER` in the source code. For reasons of space, this header is *not* shown in the following figures. The header is followed by the message type field. To distinguish it from the other fields, its text is colored green. Some message types contain a list of contacts, results, files etc. Such a list consists of several fields that are repeated for a specific amount of times. These fields are highlighted blue in the figures. The message types from the Kademlia 2.0 protocol are not presented here.

A.1 Hello

Used to test whether a contact is alive and responding, similar to a "ping" (see Section 2.4.3). The format of the `HELLO_REQ` message is:

0x10 [1]	Kad ID [16]	IP Address [4]	UDP Port [2]	TCP Port [2]	0x00 [1]

The fields contain information about the sender. The last field is not used at the moment.
To answer, a `HELLO_RES` message is sent:

0x18 [1]	Kad ID [16]	IP Address [4]	UDP Port [2]	TCP Port [2]	0x00 [1]

It contains exactly the same fields as the first message.

A.2 Bootstrap

A `BOOTSTRAP_REQ` message requests the recipient to reply with 20 contacts from its routing tree, to speed up the connection process (refer to Section 2.5.3). The message has the same format as the *hello* messages:

[9]Hexadecimal representation

0x00	Kad ID	IP Address	UDP Port	TCP Port	0x00
[1]	[16]	[4]	[2]	[2]	[1]

The reply is a `BOOTSTRAP_RES` message that contains several contacts. The format is:

0x08	NumOfContacts	Kad ID	IP Address	UDP Port	TCP Port	0x00
[1]	[2]	[16]	[4]	[2]	[2]	[1]

A contact consists of the five fields that are marked blue. These fields are repeated as many times as specified in the second field. The last field of the contact data has no function.

A.3 Search

The `KADEMLIA_REQ` message is used to obtain closer contacts to the specified ID. It has the following format:

0x20	NumOfContactsRequested	Target ID	Receivers ID
[1]	[1]	[16]	[16]

Either 2, 4 or 11 contacts can be requested (more details in Section 2.6.2). They should be as close as possible to the target ID. The receiver only answers if his ID is written in the last field.
The reply is a `KADEMLIA_RES` message with following format:

0x28	Target ID	NumOfContacts	Kad ID	IP	UDP	TCP	Type
[1]	[16]	[1]	[16]	[4]	[2]	[2]	[1]

The target ID is the same as in the request. The second field describes how many contacts are sent. A contact consists of the blue fields (the contacts Kad ID, its IP address, UDP and TCP ports and its type). They are repeated for every contact, if more than one is sent. The type was discussed in Section 2.4.3.

The actual search request for sources or keywords (see Section 2.6.2) is done by sending a `SEARCH_REQ` message, having this format:

0x30	Target ID	Sources Flag	Search Tree {*optional*}
[1]	[16]	[1]	[*arbitrary*]

A sources search has the sources flag set to true and has no search tree. The target ID is the file hash ID of the file, for which sources have to be found. A keyword search has the sources flag set to false and the target ID is the

hash ID of the first keyword. Additional keywords, if present, are contained in the last field as an expression in the manner of a tree (complicated search expressions with boolean operators are supported). The first keyword is never sent in text format.
If a node finds a match in its maps, it will reply with a `SEARCH_RES` message:

0x38	Target ID	NumOfResults	Result ID	Tag List
[1]	[16]	[2]	[16]	[*arbitrary*]

The target ID is the same as in the request. The results are sent as tuples of a result ID and a tag list, repeated as many times as specified in the third field. In case of a sources search, every tuple represents a client that is a possible source for the requested file. The result ID corresponds to a client ID and the tag list contains the IP address, the TCP and UDP port of that client. In case of a keyword search, every tuple represents a file that contains the keywords in its filename. The result ID then corresponds to the file hash ID and the tag list contains the filename, file type, file size and additional file specific tags. The message can be split into several packets, if they are to big.

If the search is looking for notes, the actual search request is a `SEARCH_NOTES_REQ` message, with this format:

0x32	Target ID	Senders ID
[1]	[16]	[16]

The target ID corresponds to the file hash ID for which the notes are being searched.
The response is a `SEARCH_NOTES_RES` message, having the following format:

0x3A	Target ID	NumOfResults	Kad ID	Tag List
[1]	[16]	[2]	[16]	[*arbitrary*]

The target ID is the same, as in the request. The result is a set of tuples which consist of the Kademlia ID of the user that created the comment and the tag list that contains the user name, file name, rating and comment. The amount of tuples that are sent in the response is defined in the third field.

A.4 Publish

If a node wants to publish a *keyword*, it uses the `PUBLISH_REQ` message, with this format:

0x40	Keyword Hash ID	NumOfTuples	File Hash ID	Tag List
[1]	[16]	[2]	[16]	[*arbitrary*]

It publishes a set of tuples, where every tuple represents a different file that contains the keyword in its filename. It contains the file hash Id and a tag list with the filename and the file size.
If the node wants to publish itself as a *source* for a file it is sharing, it also uses the `PUBLISH_REQ` message, but with this format:

0x40	File Hash ID	0x01	Senders Kad ID	Tag List
[1]	[16]	[2]	[16]	[*arbitrary*]

It is only possible to publish one source at the time (which is the sender itself, see Section 2.7.2). Therefore, the third field contains a 1. The tag list contains the filename, file size, the TCP Port of the sender and a tag which indicates that this publish request contains a source and not a keyword.
The response is a `PUBLISH_RES` message with the following format:

0x48	Item Hash ID	Load {*optional*}
[1]	[16]	[1]

The item hash ID is the file hash ID in source publishes or the keyword hash ID in keyword publishes. If it is a response to a keyword publish, the load of the keyword map is also sent (more details in Section 2.7.3).

A note is published using the `PUBLISH_NOTES_REQ` message:

0x42	File Hash ID	Source ID	Tag List
[1]	[16]	[16]	[*arbitrary*]

The file hash ID belongs to the file that is being commented. The source ID is the Kademlia ID of the user that is publishing the note. The tag list contains the user name, the filename, the file rating and the comment (see Section 2.3).
If a node could successfully store the note, it will reply with a `PUBLISH_NOTES_RES` message, having this format:

0x4A	File Hash ID	Load
[1]	[16]	[1]

The file hash ID is the one from the request. The load value indicates how full the mapNotes is (from 1 to 100, in percent). In contrast to the keyword publish, the load value is not processed here.

A.5 NAT and Firewall

To obtain the public IP Address (see Section 2.8.1) a `FIREWALLED_REQ` message is sent, having the following format:

0x50	TCP Port
[1]	[2]

It contains only the TCP Port of the sender.
The response is a `FIREWALLED_RES` message:

0x58	IP Address
[1]	[4]

It contains the public IP Address of the user that sent the appropriate request message.

To check whether a user is behind a firewall or not, a `FIREWALLED_ACK_RES` message is sent to every new contact (more details in Section 2.8.2). It does not contain any information (apart from the message type):

0x59
[1]

To find a buddy, a `FINDBUDDY_REQ` message is sent in the second phase of the search procedure. The format is:

0x51	Buddy ID	Client Hash	TCP Port
[1]	[16]	[16]	[2]

The buddy ID is the target of this search. It is the inverted ID of the sender (see Section 2.8.2). The third field contains the senders client hash, which is explained in Section 2.1. It is not equal to the Kad ID. The forth field holds the senders TCP port.
If a node receives such a buddy request and accepts to be the buddy of the sender, it replies with a `FINDBUDDY_RES` message:

0x5A	Buddy ID	Client Hash	TCP Port
[1]	[16]	[16]	[2]

It has the same format as the request. The buddy ID is identical, but the client hash and the TCP port are the ones from the buddy. The buddy's client hash is then stored by mistake as the Kad ID. But this is not a problem, as it is not used to contact the buddy.

To contact a node that is firewalled, a `CALLBACK_REQ` message is sent to its buddy, having the following format:

0x52	Buddy ID	File Hash ID	TCP Port
[1]	[16]	[16]	[2]

The buddy ID has to match the Kad ID of the receiver, so that the request is forwarded to the firewalled node. The third field holds the file hash ID of the requested file (either a download request from the firewalled node or a notification that the firewalled node can start downloading a file from the sender). The senders TCP port in the forth field is necessary, because a TCP connection has to be established between the firewalled node and the sender (see Section 2.8.2).

To check whether the buddy is still alive a `BuddyPing` message is sent:

0x9F
[1]

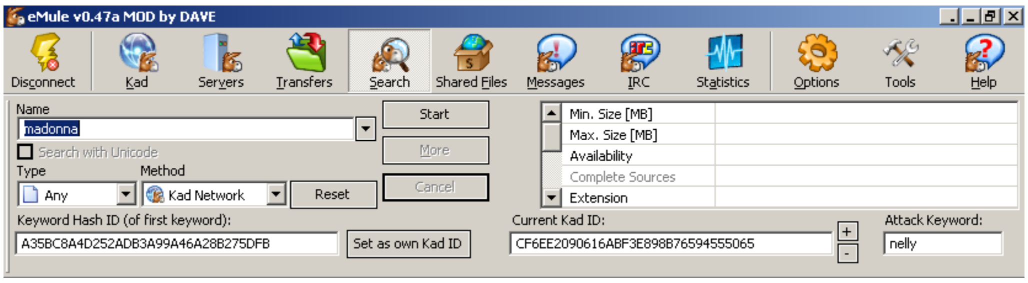
It is transmitted via *TCP*. Therefore, the header (which is not shown in the message figures) is different from the UDP messages. It is called `OP_EMULEPROT` and has the code 0xC5.
The buddy answers with a `BuddyPong` message:

0xA0
[1]

It is also empty (apart from the message type field) and sent through TCP.

B User Guide for the modified eMule Version

This user guide provides all information needed to perform an attack to a specific keyword, as described in Section 3.2. Before the attack can be started, the modified version of eMule has to be installed on a computer that uses the Microsoft Windows operating system. In addition, the computer either must not be behind a firewall or the TCP and UDP ports of eMule have to be open (this can be tested in the options dialog of eMule). When clicking on the *search tab*, one can see the modified user interface, as depicted in Figure 8. It contains additional fields and buttons, that are needed to control the attack.



Name: madonna
Type: Any
Method: Kad Network
Keyword Hash ID (of first keyword): A35BC8A4D252ADB3A99A46A28B275DFB
Current Kad ID: CF6EE2090616ABF3E898B76594555065
Attack Keyword: nelly

Figure 8: The modified version of the search tab

First, the Kademlia network has to be started. Now, the *Current Kad ID* field shows the actual ID of this client and the *Attack Keyword* field shows the actual keyword that is being attacked (if no keyword is being attacked, the field is empty). To initiate the attack, a normal search for the chosen attack keyword has to be performed, by typing it in the *Name* field and pressing the *Start* button. The hash ID corresponding to the keyword typed in, will be displayed in the *Keyword Hash ID* field. In order to attack this keyword, its hash ID has to be set as the own Kad ID. This can be done through the *Set as own Kad ID* button. Now, the field showing the clients Kad ID should change to the ID of the keyword and the keyword should be displayed in the *Attack Keyword* field. At this point, the Kademlia network should be stopped and restarted, since the ID of the client has changed. After that, the attack is active, which means that this client will answer all search requests with poisoned results.
As tests showed that more than one node is asked at the same time, when searching for a keyword (refer to Section 3.2.2), additional clients have to be set up to successfully realize the attack. In most cases two or three instances will be enough (the more instances, the greater the probability of a successful attack). Either each instance of the modified eMule version can run on a separate computer, or they can all run on the same machine. In the second case, the instances must use different ports (can be

changed in the options dialog) and must be started from different directories. Furthermore, to allow more than one instance to be run at the same time, eMule has to be executed with the `ignoreinstances` argument (e.g. "`C:\emule\emule.exe ignoreinstances`"). Then, one instance can be set to the hash ID of the attack keyword and the others around it. To achieve this, a search for the attack keyword is started in every instance and the ID is set as own Kad ID respectively. Then, the + and - buttons are used to increase or decrease the ID where needed. After a restart of the Kademlia networks in all instances, the attack is active.
The modified version of eMule additionally features a visualization of every `SEARCH_REQ` and `SEARCH_RES` message, so that the attack can better be observed. Therefore, verbose logging has to be enabled. It can be switched on or off in the options dialog (section *Extended*, *Verbose*, *Enabled* checkbox). With the visualization of the search requests, one can observe what IDs the nodes have, that are asked during a search. This helps debugging the case that an attack would not work correctly. Every time a search response message is displayed in the verbose log, a specific node asked this client for results on the attack keyword and the poisoned results were sent back.

References

[1] The eDonkey2000 Network. http://www.edonkey2000.com.

[2] The eMule project. http://www.emule-project.net.

[3] P. Maymounkov and D. Mazieres. A Peer-to-peer informatic system based on the XOR metric. In *Proceedings of IPTPS*, pages 53–65, March 2002.

[4] Ethereal. http://www.ethereal.com.

[5] Microsoft Visual Studio. http://msdn2.microsoft.com/en-us/vstudio.

[6] eMule Top 50. http://www.emule-top50.com.

[7] GNU General Public License. http://www.gnu.org.

[8] eMule MOD webpage. http://www.emule-mods.de.

[9] Sivka MOD. http://www.emule-web.de/board/10610-emule-0-47a-sivka-v17b1.html.

[10] Changelog of the eMule client. http://sourceforge.net/projects/emule.

Acknowledgements

This report and the modified eMule client are the results of my semester thesis in the Distributed Computing Group of Prof. Dr. Roger Wattenhofer at the Swiss Federal Institute of Technology in Zurich.

I would like to thank my supervisors Thomas Locher and Stefan Schmid for their help and advice. I appreciate the flexibility I had while working on this semester thesis.