ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Master Thesis

eMule Attacks and Measurements

David Mysicka
dmysicka@ethz.ch

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Thomas Locher and Stefan Schmid

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Fall 2007

Abstract

Since the demise of the Overnet network, the Kad network has become not only the most popular but also the only widely used peer-to-peer system based on a distributed hash table. It is likely that its user base will continue to grow in numbers over the next few years as, unlike the eDonkey network, it does not rely on central servers, which tremendously increases scalability, and it is more efficient than unstructured systems such as Gnutella. However, despite its vast popularity, this thesis shows that today's Kad network can be attacked in several ways. The presented attacks could be used either to hamper the correct functioning of the network itself, to censor contents, or to harm other entities in the Internet not participating in the Kad network such as ordinary web servers. While there are simple heuristics to reduce the impact of some of the attacks, we believe that the presented attacks cannot be thwarted easily in any fully decentralized peer-to-peer system without some kind of a centralized certification and verification authority.

Although there are many advantages of decentralized peer-to-peer systems compared to server based networks, most existing file sharing systems still employ a centralized architecture. In order to compare these two paradigms, as a case study, we conduct measurements in the eDonkey and the Kad network—two of the most popular peer-to-peer systems in use today. We re-engineered the eDonkey protocol and integrated two modified servers into the eDonkey network in order to monitor traffic. Additionally, we implemented a Kad client exploiting a design weakness to spy on the traffic at arbitrary locations in the ID space. We study the spacial and temporal distributions of the peers' activities and also examine the searched contents. Finally, we discuss problems related to the collection of such data sets and investigate techniques to verify the representativeness of the measured data.

Contents

1 Introduction

Peer-to-peer (p2p) computing is one of the most intriguing new networking paradigms of the last decade. Not only do structured p2p systems, which typically implement a *distributed hash table* (DHT) [1, 2], possess crucial advantages over centralized systems for applications such as reliable data dissemination, structured p2p systems may also play a pivotal role in the challenging endeavor to redesign the Internet due to their valued properties such as small routing tables, fault tolerance, and scalability. Today's peer-to-peer file sharing applications come in different flavors. On the one hand, there are completely decentralized systems such as the *Kad* network which is based on a distributed hash table where both the task of indexing the content and the content itself is distributed among the peers.[1] Other systems still rely on centralized entities, e.g., a cluster of servers takes care of the data indexing in the *eDonkey* network and so-called trackers handle the peers in *BitTorrent* swarms. A server-based solution has the advantage that it is easier to realize and that it works reliably as long as the servers function correctly. Clearly, the downside of this approach is that the servers can only sustain a certain number of peers, implying that the scalability is limited and that an overload of concurrent requests can easily cause a system failure. Purely decentralized systems do not depend on the availability of any particular entity; however, such systems often demand larger contributions from all participants.

This thesis examines popular representatives of the two network types: the server-based eDonkey and the decentralized Kad network. *eDonkey* is one of the largest p2p networks in use today; millions of users around the planet use it to share various types of multimedia contents. While there are other clients to gain access to the eDonkey network (like MLDonkey[2] or the original but deprecated eDonkey2000 client), by far the most popular client is *eMule*[3]. It is an open source project written in C++, that was founded by some anonymous developers due to missing features in the original client, like a credit system for fair downloading or the use of compression to speed up data transfer. But the most important feature by far is the integration of the *Kad* network, as an additional system for exchanging files and information between participating peers. This network, which is based on *Kademlia* [3], is currently the most popular distributed hash table. Each peer in the Kad network has a 128-bit identifier (ID) which is normally created by a random number generator. This ID is stored at the peer even after it has left the network and is re-used once the peer returns. Routing in the network is performed using these identifiers and the XOR metric, which defines the distance between two identifiers as the bitwise exclusive or (XOR)

[1] Unstructured decentralized systems such as *Gnutella* are not considered in this study.
[2] See http://mldonkey.sourceforge.net/.
[3] See http://www.emule-project.net/.

of these identifiers interpreted as an integer. For all $i \in [0, 127]$, every peer stores the addresses of a few other peers whose distance to its own ID is between 2^i and 2^{i+1}, resulting in a connected network whose diameter is logarithmically bounded in the number of peers. For each of these *contacts* in the routing table, a Kad ID, an IP address, and a port is stored. The publish and retrieval mechanisms work roughly as follows. Each keyword, i.e., a word in a file name, and the file itself, are hashed, and information about the keywords, its associated file, and the address of the owner is published in the network, i.e., this information is stored at the peers in the DHT whose identifers are closest to the respective hash values. If a peer wants to download a file with a certain name (a particular sequence of keywords), it first queries the peer whose identifier is closest to the hash of the *first* of the specified keywords, and this peer returns the information of all files whose file names contain all the given keywords, and also the corresponding file hashes. The requesting peer p_1 can then download the desired file by querying the peer p_2 whose identifier is closest to the file hash, as p_2 keeps track of all the peers in the network that actually own a copy of the file. For detailed information about the implementation of the Kad network refer to [4], where we have analyzed the source code of eMule and reverse-engineered all message types.

In the first part of this thesis we question whether the p2p approach is mature enough to step outside of its "comfort zone" of file sharing and related applications. In particular, not much is known about the ability of DHTs to meet critical security requirements (as those required nowadays, e.g., for domain name servers) and its ability to withstand attacks. To this end, as a case study, we evaluate the feasibility of various attacks in the Kad network, as it is currently the most widely deployed p2p system based on a DHT with more than a million simultaneous users [5]. Our study reveals that while the Kad network functions reliably under normal operation, today's Kad network has several critical vulnerabilities, despite ongoing efforts on the developers' part to prevent fraudulent and destructive use. This thesis describes several protocol exploits which prevent peers from accessing particular files in the system. In order to obstruct access to specific files, file requests can be hijacked, and subsequently, arbitrary information can be returned instead of the actual data. Alternatively, we show that publishing peers can be overwhelmed with bogus information such that pointers to the original files can no longer be accessed. Moreover, it is even possible to *eclipse* certain peers, i.e., to fill up their routing tables with information about malicious peers, which can subsequently intercept all messages. Additionally, we briefly discuss how our network poisoning attacks can also be used to harm machines outside the Kad network, e.g. web servers, by tricking the peers into performing a Kad-steered distributed denial of service (DDoS) attack. It is virtually impossible to determine the true culprit in this scenario, as the peer initiating the attack does not take

part in the attack, which makes this kind of attack appealing to malicious peers. All our attacks have been tested on the real Kad network using the eMule client, which we modified for this purpose. Already with three attackers, virtually no peer in the system was able to find content associated with any given keyword for several hours, which demonstrates that with moderate computational resources, access to any targeted content can be undermined easily.
In the second part of this thesis wich is located in Section 3, we investigate various properties of eDonkey and Kad. Therefore, we have collected large amounts of data from both networks. For this purpose, we reverse-engineered the eDonkey server software and published two own servers which successfully attracted a considerable amount of traffic despite the fact that we never returned any real content. For our Kad tests, we implemented a versatile *Measurement Framework* that is capable of spying on the traffic at any desired position in the ID space. It also permits running periodical searches to be able to analyze changes of published content in the network. Furthermore, the framework can be extended to run arbitrary measurements in the Kad network.
In Section 4, we review related work. After that, ideas for future work and extensions are presented in Section 5. This thesis then concludes in Section 6. During our research, we reverse-engineered many message types. They are all listed in Appendix A. To reproduce our measurements and attacks, Appendix B contains detailed instructions on how to set up and run them.

2 Attacks on the Kad Network

This section presents three different attacks on the Kad network which limit the access to a given file f. In a *node insertion attack*, an attacking peer seeks to attract search requests for f, which are answered with bogus information. Alternatively, access to f can be denied by filling up the index tables of other peers publishing information about f (*publish attack*). Finally, we describe how an attacker can *eclipse* an arbitrary peer: By controlling all the peer's incoming and outgoing traffic, the attacker can prevent a peer from either publishing information about f or from accessing it. A guide on how to run each of the three attacks is available in Appendix B. The success rate of the attacks was computed with the LogAnalyzer, a tool specially developed for the analysis of the log files of the attacks and measurements implemented in this thesis. It will be presented in Section 3.1.3.

2.1 Node Insertion Attack

By performing a *node insertion attack*, it is possible to corrupt the network by spreading polluted information, e.g., about the list of sources, keywords, or comments. We have implemented the attacks for *keywords*, that is, a

search for the attacked keyword will not give the correct results, but instead arbitrary data chosen by the attacker is returned. For this attack to work, we have to ensure that the search requests for the specific keyword are routed to the attacking peer rather than to the peers storing the original information. This can be achieved as follows. In the Kad network, a peer normally creates its ID using a random number generator; however, any alternative mechanism will work as well, as there is no verification of a peer's ID. In our modified eMule client, it is possible to select the peer's Kad ID manually. Thus, an attacker can choose its ID such that it matches the hash value of the targeted keyword. Consequently, the peer will become the node closest to this ID and will receive all the corresponding search requests. The nodes storing the correct files typically have a larger distance to the keyword's ID than the attacker, as the probability for a peer to have a random ID that perfectly matches the 128-bit keyword ID is negligible. In order to guarantee that peers looking for a certain keyword only receive faked results, the attacker must provide enough result tuples, as the eMule client terminates the search after having gathered 300 tuples. The attacker further has to include the keywords received from a peer in the filenames, otherwise the replies are not accepted. In our attacks, we use filenames that contain a unique number, the message "File removed from Kad!", and the keywords. Unique file hashes are needed such that the 300 tuples are not displayed as one tuple in eMule's search window.

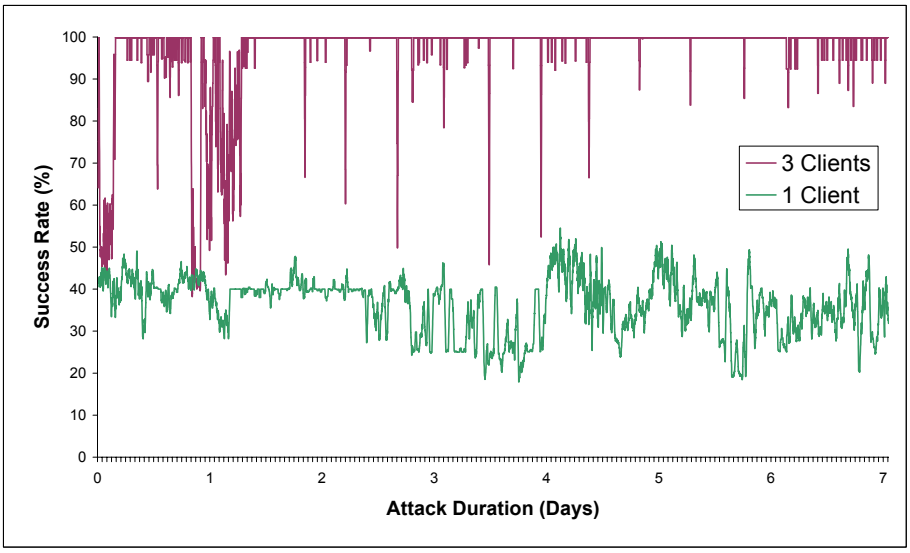


Figure 1: Percentage of successfully hijacked keyword requests in a node insertion attack for 1 and 3 attackers during a time period of one week.

We frequently observed that eMule sends search requests not only to the

closest peer, even though this peer provided enough answers. This can be explained by the delay caused when transmitting the 300 search results from the closest peer. eMule will send another request to the second closest peer before all of the results are received from the closest one. This of course may harm the effectiveness of the attack, and hence it is beneficial to gain control over the second, third, etc. closest IDs as well by means of additional attackers. These attackers behave exactly the same way: All requests are answered by supplying 300 faked tuples. Figure 1 depicts the traces obtained during one week-long node insertion attacks performed using our modified eMule client on the keyword "Simpsons." Note that this attack influences all queries in the entire Kad network not only for the search term "Simpsons", but also all other queries starting with the term "Simpsons" such as "Simpsons Movie" or "Simpsons Soundtrack" etc. are affected automatically. In the first trace, only one attacker whose ID exactly matches the hash of the keyword infiltrated the network. To measure the success of the attack, we used another client, that we again modified to be able to periodically search for an arbitrary keyword and log the results. With this client, we searched for the term "Simpsons" once a minute and counted the number of returned faked results. Since a single attacker is not sufficient, as mentioned before, the attack is moderately successful in that only approximately 40% of the returned results originated from the attacker. What is more, every single query returned at least some results that are not faked. Further experiments showed that using two attackers instead of one does not increase the success rate substantially, but three attackers is already enough to hijack virtually all requests. The second trace shows the success rate of the node insertion attack using three attackers. On average, more than 95% of all returned tuples were faked, and every batch of tuples contained at least some bogus data created by the attackers. The plot shows that there are sudden drops of the success rate once in a while. An explanation for this behavior is that peers join and leave the network at a high rate, resulting in inaccurate routing tables. Consequently, a lookup request can be routed to a peer that still stores results for this request and does not know about our attacking peers yet.

The attack was repeated at other times using different keywords. All our other experiment resulted in a similar picture and confirmed our findings made with the "Simpsons" keyword. Our attacking peers received roughly 8 requests per minute from other peers in the network during the experiments. As expected, the peer having the closest ID received the most requests at a rate of roughly 4 requests per minute.

2.2 Publish Attack

In contrast to the node insertion attack, which forces the search requests to be routed to the attacker, the publish attack directly attacks the peers

closest to the ID of the attacked keyword, comment, or source entry. The index tables stored by the peers in the Kad network have a limited length; for instance, the keyword table can store up to 50,000 entries for a specific ID. Moreover, a peer will never return more than 300 result tuples per request, giving priority to the latest additions to the index table. This makes it possible to replace the original information by filling up the tables of the corresponding peers with poisoned entries. Thus, an attacker seeks to publish a large amount of information on these peers. Once the index tables of the attacked peers are full, they will not accept any other publish requests by other peers anymore. Therefore, the attacked peers will only return our poisoned entries instead of the original information. Since every entry has an expiration time (24 hours for keyword and comment entries, and 5 hours for source entries), the clients have to be re-attacked periodically in order to achieve a constant fraction of poisoned entries. In addition, an attacker has to take into consideration the newly joining peers in the network; if they have an ID close to the one attacked, their tables also have to be filled.

We have implemented the publish attack for keyword entries as well, yet again by modifying the original eMule application. An existing timer method is used to run the attack every 10 minutes. It consists of two phases. In the first phase the 12 peers closest to the targeted ID are located, using eMule's search mechanism. In each run, only peers are selected that have not been attacked before or that need to be re-attacked due to the expiration of the poisoned entries. In the second phase, all the peers found in the first phase are attacked, beginning with the closest peer found. To guarantee a full cache list, 50,000 poisoned entries are sent divided into 250 packets containing 200 entries each. In order to prevent overloading the attacked client, the sending rate was limited to 5 packets per second. Every entry consists of a unique hash value and filename as in the node insertion attack. Since these entries should match all search requests containing the attacked keyword, it is necessary to include all additional relevant keywords (e.g. song titles for an interpreter, year and language for a film title) in the filename; otherwise, all the lookups with additional keywords would not receive the poisoned entries, because not all the keywords are included. In the node insertion attack, this problem does not occur as the additional keywords are obtained from every search request and can directly be appended to the filename to match the request. The success of each run is indicated by the load value sent in every response to a publish packet. This value should increase with every poisoned packet sent, from a starting level of about 10 - 20% to 100% when the attack is finished. To measure the success more precisely, we again use another client to periodically search for the keyword being attacked.

In comparison to the node insertion attack, it is clearly harder to maintain a high success rate using the publish attack, due to the permanent arrivals of new peers and the need to re-attack several peers periodically. While the node insertion attack yields constantly high rates, this is not true for the

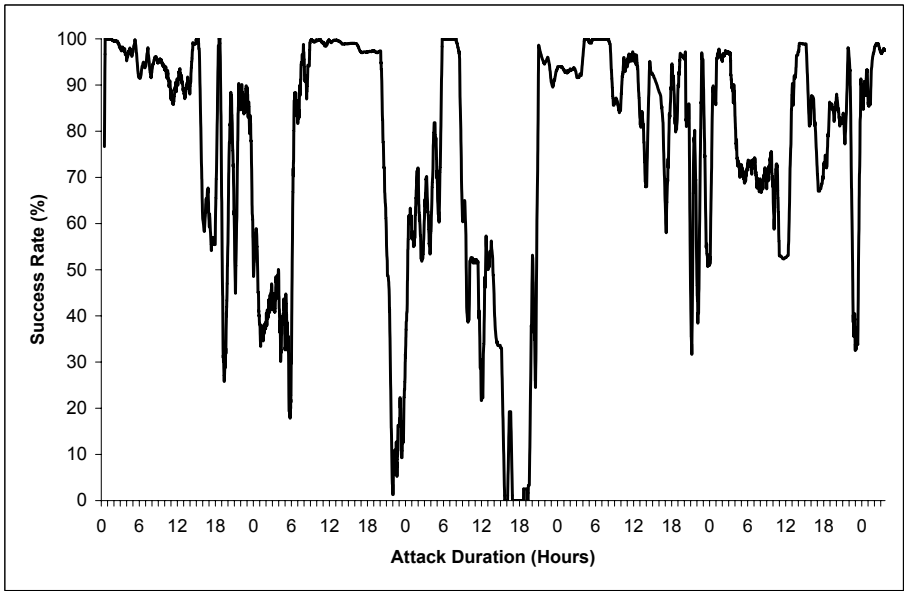


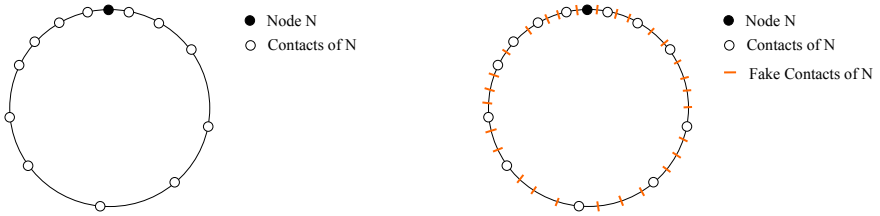
Figure 2: Percentage of faked replies received in a publish attack for the keyword "Simpsons" during a time period of 5 days. Sometimes, the success rate drops but then recovers again quickly.

publish attack. Figure 2 plots the success rate of an attack on the keyword "Simpsons" over a period of 5 days. The success rate was measured with another client periodically searching for the attacked keyword, as described in the node insertion attack. While the attack works fairly well on average, at a success rate of roughly 80%, the success rate periodically drops and remains low for a certain time period before it recovers again. Overall, the success rate is much lower than in the case of a node insertion attack, although performing a publish attack is much more expensive. Again, repeating the attack at other times using different keywords results in a similar pattern. The reason for this peculiar behavior is that the peers responsible for the targeted IDs that are online during the phase where the success rate is low refuse to accept our publish messages. In fact, these peers do not even reply to publish messages, even though they can be contacted, otherwise we could not receive any lookup results from them. This behavior is unusual, because a peer always has to respond to a publish message, even if its index table is full. The only exception where a peer does not have to send a reply, is if it is behind a firewall and therefore cannot be reached by other peers. To eliminate this case, we send special firewall messages to all peers that do not reply during our attack. These messages are normally used to determine whether a joining peer is directly connected or behind a firewall. This improvement increases the number of attackable peers by a fifth, reaching

a level 85%. The remaining part of peers that do not accept our publish messages explain the lower success rate of this attack. As the behavior of this peers is not in accord with the protocol implemented in the real eMule client, we suspect that modified versions of the original clients cause this effect. They seem to be constructed for free riding purposes, without any contribution to this file sharing network. What clients are used is hard to determine as they do not directly provide this information. Thus, the use of modified clients appears to be another reason why the node insertion attack is superior to the publish attack.
In order to improve the success rate a more sophisticated implementation could be used where the attack is split up into two concurrent processes. The first one would permanently search for new peers with an ID close to the one attacked and pass them to the second process which then would attack these peers simultaneously. This would minimize the time during which peers can respond with original data. As this improvement would not solve the situation mentioned before, it was not implemented.

2.3 Eclipse Attack

Instead of poisoning the network to keep peers from obtaining certain information, we can also attack the requesting peers directly and keep them from sending requests into the Kad network. In the eclipse attack, the attacker takes over the targeted peer's routing table such that it is unable to communicate with any other peer in the Kad network except the attacker. As the attacker simulates the whole Kad network for that peer, it can manipulate the attacked peer in arbitrary ways, e.g., it can specify what results are returned for any lookup, or modify comments for any file. The peer's requests can also be directed back into the Kad network, but modified arbitrarily.
Typically, the contacts in the Kad routing table are not uniformly distributed over the whole ID space. Rather, most of the contacts are located around the peer's ID to maintain short lookup paths when searching for other peers in the Kad network (cf. [3]). The attacker takes advantage of the fact that there are relatively few contacts in most parts of the ID space. Concretely, we inject faked peer entries into these parts of the routing table to achieve a dominating position. Subsequently, the faked peers are selected for almost all requests. Figure 3 sketches this process. If we set the IP address of all those faked entries to the address of our attacking peer, we receive most requests of the attacked peer and can process them as desired. We make use of the fact that the standard eMule client accepts multiple neighbors of the same IP address.
Our measurements showed that a peer running eMule for an extended time period has about 900 contacts in its routing table. As the maximum number of contacts is 6,310, there is plenty of space in the routing table for the faked



(a) Before Attack (b) After Attack

Figure 3: Changes to the routing table of the attacked peer during an eclipse attack. The circle represents the ID space.

entries. To inject these faked entries the *Hello Request*[4] message is used, which is normally utilized during connection set up to check whether known peers are still alive. As a side effect of this message, the sender of the message is added to the receiver's routing table. After enough entries are injected, the attacking peer has to process the requests from all those entries in order to keep them in the routing table of the attacked node.
To implement the eclipse attack by modifying the original client would be very complicated, because the requirements for this attack are more complex than for the other ones, as we will see in the following. It is necessary to maintain a list holding all faked entries sent to the attacked peer, because every new entry in the routing table is validated by sending a hello request. As a consequence, we will receive such a request for every entry we have injected. This request has to be answered with the same ID as we have chosen when injecting the entry. In order to differentiate between these requests, we assign a new port to every faked entry and maintain a data structure to store this information. But this implies that we have to listen on all those UDP Ports, to be able to process the requests. Since opening such a large number of sockets is very CPU and memory intensive, we use *WinPcap*[5], a packet capture library for Windows, to bypass the protocol stack and get direct access to all packets received. This, of course, requires a filtering mechanism to pick out only the hello requests from the whole traffic. After that, we generate a response for the requests using the corresponding ID, which we look up in our data structure mentioned before, using the incoming port of the request. To eclipse the attacked peer in the best way, we answer all types of requests, we receive from that peer. If it asks for new peers close to a specific ID, we reply with new faked peers that match this ID, or are very close to it, to guarantee the success of the attack. If the peer

[4]A list of all message types and their format can be found in Appendix A
[5]See http://www.winpcap.org/.

asks for stored information we deliver poisoned results, as in the two attacks discussed before.
As mentioned before, the eclipse attack cannot easily be realized by modifying the eMule client. We therefore implemented it as part of our Measurement Framework, which basically offers the core functionality to access the Kad network, as it contains all necessary parts from the source code of eMule. The Measurement Framework will be discussed in Section 3.1.2.
Our experiments have revealed that the eclipse attack is effective, in particular if only certain keywords are targeted. In that case, the attacker only has to partially fill the routing table of the attacked peer, which renders the attack more efficient. We observed that in this case the success rate virtually always reaches 100% within seconds. In the general case, where the attack strives to eclipse a peer totally and not only for a specific keyword, the success rate would be lower, depending on the distribution and the number of contacts in that peer's routing table. But eventually, the success rate would also reach the 100% mark, at the latest at the point of time when the majority of the contacts has left the network. Naturally, the eclipse attack is limited to merely a single attacked peer. The other two attacks are clearly preferable if an attacker aims at hiding content from *all* peers.

2.4 Discussion

The preceding sections have presented three different attacks that can be used to keep peers from acquiring the requested information. Naturally, these attacks can also be combined in order to increase the chances of a successful attack. However, these poisoning attacks cannot only be used for this purpose. Rather, they can serve an attacker as basic building blocks to pursue completely different aims.
We will now briefly illustrate how they can be used for another attack. The resources of the Kad network's peers and our attacks can be used to drive a *distributed denial of service attack* (DDoS) against any machine internal or external to the Kad network as follows: A node insertion attack is performed in order to occupy some popular keywords. Let μ be the machine (e.g., a server) to be attacked. We inform all requesters that μ contains the desired files. Consequently, all requests are directed to the attacked machine. Of course, the resulting load on μ is not larger than on the machine performing the node insertion. However, the advantage of this attack is that the attacking machine *remains hidden*; moreover, it is generally harder to counter a distributed DoS attack than a normal DoS attack as the request originate from different (and valid) IP addresses. Also the Publish Attack can be used for the DDoS attack if we advertise wrong IP-bindings of keywords. This has the additional advantage that the attack induces more load on the attacked machine than on the attacker, as the different Kad peers are directed to the attacked machine directly. Note that

DDoS attacks using a p2p system such as Kad are particularly nasty as the peers store information about sources for a long period of time, implying that such an attack could last several days with steadily changing peers involuntarily performing the attack.
As all the described attacks can be performed easily and have a large impact, it is mandatory to derive and implement counteractive measures. In order to overcome the node insertion attack it must be guaranteed that choosing specific IDs is infeasible. A straightforward approach, which is often described in literature, is to bind the ID directly to the peers' IP addresses, e.g., by hashing the IP address. However, there are several reasons why real-world p2p systems do not adhere to this simple rule. First, multiple peers may share the same IP address, for example, peers in a local area network behind a NAT router are typically addressed using the same public IP address. These peers would all have the same peer identifier. Second, IP addresses are often given out dynamically and the assignment of addresses may change. In case of an ID-IP binding, this implies that peers have to rebuild their routing tables when reconnecting to the network with a new IP. Additionally, all the credits gathered by uploading data would be lost irretrievably because the peer ID changed and hence the peer cannot be recognized by other peers anymore. It seems that some of these problems can be solved easily and the IP address can still be incorporated into the ID, e.g., by hashing the IP address and a randomly chosen bit string to solve the NAT problem, or by using a different, randomly chosen ID for the credit system, together with a public and private key pair to protect it against misuse.[6] Hashing the IP address and a user-generated bit string is preferable to including the port as this would require a static assignment of ports, and switching ports would also lead to a new ID. However, the crucial observation is that creating such a binding is not sufficient to avert the attack in general, as long as the ID includes a user-generated part. Assuming that a hash function such as SHA-1 is used, an attacker can try out millions of bit string in a short period of time in order to generate an ID that is closest to the targeted keyword even in a network containing more than a million peers. Thus, another form of peer authentication would be required which is hard to achieve without the use of a centralized verification service. As part of the strength of the network is its completely decentralized structure, relying on servers does not seem to be an acceptable solution.
A simple heuristic to render the Kad network more resilient to publish and eclipse attacks is to limit the amount of information a peer accepts from the same IP address, i.e., a peer does not allow that its entire contact list is filled by peers using the same IP address. This is also a critical solution as several peers behind a NAT may indeed have the same public IP address. What is

[6] In fact, Kad already uses public and private keys to authenticate peers whenever a new session starts.

more, an attacker with several IP addresses at its disposal can circumvent this security measure. We conclude that straightforward modifications may lead to an increased robustness, but a powerful attacker can nevertheless launch various effective attacks, and deriving strong disincentives is challenging. Furthermore, a crucial observation is that many of the discussed vulnerabilities do not only pertain to the Kad network, such attacks can be launched against any fully decentralized system that does not incorporate strong verification mechanisms.

3 Measurements

In the second part of this thesis, several measurement results from the eDonkey and the Kad network are presented. We were particularly interested in the user base of both networks. In this thesis, in contrast to other literature, we monitor the actual user requests, namely the *keyword search requests* and ignore automated requests which can also occur without any user intervention (*source requests*). Keyword search requests are sent by eMule whenever the user types in words to search for, whereas source requests are automatically sent by eMule, to periodically retrieve new peers that share the same file, the user is currently downloading. Peers that share a specific file are called *sources* of that file.
Our measurements show that the temporal request distributions of the two networks are very similar, exhibiting a high activity in the early evening with high loads at the eDonkey servers or at the peers hosting popular files in Kad. We also found that both networks are predominantly used in European countries, but there are also many active users from Israel, China, Brazil, and the U.S. This section also investigates the content shared in the two systems. For example, we find that popular content in the eDonkey world is often also popular in Kad and that eDonkey follows the popularity trends of the real world. In general, our results indicate that peer activity results in eDonkey directly carry over to the Kad network and vice versa.[7] Finally, we raise the question of the representativeness of the collected data. In the Kad network, accurate data on the activity of a specific file can be obtained, but due to the distributed nature of the DHT, it is inherently difficult to compute global aggregates such as the most active file in the network. On the other hand, in the eDonkey network, a server receives queries for virtually all keywords, but it has to compete against other servers for the requests. If only a minor fraction of the traffic arrived at our servers or if the servers to be queried were selected with respect to specific properties such as latency, the data could become biased. We will provide evidence that there is no critical bias in our measurements.
This Section is divided into two parts. First, we present the setup of our

[7] This observation is not self-evident, given that we analyze only user-generated events.

measurements for both networks. In the second part, we discuss the results from each measurement.

3.1 Measurement Setup

The eMule client enables access to the classic, server-based eDonkey network and the decentralized Kad network, which is, as mentioned before, an implementation of the distributed hash table Kademlia [3]. The different nature of the two networks requires different measurement techniques. In the following, we will first present our approach to collect data in the eDonkey network. Subsequently, we will report on the functionality of our Measurement Framework which allows us to monitor traffic at arbitrary spots in the ID space of the Kad network. After that, we will provide details on the LogAnalyzer, a analysis tool which we implemented specifically for this thesis.

3.1.1 eDonkey Server

eMule offers 3 different types of lookups, whereof two types search in the eDonkey network. Either only the server which eMule is currently connected to is asked (*local search*), or in the case of a *global search*, all known servers are queried. When a user issues a query using the global search, the keywords of the query are sent to a subset of servers, which subsequently respond to the client with information about where to obtain the requested file. We found that the peers iterate over the list of servers contained in their server file, querying one server after the other as long as less than 300 results have been returned. The order of servers in this list reflects the history of when peers learned about these servers, i.e., old servers are at the top of the list while new servers are appended at the end of the list.
Today, there is a large number of eDonkey servers all over the world, most of which are based on the *lugdunum*[8] software. This software is not open-source as the developers try to prevent the creation of fake servers or any other undesirable modification that could endanger the correct functioning of the lugdunum servers. In order to collect data in the eDonkey network, we reverse-engineered the server software (version 17.13) and implemented our own server application, which we call the *eMule Tracker*. It is written in C#, using Microsoft's Visual Studio 2005. Figure 4 provides a picture of the graphical user interface. A detailed description of all the components of the interface and a guide on setting up the server can be found in Appendix B. In the following we will describe in detail how our tracker operates.
Initially, our tracker imports all known eDonkey servers from a file. We use the same file format as eMule to be able to load prepared server lists, also known as *server.met* files, which are obtainable from various websites.

[8] http://lugdunum2k.free.fr/kiten.html

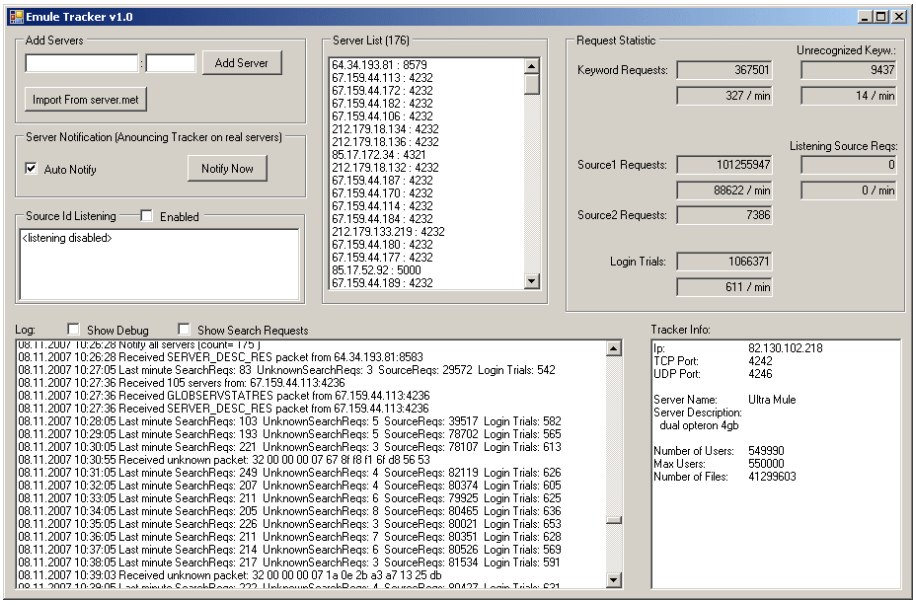


Figure 4: The graphical user interface of the eMule Tracker.

The tracker announces itself to every server on that list, one after the other. By analyzing the communication between two servers, we found that the following messages are sent from a joining server: First, a *server list request* is sent, followed by a *server status request* and a *server description request* (the format of these messages can be found in Appendix A). In return, our tracker receives from each contacted server a list of servers that are alive, and the current status and description of the corresponding server. If there are any new servers, they are added to the server list and also contacted. As a side effect of these queries, our tracker is added to the other server's list. This is vital as peers keep their server lists up to date by periodically asking the servers they are connected to for their lists of currently known servers; i.e., once our tracker appears in these server lists, all peers will quickly learn about the existence of it. In order to remain a member of these lists, our tracker correctly answers the status requests of other servers. In addition, the tracker re-announces itself every hour to all servers on the list. However, due to legal concerns, we neither store nor return any real data. Moreover, we pretend having a high number of users and shared files, but we deny any login requests and reply with a message indicating that our server is full. Since the number of users and shared files is sent in the server status response, any values can be chosen and spread to the other serves and clients. We have set these numbers for our tracker in accord with the largest servers currently present in the eDonkey network. This gives our tracker good chances to become as popular as other servers. To give the

impression that our tracker is fully occupied and cannot serve any additional peers, we set the actual number of users very close to the maximum number of users allowed (another value which is sent in the server status response). As the load on a server usually varies over time, we simulate this behavior through randomly changing our number of users and files every once in a while.

We found that there are several possibilities how to implement a fake server with the purpose of logging search requests. The one that we presented before is simple and yet effective, as we will see when we present our measurement results later on. Another more complex variant would be to allow peers to log on the server and send information about their shared files. The problem which then arises is that the peers logged on the server can now locally search on this server. If we did not respond with real data in this case, the peers would soon discover the fake server. In contrast, this is no problem in our implementation because peers cannot log on to the server and therefore the server is only used for global searches, where many servers are asked and not responding does not stand out, because the search results of all servers are combined. A possible solution for this problem would be to send real results (i.e., matching filenames received from joining peers before) but change the IDs of the files to a random value, so that the peers see real results and can select them for downloading. But these files will never download, because of the random file hash ID for which no other peers sharing a file with exactly this ID will be found. Hence, the peer will believe that there are no other peers sharing this file and will not blame the (faked) server in the first instance. It is difficult to say whether this second variant would lead to more search requests and therefore more data to collect. But it is clear that it has several disadvantages. The biggest disadvantage concerns the resources, as responding with real data would not only drastically increase the bandwidth used by the server but also require far more processing power and memory. Besides, the chances to spot the fake server would be greater, if malicious data is returned than if no data is sent.

Due to the iterative lookup procedure of the global search, which is by far the most popular search, our tracker is contacted continually, regardless of which servers the peers are connected to. As a result, we can collect a large amount of data about many different kinds of requests. To measure the temporal distribution, we count the number of source and keyword requests and the login trials every minute. Additionally, we store all keyword searches including their IP address and time of request as they are the most interesting type of requests. Due to the enormous amount of source requests which we receive, they must be filtered and only requests matching specified IDs are logged. Storing the mentioned requests makes it possible to compute global aggregates such as the most popular keyword in the network, or the most active peer's IP address. Naturally, this data is only representative if

we receive a substantial fraction of all requests in the network. This issue is discussed in more detail in Section 3.2.4.

3.1.2 Measurement Framework for the Kad network

In the Kad network, information about the location of specific files is stored at the participating peers themselves, which all have so-called *overlay IDs*. In order to find a file for a given keyword k, a peer computes a hash function $h(k)$ of k and routes, in a multi-hop manner, the request to the peer P having the overlay ID closest to $h(k)$. This peer P stores the hash codes of all the files associated with this keyword. The matching filenames and the corresponding hash codes of these files are then returned. Given a hash code $h(f)$ of a file f, it is then possible to get a list of all the peers possessing a copy of f by again routing to the peer whose ID is closest to $h(f)$ as this peer is responsible for the sources of f. The hash function used in the Kad network is a modified version of the common but deprecated MD4 algorithm. It was intentionally chosen by the developers to be compatible with the eDonkey network, which also uses this hash function.
We created a Measurement Framework based on the eMule client's algorithms for the Kad network, in order to collect data on the peer activity in the Kad network. Our framework exploits the fact that Kad uses randomly chosen overlay IDs, which enables us to place our peers at any desired place in the ID space. On the one hand, performing measurements in the Kad network is simpler than in the eDonkey network. This is due to the fact that the peer closest to the hash of a file f will be contacted by all peers interested in obtaining this file f. Thus, as there is a unique location where peers obtain information about f, data of good quality can be collected by occupying the corresponding ID and spying on the traffic. On the other hand, the distributed nature of the Kad network renders it more difficult to measure global quantities such as the most popular file in the network. Answering such a query would require to occupy a large portion of the entire ID space. Hence, we confine ourself to acquiring small samples of the entire traffic and try to juxtapose these samples and the data acquired in the eDonkey network in a reasonable manner. In the following we will describe the design of the Measurement Framework with its different components. Moreover, we will report on all measurements that are realizable with this framework and how it can be extended to implement new measurements.
We implemented the Measurement Framework in C# like the other applications in this thesis. Basically, it consists of 5 components: the Kad core, a set of tasks, a task scheduler, a logging component and the graphical user interface. Figure 5 shows the relations between these components. The *Kad core* component implements all functions needed to communicate with the Kad network, i.e. searching for nearest nodes to a given ID, searching for files matching a specific keyword (*keyword search*), searching for peers that

share a specific file (*search for sources*) and searching for comments on a given file. It maintains a routing table, containing other peers of the network, called its *contacts*. The content of the routing table is stored on disk when the framework is closed, so that the contacts are available on the next start. This is crucial as in every peer-to-peer system, because a peer cannot connect to the network without knowing at least one other peer. We have chosen to use the same file format for storing contacts on disk as in eMule[9], to be able to exchange contacts between our framework and eMule. Since the Kad core component is also contained in eMule, we ported the source code to our framework where possible. The only difference to the original code concerns the routing table. While eMule uses a highly unbalanced binary tree that stores far more contacts near the ID of the user (see [4] for an analysis of eMule's routing tree), we did not incorporate any limitations to the binary tree. Consequently we can store more contacts, which can be useful for various measurements.

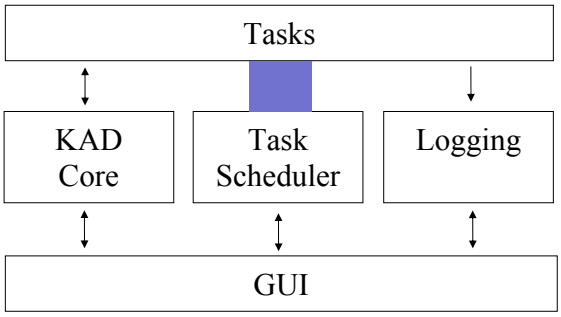


Figure 5: Components of the Measurement Framework.

When measuring temporal changes of the network, a periodically repeating process is needed, that performs a specific measurement. This might be a search operation for a certain keyword or file, or the accumulation of all peers currently present in the network. We have therefore developed the *tasks component*, which consists of freely definable tasks. Each task contains a method `Run()` whose content is repeatedly executed. From this method it is possible to access all needed functions from the Kad core component to implement the desired measurement. Three special tasks are used to maintain the routing tree. This is necessary, as peers join and leave the network at a high rate. To distinguish these tasks from others, they are marked as *internal tasks* in the graphical user interface, whereas the others are called *user tasks*. One of them removes dead peers from the routing tree through periodically asking all peers if they are still alive. The second task searches for new peers to populate the routing tree and to compensate for leaving

[9]Contacts are stored in the file *nodes.dat*.

peers. The third internal task stores all peers currently contained in the routing tree to disk in arbitrary definable intervals. For our measurements we have implemented two user tasks. To be able to monitor peers over time, the *Scan IDs* task creates a snapshot of all peers that can be reached in the network at a specific point of time. For this purpose, it asks the peers for their neighbors, beginning with the smallest ID going up to the largest. The resulting peers of each snapshot are then stored to disk, so that they can be analyzed later. The second user task which we implemented, is used to periodically run *keyword searches* for a defined keyword. It uses the search function from the Kad core and stores the results into a file. The Measurement Framework can also be extended with new tasks. For this purpose, we used a task interface, which is implemented by all tasks. To create a new task a new class has to be created that implements the task interface. Then, the method that is periodically executed has to be written. Finally, the new task has to be added to the list of tasks (refer to Appendix B for detailed instructions).

To manage the tasks explained before, we have designed a *task scheduler*. Every task defines an interval, stating how often it should be executed. The task scheduler computes the next execution time for every task using its interval. As soon as this point of time is reached, the corresponding task is executed. For this purpose, the task scheduler invokes the task's `Run()` method. After this method has finished, the new execution time is computed. To prevent all tasks from being started at the same time when the framework is launched, the first execution time includes a random time shift of at most one minute. All tasks are run in separate threads so that they can be executed in parallel. Shared resources, like the routing table, use locking mechanisms to protect themselves from being accessed by more than one task at the time.

To store measurement results a *logging component* was developed. It offers the possibility to show the results in a log window of the graphical user interface. To be able to analyze these results at a later time, they are also written to a log file. If a measurement produces plenty of data that cannot be shown in the user interface, it can also be written directly to a file. For such occasions, the task can use an own instance of the file logger enabling it to store its data to a separate file. This also simplifies the analysis of that task's data, because it is separated from other measurements. To write a new information to the log, the file logger and the user interface logger offer a method to append a new line to the log. Additionally, the latter also has a method to add debug text to the log, which is not written to disk and not even shown in the user interface if the corresponding option is chosen. It can be used for debugging purposes of newly developed tasks.

The Measurement Framework uses its *graphical user interface component* to inform the user about the state of all the other components. Furthermore, the user can interact with the framework via this user interface. In the upper

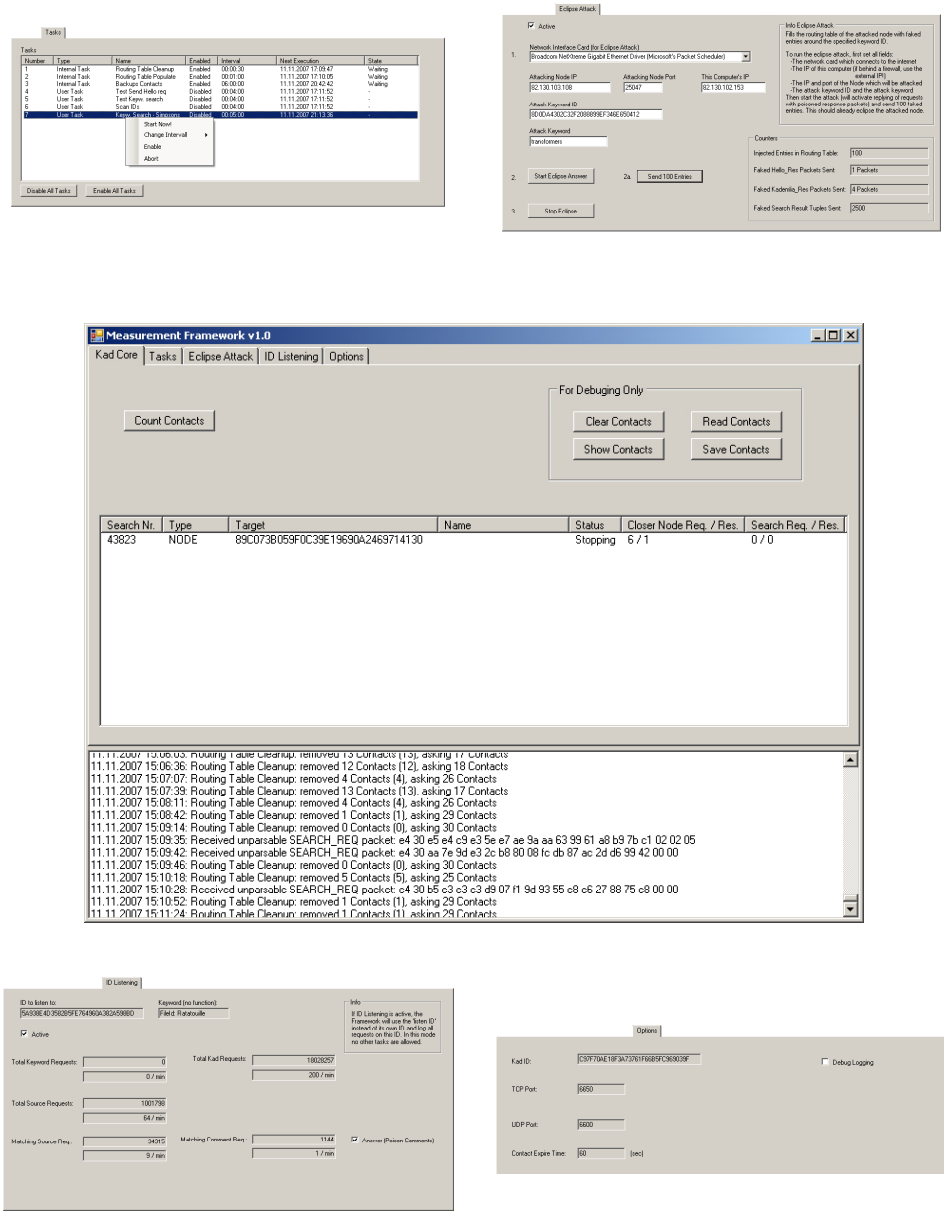


Figure 6: The graphical user interface of the Measurement Framework, including all its tabs.

half, the interface is divided into several tabs, whereof only one tab is visible at the same time. The log window is placed underneath these tabs, so that it is always visible. See Figure 6 for a screenshot of the user interface including all tabs. The first tab shows the Kad core component. It includes several buttons for operations on the contacts in the routing table. They are mainly used for debugging purposes, e.g., for printing out all contacts or writing them to disk. A list is placed beneath these buttons, showing all currently running searches in the Kad network. It displays various information about each search, such as the type and status, the target and the number of requests and responses. The next tab visualizes the tasks. It contains a list that shows properties for each task. This includes the name and type of the task, the interval and the next execution time, as well the current state of the task. A context menu is available upon right click on an arbitrary task. It allows to enable this task, set a new interval and also to immediately start or abort that task. The tasks tab is followed by two tabs that belong to extensions of the Measurement Framework (which will be discussed later on). The first tab controls the eclipse attack. It includes several fields that are needed to configure the attack and buttons to control it. There are also counters, which visualize the state of the attack. The next tab controls the ID listening extension, with several counters to see the state of this extension. The last tab is the options tab. It shows configuration settings of the framework, such as the ID used in the network and the ports to communicate with other peers. Additionally, the user can choose whether to show debug information in the log window or not, by checking the appropriate checkbox.

As already mentioned, the Measurement Framework contains two extensions, which will be explained now. One extension is the *eclipse attack*, whose implementation was already discussed in Section 2.3. Since the Measurement Framework contains all functionality to communicate with the Kad network, it has everything needed for this attack. It was therefore integrated into the framework, instead of creating a new application. Detailed instructions on how to configure and run the eclipse attack can be found in Appendix B. The second extension is the *ID listening* functionality, which allows us to position the framework on an arbitrary ID in the network and to listen to the requests sent to that ID. After the desired ID has been entered into the appropriate field and the listening was activated, the framework will change its ID from the one displayed in the options tab to the entered listen ID. Through periodically checking on the contacts in the routing table, these will spread the new ID into the network and soon all requests concerning this ID or an ID close to it will be routed to the framework. The ID listening extension logs these requests to a file. In particular, the keyword requests, source requests, and comment requests are logged including the IP address of the sender and the timestamp. Several fields visualize the request counts and the request rates (requests per minute). Furthermore, poisoned

comments can be sent to requesting peers. If this option is activated, every comment request is answered with six comment entries, stating that the file is a fake or that the filename does not describe the content of that file. This is done in several languages, to give the impression that different peers have written these comments.

3.1.3 LogAnalyzer

Every attack or measurement in this thesis uses a log file to continuously store the progress or the results. These log files have to be further processed to measure the success of an attack or to extract the data from a measurement that we are interested in. In the majority of cases, a sequence of data is needed which can then be visualized in a chart. For this purpose, we have developed the *LogAnalyzer*, an analysis application that is able to process all the different log files we have generated. In fact, all our measurement results in Section 3.2 and all the success measurements of our attacks were obtained using the LogAnalyzer.

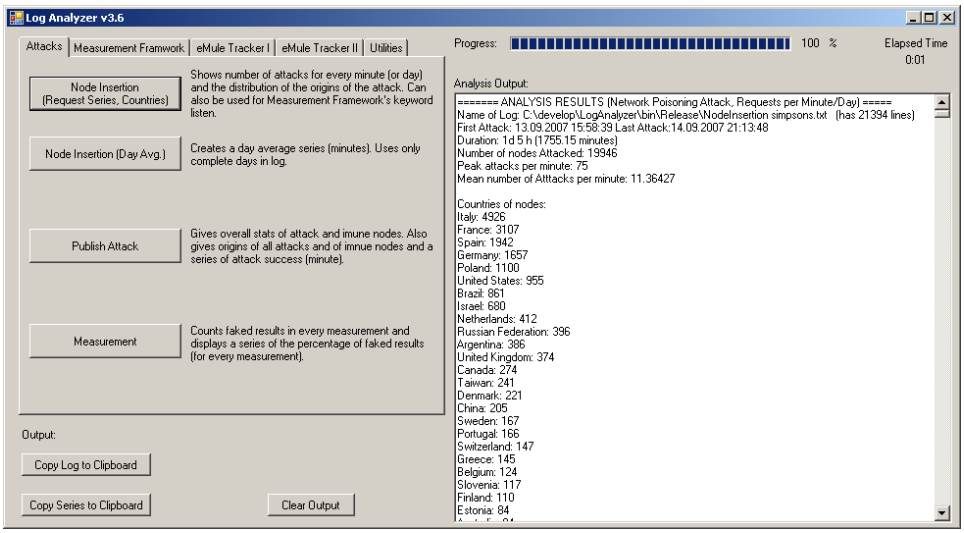
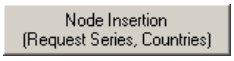


Figure 7: The graphical user interface of the LogAnalyzer.

For every analysis we have implemented, the graphical user interface of the LogAnalyzer provides a button to start it. These had to be organized into several tabs to fit on the screen, as can be seen in Figure 7. Thus, there is a tab for analyses concerning the attacks, the Measurement Framework, and there are two tabs for the eMule Tracker. Finally, there is also a tab for other tools. An output window is placed on the right hand side of the interface. It displays the output after an analysis has finished. Above this window, a progress bar shows the progress of a running analysis and the

elapsed time. This is useful as some analyses need up to several minutes to finish, especially when log files of hundreds of megabytes are processed. For further processing, the output can be copied to clipboard with the appropriate button. It is also possible to copy the series of the output only. This is useful when creating charts of the series. A button to clear the output window is also provided.
Some of the analyses identify the peer's origin from its IP address. To realize this, we integrated MaxMind's GeoIp Country Database[10] into our analyzer. We have also extended it, to be able to categorize not only by country but also by continent. After pressing the button of a specific analysis, first the log file which will be analyzed has to be specified. Some analyses also require additional options which can be chosen in a pop up window shown after the *open log file* dialog. In the following we will describe each analysis of the LogAnalyzer by giving information on what it needs as input data, how it processes the data, and what the output is.



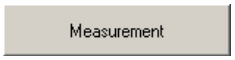
Attacks - Node Insertion (Request Series): This method analyzes the log of the node insertion attack, which contains an entry for every request received, including its timestamp, the IP address and ID of the requesting peer. It counts the requests per minute or day and outputs a sequence of it. In addition, the origins of the requests are evaluated. Only requests that match the specified ID are considered.

Attacks - Node Insertion (Day Average): This function also analyzes the log of the node insertion attack. It computes the course of the requests on an average day, by creating the average over all measured days. The output is a sequence of 1440 requests, one for every minute of the day. To produce this sequence, the file first has to be pre-scanned to determine the measurement period. This period has to be trimmed to a multiple of 24 hours to prevent distortions. Then, the file is scanned again to count the requests. For each request, the minute number of the day is determined and the number of requests of that minute is increased. At the end, every entry in the array holding these 1440 minute requests is divided by the number of days of the measurement period. In addition to the day average of all requests, also separate day averages are calculated for every country.

Attacks - Publish Attack: With this method the log file of the publish attack is analyzed, which contains the IP address, ID, type (Kad1 or Kad2) and version of every attacked peer. For every round of the attack (which is periodically repeated and new peers are attacked) the number of attacked peers is computed and also how many thereof could be attacked successfully. After outputting this sequence, a summery of all rounds is printed, including the percentage of successfully attacked peers and the increase of the success

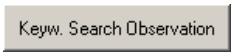
[10] See http://www.maxmind.com/app/geolitecountry/.

originating through the usage of firewall packets, as described in Section 2.2. Additionally, a list of countries of all attacked peers is generated and another list of countries for those peers that could not be attacked.



Attacks - Success Measurement: The modified eMule client for the publish attack also has the feature of periodically running searches for a specified keyword. It is used to measure the success of all the attacks presented in this thesis. The log of these periodic searches contains entries of all peers that returned results to the search for every search round. An entry consists of the IP address and ID of the peer, the number of returned results, and how many thereof are faked. The analysis creates a sequence containing the current search round number and the percentage of faked results obtained in this round.

Measurement Framework - Keyword Search (Top 100): This method reads the log of the keyword search task containing all results received in each search round and computes the 100 most popular keywords found in all filenames. Of course, the first place will always contain the keyword that we searched for as this occurs in every filename. The ranking obtained can be used to pick out interesting keywords whose frequency is then observed, using the following analysis.



Measurement Framework - Keyword Search (Observation): In this analysis the same log is used as in the one before. Now, interesting keywords can be entered and their frequency will be computed for every search round, creating a sequence of them. As the number of search results varies over time, the keyword frequencies can also be computed relative to the first keyword entered (which is in most cases the keyword that was searched for). Plotting these frequencies indicates the changes of the observed keywords over time.

Measurement Framework - Keyword Listen (Day Average): The format of the keyword listening log (from the ID Listening extension of the Measurement Framework) has the same format as the log of the node insertion attack. This way we are able to use the same analysis for the attack and the measurement. This analysis is very similar to the node insertion day average analysis, except that the day average sequences are created for continents and not for countries.

Measurement Framework - Source Listen (Statistics): This analysis reads the general log of the Measurement Framework (*mflog.txt*) and filters out the statistics of the source listening feature. They include the number of source requests, matching[11] source requests, matching[11] comment requests,

[11]The requests that match the ID that we are listening to.

and kademlia requests[12] received per minute and are logged every minute. This analysis prints the sequence of these requests.

Scan IDs Search

Measurement Framework - Scan IDs Search: This method runs through the log generated during an ID scan using the scan IDs task of the Measurement Framework. First, a sequences of distances between neighboring peers is printed (going linearly through the ID space). In addition, a sequence of 1000 items is printed showing the distribution of these distances. If plotted, this sequence corresponds to a histogram with 1000 buckets.

Keyword (Top 500)

Tracker - Top 500 Keywords: In this analysis a list of the 500 most searched words in the eDonkey network is created. Furthermore, a ranking is generated showing the countries with the most requests.

Keyword (Day Avg.)

Tracker - Keyword Requests Day Average (Overall): This method produces an overall distribution of the keyword search requests on an average day. The procedure is similar to the other day average analyses discussed before. Also it produces day averages for the 20 most active countries.

Keyword (Day Avg.) Continents

Tracker - Keyword Requests Day Average (Continents): Similar to the previous analysis. The difference is that the day averages are created for continents and not for countries.

Keyw. Country Activity

Tracker - Keyword Requests (Country Activity): This analysis calculates the activity of countries by cumulating keyword requests per IP address. The more requests per IP address there are on average in a country, the more active it is. First, the ranking of the countries' activities are printed. Furthermore, all IP addresses and their request numbers for the 10 most active countries are written to separate files.

Keywords Observation

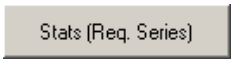
Tracker - Keyword Observation: This analysis computes the frequencies of specified keywords from the keyword search requests. These are printed as sequences either in hour or day intervals. It is possible to define that several keywords have to appear together in a request in order to have a match. This is necessary if one of the keywords alone could be used to search for other objects (e.g., if observing a specific song, the title and the interpreter have to be present in its filename).

First Keyword Counting

Tracker - First Keyword Counting: This analysis counts occurrences of a specified keyword, only if it appears as the first word in the request. This is needed to compare with keyword listening requests in the Kad network, because the Kad keyword searches are hashed using the first word only. If we place our framework to a specific ID, then we will receive all keyword

[12] Such requests are used to find peers close to a specified ID.

requests whose first word's hash matches the specified ID. The occurrences are returned as sequences in hour or day intervals.



Tracker - Statistics: This method filters the debug log of the eMule tracker for entries stating the number of keyword requests, source requests and login trials received per minute. It generates sequences of these three measured values (in minute intervals).

Tracker - Source Listen: The tracker filters the source requests received for three specified IDs and logs the number of requests of these three IDs every minute. This analysis collects these log entries and generates sequences of the request frequencies for the three IDs. This is useful when comparing the eDonkey network with the Kad network regarding the request numbers.

Tracker - Representativeness: This analysis computes the representativeness of the tracker by showing differences between Kad and eDonkey regarding the origins of the requests. For this purpose, a log file of the Measurement Framework and one of the eMule Tracker is compared. IP addresses that occur in both logs are placed into a joint list. If an address only occurs in one of the logs it is placed into a separate list of the corresponding network. After the logs are scanned, the origins of the IP addresses are retrieved, so that we can find out if there are differences between the two networks.

Tracker - Filter for IP: Filters the log of the Tracker for a specified IP address and prints only requests that match this address. This is useful when analyzing whether a specific client has sent some requests to the tracker or not.

Tools - File Merge: This utility merges several files into a single one. For this purpose, all files to merge have to be selected in the file open dialog. The merging order is determined by the alphabetical order of the filenames. This utility comes in very handy, as large log files often have to be split for better handling. Furthermore eMule splits its log files automatically. In our implementations we split them by stopping the application and moving the file into another directory. Then, we restart the application which creates a new file.

3.2 Measurement Results

This section summarizes our measurement results. We investigated the distribution of the user base across countries of both eDonkey and Kad and also the temporal and spacial distribution of the users' requests. In addition, the concrete content that users search in the system is examined.

3.2.1 Request Distributions

Using our eMule Tracker, we set up two fake servers, each running on a dedicated machine. Within a few days after announcing our servers, they attracted much traffic. Figure 8 shows the activity of our servers during 4 days. We see that the request pattern remains fairly stable across all days. On average, during a measurement period of 4 weeks, our servers received roughly 1,550 login requests, 448 keyword requests and 150,228 source requests per minute. The average bandwidth required to run each server is approximately 300 KB/s. Note that a correct server requires substantially more bandwidth as it has to reply to all keyword and source requests. Due to the additional traffic caused by re-announcing our servers at other servers once per hour, our servers are overloaded for a short time resulting in regular drops of handled requests, which is most apparent in the curve of the recorded source requests.

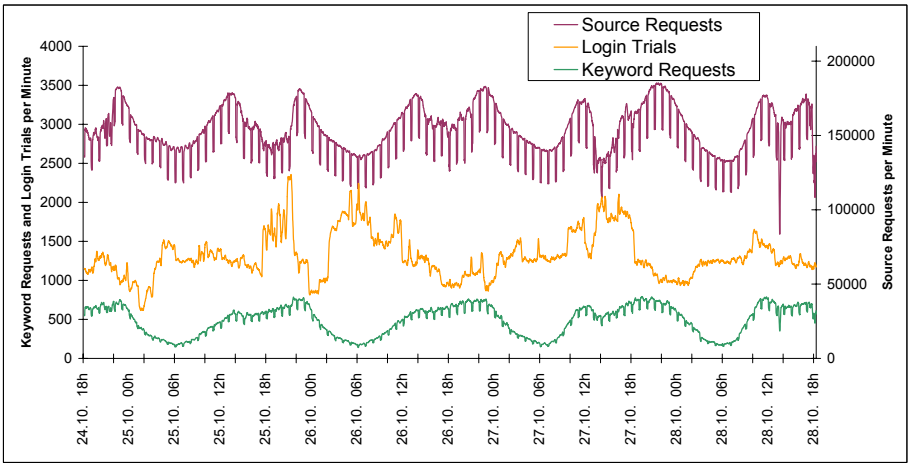


Figure 8: Different server requests over time. The y-axis for the source requests is shown on the right, for the login trials and the keyword requests it is shown on the left.

The keyword searches are particularly interesting to study, as they are entered by users directly and are hardly automated. Consequently, the amount of search requests varies over the day. Figure 9 shows this distribution for different continents. The figure reveals that in Europe and America the minimum number of requests is reached in the early morning and this number continuously increases until midday, where it stays on a more or less constant level during the whole afternoon. Then it increases again after the working hours until the maximum is reached at around midnight. The curve for Asia looks slightly different; the maximum is also reached at midnight, but there is not such a sharp decline during the night, and the number of

requests even increases again reaching a second local maximum in the early morning. Note that the maximum number of requests is set to 100% for each continent in order to show this diurnal pattern. The total number of requests per day in Europe, America, Asia, and Africa plus Middle East are 397,060, 156,322, 42,287, and 48,850, respectively, which necessitates this normalization and also demonstrates the predominance of Europe in the eDonkey network.

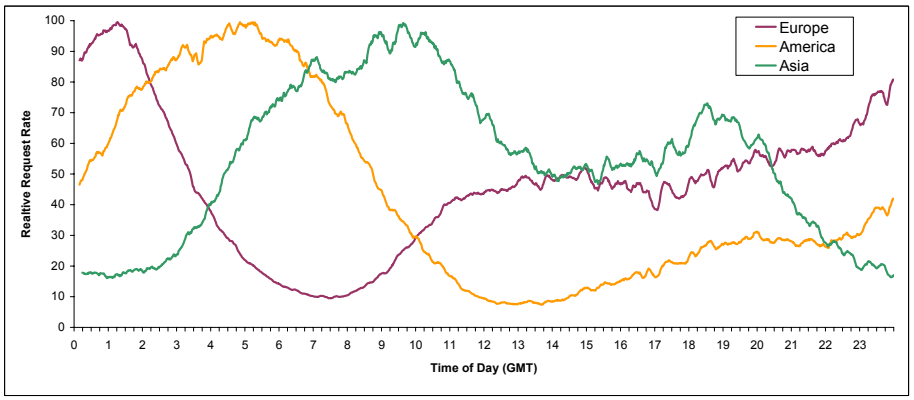


Figure 9: Temporal distribution of keyword search requests on an average day on eDonkey, grouped by continents. The time on the x-axis is based on the Greenwich Mean Time.

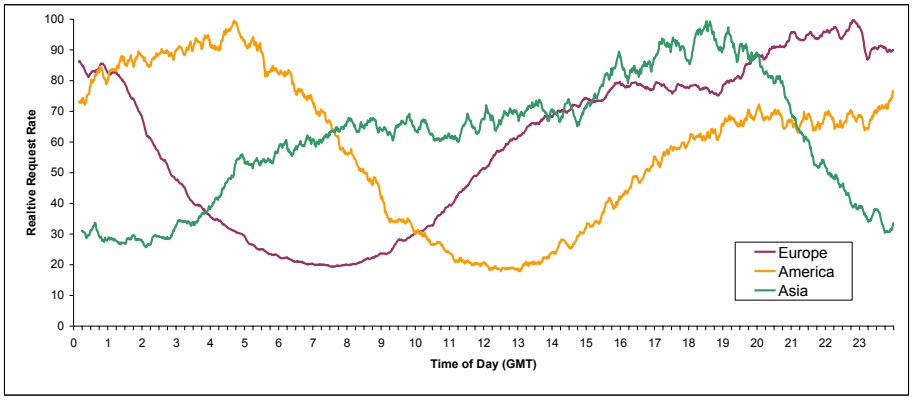


Figure 10: Temporal distribution of keyword search requests on an average day on Kad, grouped by continents. 14 monitoring peers in Kad are used to compute these numbers. The time on the x-axis is again based on the Greenwich Mean Time.

As one might expect, the distribution of the search requests in the Kad network is similar. Figure 10 depicts the temporal distribution of requests

again for the three continents in the Kad network. Again, the curve for Asia is quite different from the others. As opposed to the other continents, the maximum number of requests in Asia is reached in the morning and not late in the evening. We obtained these measurements with the Measurement Framework's ID Listening extension. We occupied 14 randomly chosen IDs and logged all requests on these peers and used the average number of requests in this figure.

We can look at the origins of the requests in more detail and observe that European countries play an important role in eDonkey, the only country among the five most active countries outside of Europe is Brazil. Figure 11 depicts the percentage of all requests originating from each of the 20 most active countries per month, both for the eDonkey and the Kad network in descending order of activity in the eDonkey network. A first observation which can be made is that the spacial distribution is more concentrated in Kad than in eDonkey. Moreover, it can be seen that the same countries are the most active ones in both networks. Note that, although eMule grants access to both networks, users have to enter *manually* where they want to search and thus this result is not self-evident. Furthermore, the Kad network seems to be significantly more used in Europe, especially in Italy and France, than elsewhere. The question whether this is due to a more strict legislation remains open.

It is difficult to assess the popularity of these networks by comparing the absolute number of requests, as there are countries with a much larger population or a higher Internet penetration rate. For this reason, we have normalized the request rates received from each country by the number of Internet users in that country.[13] As can be seen in Figure 12, the picture looks different in the normalized case. There are three quite active countries, Morocco, Algeria, and Israel, while all other countries have a comparably small number of requests per Internet user per month. The reason for this exceedingly high number of request originating from Morocco and Algeria might be simply due to the small number of Internet users in these countries. Another possible reason is that relay servers are positioned in these countries in order to obfuscate network traffic. The observation that a large number of requests originate from a small number of IP addresses supports this claim. As there are many different IP addresses active in Israel and given that it is generally one of the most active countries, it seems that these networks are simply highly popular, even more so than in Europe. As far as the other countries are concerned, the graph shows that there is not a significant difference between the popularity of eDonkey and Kad among them. What is more, the distribution for both networks has a long tail; as many as 21 countries exhibit a normalized search activity of at least 20% of the search activity of Spain, implying that both networks are pop-

[13]Data obtained from http://www.internetworldstats.com.

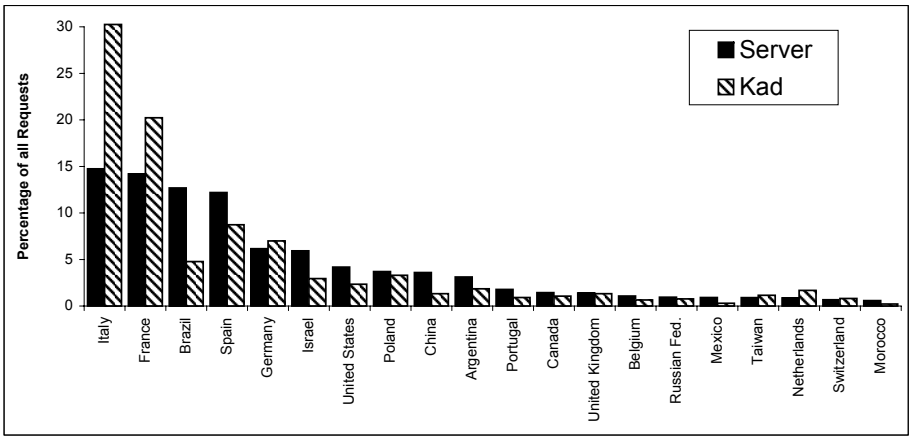


Figure 11: Origins of keyword search requests on our servers and in the Kad network.

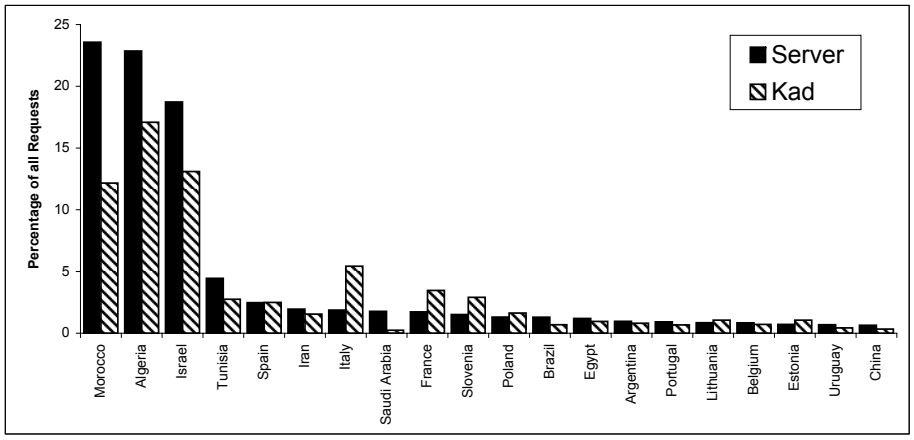


Figure 12: Keyword search requests normalized by the number of Internet users of the 20 most active countries on our servers and in the Kad network.

ular in many countries. We further found that both networks are indeed much more popular in Europe than in the United States, the activity of the United States normalized by the number of Internet users is about 30 times smaller than the activity of Spain, making it the country with almost the smallest activity overall. Clearly, this is partly due to the large number of Internet users in the United States. Overall, only six countries contribute more keyword searches than the United States, which indicates that also in the United States both networks have a large user base. Finally, however, note that the data in Figure 12 could also be slightly biased, as the Internet penetration data might not be perfectly accurate.

3.2.2 Search Contents

The main objective of both the eDonkey and the Kad system is to provide users with a mechanism to find and download files. Information about the searched content can be an interesting source for research, for example, such data might give insights into the potentially different preferences of users in different countries.

For this purpose, a record indicating the popularity of each data item in each country would be required. Unfortunately, the compilation of such a record is quite difficult—not only in Kad, but also in the eDonkey network. One reason is that there is no automatic one-to-one correspondence between keywords and files. There might be different spellings of the same keywords, files containing the same content are typically available in different languages, and the corresponding filenames often contain typing errors. Moreover, the popularity of the files we monitor in Kad can change quickly, particularly when versions of the same content of increased quality appear. Figure 13 plots different versions found when querying for a specific exemplary keyword during a period of 70 days. We used the keyword search task of the Measurement Framework to perform a search every two minutes. During this measurement period we collected over 5 GB of data. Version v_1 is the worst quality, v_2 is the same content in better quality, and v_3 has the best quality. As expected, the number of occurrences of v_1 decreases over time, first at the expense of v_2, and after v_3 becomes more and more popular, the number of occurrences of v_2 start decreasing as well.

Despite the difficulties mentioned before to generate a ranking of the popular data items, we can at least produce a ranking of the most popular keywords searched on our servers. Table 1(a) shows this ranking for the 30 most searched keywords, including the occurrences of a particular keyword per day. Due to the fact that there are many short keywords among the most searched, we present in Table 1(b) only keywords that are longer than three characters.

In another experiment, we tried to evaluate to what extent the popularity of certain content in eDonkey and Kad corresponds to the popularity of the

(a) Unfiltered

Rank	f	Keyword
1.	21037	the
2.	15022	de
3.	12975	la
4.	8849	a
5.	7573	of
6.	6079	i
7.	5562	ita
8.	5508	el
9.	5346	2007
10.	5331	2
11.	5167	fr
12.	5050	in
13.	5021	e
14.	4989	le
15.	4431	you
16.	4303	me
17.	3881	les
18.	3709	3
19.	3512	love
20.	3471	to
21.	3430	and
22.	3108	my
23.	2958	il
24.	2946	dj
25.	2943	los
26.	2901	o
27.	2875	di
28.	2684	del
29.	2601	do
30.	2581	no

(b) Keywords > 3 Characters

Rank	f	Keyword
1.	5346	2007
2.	3512	love
3.	2155	spanish
4.	1934	prison
5.	1927	remix
6.	1880	live
7.	1874	dvdrip
8.	1779	heroes
9.	1699	break
10.	1530	xvid
11.	1488	black
12.	1472	house
13.	1378	album
14.	1371	film
15.	1347	your
16.	1332	amor
17.	1330	french
18.	1319	feat
19.	1284	girl
20.	1219	david
21.	1197	high
22.	1112	karaoke
23.	1104	naruto
24.	1058	dance
25.	1041	star
26.	1016	night
27.	1013	world
28.	1003	girls
29.	1000	life
30.	1000	music

Table 1: Ranking of the most popular keywords searched on our servers. Frequency f in requests per day.

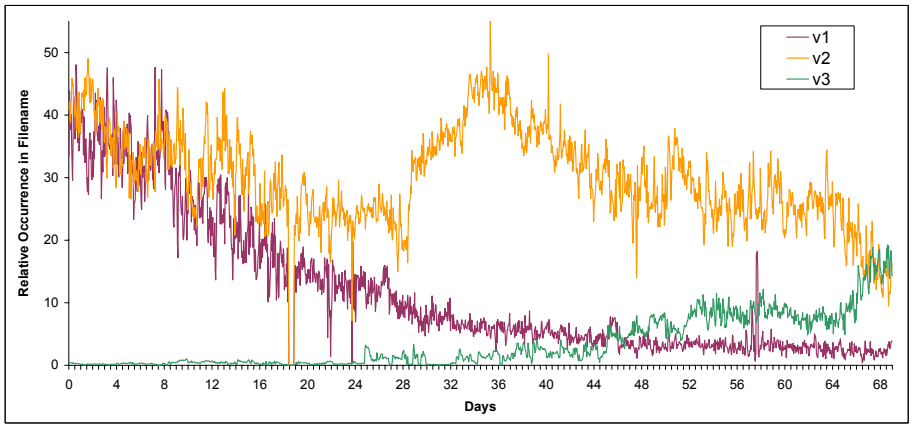


Figure 13: Different quality versions, distinguished by specific keywords in the filename, in percentages of all files.

same content in the real world. To this end, we observed the popularity of newly released movies in eDonkey and Kad. We find that there is indeed a strong correlation, i.e., movies that are currently playing in movie theaters are popular both in eDonkey and Kad. Figure 14 shows this correlation for a specific movie. In this figure, the total gross[14] in the U.S. is depicted for each day and also the number of requests for this movie on our servers. The movie opened on October 5, but it did not attract many movie-goers until the next weekend. Since then, the daily gross is declining again with smaller peaks at the weekends as usual. In this graph, we see that the popularity in eDonkey roughly follows these trends. Observe that the request pattern in the network is delayed for about a week, reaching its maximum about a week after the movie reached its peak. Experiments using other content yielded more or less the same graph, also with a certain delay. In order to take the Kad network into account, we further compared how often keywords are looked up in eDonkey and in Kad and found that basically the same keywords are looked up more often than others in both networks.Our findings all indicate that there is not only a strong correlation between eDonkey and Kad, but also between the two networks and the popularity of content in the real world.

3.2.3 Kad Comments

The idea of realizing a system as it is in the Kad network, that allows peers to rate their shared files and write comments on them seems to be very useful, in particular when there are many faked and corrupted files shared (which is the case at the moment, according to our observations). We were

[14]Data obtained from www.boxofficemojo.com.

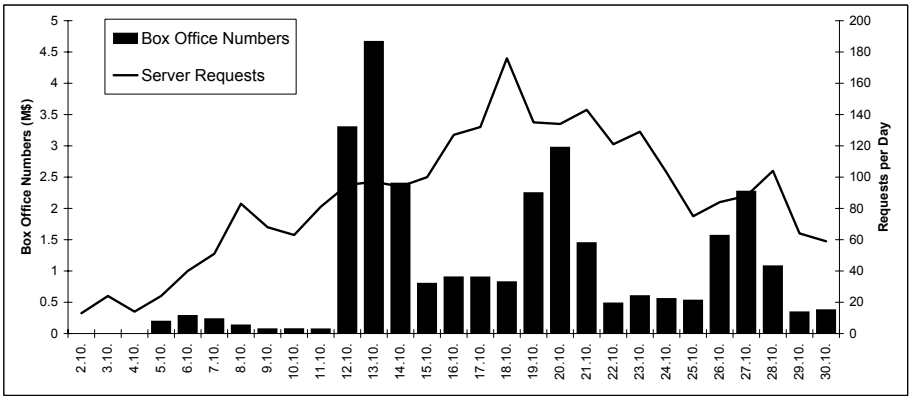


Figure 14: Comparison of the box office gross and the requests on our servers for a specific movie.

therefore interested in finding out by how many peers the comments system is used and what impact it would have, if we spread bad comments on certain files. Hence, we extended the Measurement Framework to listen to specified file IDs and also to listen to comment requests on these IDs. Furthermore we are able to reply with faked comments, stating that the file is corrupt or a fake. Again, as we set the ID of the framework to the file's ID, almost all comment requests are routed the framework and therefore almost all the requesting peers will see only our faked comments. To measure the impact of poisoned comments, we placed instances of our framework onto several popular files and logged the source request amounts. Since these are sent automatically by eMule and not by the user, we can deduce from these requests how many peers currently download the observed files. Figure 15 shows these source request for one of the files. After observing the file for about 9 days, we started to reply with faked comments. This point of time is marked with a vertical red line in the figure. We can see that the request rate drops minimally after faked comments spreading has started, but it is not significant, as the average request rate only drops by 10%. We conclude from these values that the comments system is hardly used. When comparing the source request rate with the comment request rate, our conclusion seems to be confirmed, because for an average of 86 source requests per hour, we only received 3 comment requests.

3.2.4 Representativeness

Conducting measurement studies of distributed systems is a difficult endeavor. Even if large amounts of data is collected, the statistical significance of the empirical results might be limited if the data is biased. In order to obtain solid claims, it is important that the underlying data be either com-

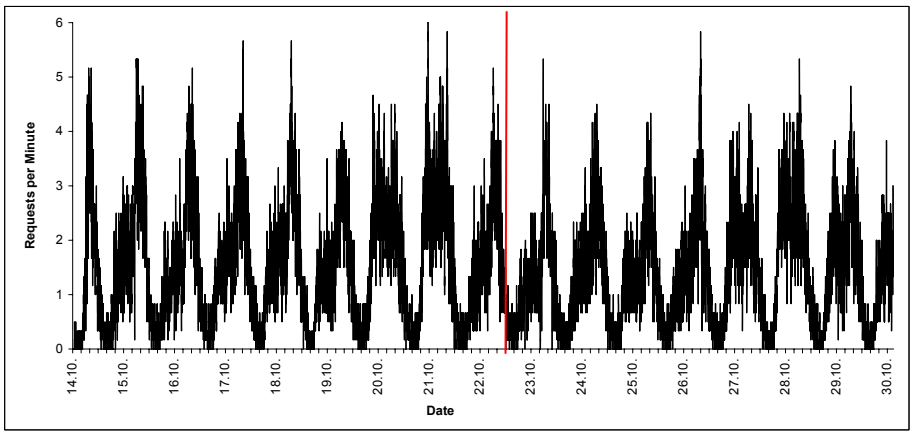


Figure 15: Source Requests for a popular file, before and after faking its comments.

plete, or a uniform and random subset thereof. In this section, we provide evidence that our data can be considered representative.

We consider the data collected by the servers first. As mentioned before, the servers receive requests for all possible keywords. However, since a peer does not send requests to all the servers in its server list, i.e., some servers might receive completely different requests, which could potentially bias the collected data. As the eMule clients typically send source requests to both networks, in order to estimate what fraction of all keyword requests we receive, we compared the number of source requests at our eDonkey servers with the number of source requests obtained in Kad. Our experiments showed that for a given file, we receive roughly 10 times more request in Kad than at the servers. Since virtually all requests for a given file are received in Kad, this indicates that our servers roughly receive 10% of all keyword requests in the network—a surprisingly large number. At the same time, the distribution of the origins of the requests does not differ between the two networks. Furthermore, when comparing the requests of arbitrary keywords in both networks, we can observe that they are very similar. As an example, Figure 16 shows three different keyword request rates for both networks. Note that the server request rates were multiplied by 10, because of the observations mentioned before. This all suggests that our servers are already contacted with a reasonably large probability, although they are relatively new, and also that they get a more or less random subset of the entire traffic.

In the Kad network, it is easy to obtain unbiased request data for a given file, since all requests for a particular file are routed to the same ID. However, making statements about the global distributions of the requests requires to collect data at all locations in the ID space, which is impossible. In

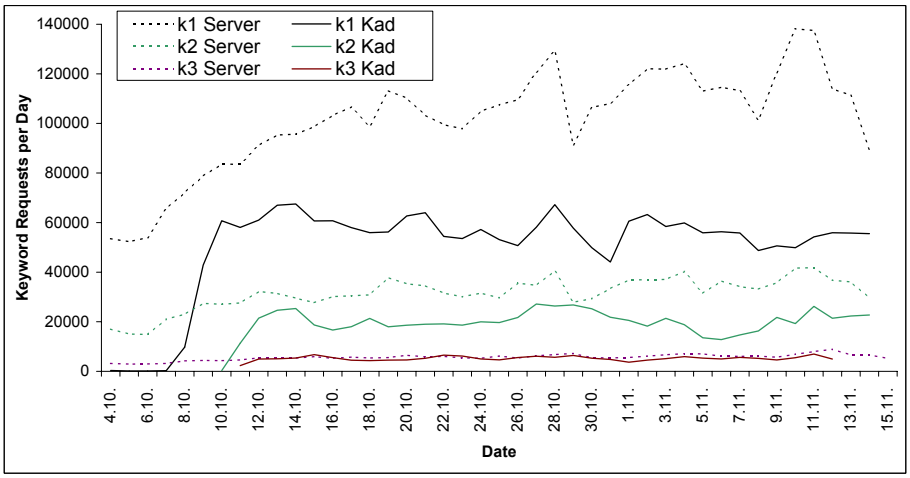


Figure 16: Comparison of the request rates of three different keywords in the eDonkey network and the Kad network (dotted lines).

this thesis, we have taken a best-effort approach and aimed at getting data from a moderately large set of peers whose IDs are distributed uniformly at random. By averaging these measurements, we get similar distributions as those measured in eDonkey, which indicates that the obtained data is fairly representative. Although we believe that the quality of our results is quite good, it has to be taken into account that, similarly to our client, other peers can also choose their overlay IDs at will, which could bias such a random sampling approach. It is known that there are communities that select their Kad IDs from a small subset of the entire ID space [6].

4 Related Work

In the following two sections, we will describe related work for the attacks on the Kad network and for the measurements in eDonkey and the Kad network, respectively.

4.1 Attacks

Peer-to-peer networks have become the most popular medium for bulk data dissemination, and a large fraction of today's Internet traffic is due to p2p file sharing.[15] The immense computational resources of p2p networks are also attractive to attackers, and there is already a large body of literature on the subject [7, 8].[16] Reasons to attack a p2p system can be manifold:

[15]See http://www.cachelogic.com/research/.
[16]See also http://www.prolexic.com/news/20070514-alert.php/.

For example, a peer may seek to perform a more or less passive "rational attack" [9] to be able to benefit from the system without contributing any resources itself [10, 11].
While such selfishness can threaten a peer-to-peer system, which essentially relies on the participant's contributions, there are more malicious attacks seeking to harm the system directly. An attacker may, for example, strive to partition the system or to eclipse individual nodes. The eclipse attack [12], as also described in this work, can be used by a set of malicious peers to position themselves around a given peer in the network such that the peer's contact list consists only of the colluding peers.
In a *Sybil attack* [13], a single entity creates multiple entities of itself in order to gain control over a certain fraction of the system. Such an attack can undermine redundancy mechanisms and is hard to counter in a completely decentralized environment.
Attackers may also exploit a peer-to-peer system to efficiently spread a *worm* [14]. Furthermore, the resources of a p2p system may also be used to attack *any* machine connected to the Internet regardless of whether it is part of the peer-to-peer network or not. A *denial of service attack* can be launched in various p2p systems, e.g., Gnutella [15], Overnet [16], and BitTorrent [17]. During this attack, information about the victim, i.e., the targeted machine in the attack, is spread in the system. The victim is falsely declared as an owner of popular content, causing other peers searching for this content to contact the victim repeatedly. In BitTorrent, tracker information can be faked which leads peers to believe that the victim is a tracker for the desired content [17]. In the Kad network, DoS attacks can be launched by means of a redirection attack where a queried peer, the attacker, will return a response containing the address of the victim [18]. As mentioned before, the attacks presented in this work can also be used to launch a DoS attack.
The work closest in spirit to ours is the study of *index poisoning attacks* in FastTrack and Overnet [19]. Their index poisoning attack is akin to our publish attack where bogus information is pushed aggressively to the nodes responsible for the desired keywords. However, while this attack is also quite successful, it is not as effective in the Kad network as it is in FastTrack and Overnet. We showed that a different, even simpler poisoning attack is feasible and even more effective. Moreover, our study of attacks in the Kad network is not limited to content poisoning and index poisoning, but also considers the eclipse attack to prevent peers from accessing a specific file. It is also worth pointing out that, in comparison to Kad, it is generally easier to perform attacks on Overnet, as it, e.g., does not check whether the sender of a publish message provided its own IP address as the owner of the file, and no cryptography is used for authentication.
While we believe that there are methods to contain the potential damage caused by such attacks to a certain extent, it is known that certain attacks require some sort of logically centralized entity [13]. There is also some inter-

esting theoretical work on identifying and excluding large sets of colluding peers [20]. However, these results cannot be used to counter our attacks as we require only a very small number of attackers close to a given ID, which is not sufficient to raise suspicion. For a more thorough discussion of possible countermeasures against attacks in p2p networks, the reader is referred to the corresponding literature (e.g., [7]).

4.2 Measurements

Measurement studies are an important means to gain deeper insights into the working of distributed systems. While theoretic models allow researchers to reason formally about a system's behavior and to prove its properties, such models are often simplifications and may not reflect reality well. For more complex systems, *in silico* experiments are conducted, desirably for as many points in the parameter space as possible. However, although such simulations—and also experiments on PlanetLab [21]—can provide additional confidence in a system's performance, it is not until the real deployment when the system properties become clear.

There exist many measurement results for various systems today. Saroiu et al. [22] have analyzed several characteristics such as the bottleneck bandwidths of the peers participating in Gnutella and Napster. Adar et al. [23] has investigated the contributions of the Gnutella users. An important algorithmic challenge in p2p computing is understanding churn, and hence traces of membership changes in the systems deployed today [5] have been collected. There is also a community aiming at reverse-enineering closed-source projects such as Skype by studying the traffic patterns [24].

We have decided to study the eDonkey and the Kad networks as they are two of the largest distributed systems in use today, and as there does not exist much literature on these networks. The *Kad network* itself has been the subject of various studies. Stutzbach et al. [25] describe implementation details of Kad in eMule, and [26] presents crawling results on the behavior of Kad peers. The work closest to ours is by Steiner et al. [5]. The authors have crawled the Kad network during several weeks and found e.g. that different classes of participating peers exist inside the network. In contrast to their work which has studied the churn induced by the peers' joins and leaves, our focus is on the peer activity while the peers are *online*, which we measure by monitoring the lookups. As stated in [5], peer IDs can change frequently, even as often as once per download session while other IDs remain in the network for several weeks. Due to these conditions and the fact that several peers might share the same IP address, it is hard to draw any conclusions about peer behavior when monitoring the peer IDs and the IP addresses in the network. Since keyword lookups are hardly automated, observing lookups is the best and presumably the only way to get insights into the activities of users in such networks. To the best of our knowledge, this

is the first peer activity study by means of monitoring lookup requests in distributed networks. It is also the first study to take both server-based and decentralized systems into account.

5 Outlook

During our measurements, we have collected an immense amount of data. We have analyzed this data in many ways and gained a lot of interesting results. Still, there are many analyses that could be done with this data. For example, more research could be invested in the source requests. It would be interesting to solve the problem with the missing one-to-one correspondence between keywords and files. Also, the most searched file IDs could be computed and compared with our keyword rankings. A lot of research should also focus on the question if there are closed sub networks in the Kad network, which disconnect it. We mention this topic, because we believe that we observed this phenomenon during our tests with the attacks. In some cases we never received faked results throughout the entire attack. This only changed if we used the same contacts for both the attacker and the measuring peer.
The three applications we have developed in this thesis are all in a stable release version. Nevertheless, all of them can be extended with new functionality. This applies in particular to the Measurement Framework, which offers the possibility to implement new tasks. Nevertheless, there are some open problems remaining. The two most important concern the Measurement Framework. We did not manage to implement the MD4 algorithm used in eMule, as it is not directly portable. Therefore, the user has to enter the ID of the keyword manually. Another unsolved problem is the scanning of all IDs in the Kad network to gain a snapshot of all peers currently active. Our experimental scan algorithm collects only a small fraction of all peers compared to the results of [5].

6 Conclusion

Structured peer-to-peer systems are likely to gain importance in the near future. This is mainly due to the fact that structured p2p networks have many desirable properties whose usefulness goes far beyond efficient file sharing. Driven by these properties, the use of DHTs or similar structured networks has been proposed as the foundation of the "future Internet" in order to overcome the deficiencies of today's Internet. This thesis has provided evidence that the Kad network, which is currently the only widely deployed p2p network based on a DHT, can be attacked with a small amount of computing resources such that access to popular files is denied. It is clear that such attacks could significantly lower the throughput of the entire system

as the sought-after files are no longer found, and that this imposed censorship would frustrate the users. Moreover, the possibility of leveraging the immense computational resources of the entire system to attack arbitrary machines constitutes a serious threat. We argue that the presented attacks can basically be launched in any peer-to-peer system that does not incorporate sound peer authentication mechanisms. While certain vulnerabilities can be mitigated to a certain extent, more research is needed on how to avert attacks on p2p networks, such as those presented in this work, before its importance can reach the level of the Internet itself.

Understanding the behavior of peers in such large networks might enable the development of new and more efficient distributed algorithms or even pave the way for novel applications in distributed systems. In this thesis, we have compared the peer activity in the server-based eDonkey network with the distributed hash table Kad, two of the largest peer-to-peer networks in use today. We have found that not only do most requests arrive roughly during the same time interval every day in both networks, the searched content is also quite similar. Moreover, by counting the number of source requests we found that our server receives roughly 10% of all eDonkey requests. Using this estimate, and given that we receive virtually all requests for certain keywords in Kad, we conclude that the eDonkey network is still more popular. In total, we estimate the total number of requests in eDonkey to be somewhere between 1.3 and 2 times larger than in Kad. It will be interesting to see how the situation develops in the near future. Furthermore, we conclude that data on the peer activity collected in either eDonkey or Kad can be used as a rough estimate of the behavior in the other network. For the future, we plan to keep collecting data for further experiments.

A Reverse-Engineered Message Formats

This section serves as a reference for the message types used in eMule to communicate in the Kad network and the eDonkey network. Unless noted otherwise, the messages are sent via UDP. A message is either a request or a response. This can be seen from the message name's suffix (the name ends with either `_REQ` or `_RES`). The format of every message that we reverse-engineered is presented in the following subsections. We first list all message types used in the Kademlia 1.0 protocol. This is followed by the message types of the Kademlia 2.0 protocol, which is now being used by eMule since several versions and which should replace the old protocol in the future. The third section presents some of the server message types, used in the eDonkey network to communicate between the eMule client and a server or between two servers. Not all message types were analyzed, because they were not needed to implement the eMule Tacker.
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. Kad messages all start with the same header of 1 Byte length. It contains the value 0xE4[17], which is referred as `OP_KADEMLIAHEADER` in the source code. Longer messages can also be sent compressed using the *zlib* compression format. The header's value is than changed to 0xE5. eDonkey messages use the header 0xE3. For reasons of space, the 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 number of times. These fields are highlighted blue in the figures.

A.1 The Kademlia 1.0 Protocol

Hello

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

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

[17] Hexadecimal representation

It contains exactly the same fields as the first message.

Bootstrap

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

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

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

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

Search

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

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

Either 2, 4 or 11 peers can be requested. 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 [1]	Target ID [16]	NumOfPeers [1]	Kad ID [16]	IP [4]	UDP [2]	TCP [2]	Type [1]

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

The actual search request for sources or keywords 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 (See the explanations for the KAD2_SEARCH_KEY_REQ message in the Kademlia 2.0 protocol). The first keyword is never sent in text format.
If a peer 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 peer that is a possible source for the requested file. The result ID corresponds to a peer's ID and the tag list contains the IP address, the TCP and UDP port of that peer. 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 comments, 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 comments 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 peer that created the comment and

a list that contains the peer's name, the file name, rating and comment. The number of tuples that are sent in the response is defined in the third field.

Publish

If a peer 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 list with the filename and the file size.
If the peer 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 always the sender itself). Therefore, the third field contains a 1. The tag list contains the filename, the 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 (indicating how full this map is).

A comment 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 peer that is publishing the comment. The tag list contains the peer's name, the filename, the file rating and the comment.
If a peer could successfully store the comment, 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 map containing the comments is (from 1 to 100, in percent). In contrast to the keyword publish, the load value is not processed.

NAT and Firewall

If a peer needs to obtain its public IP Address, it sends a `FIREWALLED_REQ` message, 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 peer that sent the appropriate request message.

To check whether a peer is behind a firewall or not, a `FIREWALLED_ACK_RES` message is sent to every new peer it receives. It does not contain any information (apart from the message type):

0x59
[1]

If a peer is behind a firewall it cannot be contacted directly. Therefore, this peer needs the help of another peer in the system which is directly reachable. Such a helping peer is called the *buddy* of a firewalled peer. To find a buddy, a `FINDBUDDY_REQ` message is sent in the second phase of the search procedure (instead of sending a `SEARCH_REQ`). The format is:

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

The buddy ID is the target of this search. It is the inverted ID of the sender. The third field contains the senders hash, which is not equal to its Kad ID. The forth field holds the senders TCP port.
If a peer receives such a buddy request and accepts to be the buddy of the sender, it replies with a `FINDBUDDY_RES` message:

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

It has the same format as the request. The buddy ID is identical, but the peer hash and the TCP port are the ones from the buddy. The buddy's peer 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 peer 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 peer. The third field holds the file hash ID of the requested file (either a download request from the firewalled peer or a notification that the firewalled peer 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 peer and the sender to bypass the firewall.

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.

A.2 The Kademlia 2.0 Protocol

This section lists the most important message types used in the Kademlia 2.0 protocol. There are additional message types (in fact, the concept is very similar to the Kademlia 1.0 protocol), but as they were not needed in our implementations, we did not analyze them.

Hello

To test if a contact is alive, the `KAD2_HELLO_REQ` is used:

0x11 [1]	Kad ID [16]	TCP Port [2]	Version [1]	0x00 [1]

The fields contain information about the sender. The version field provides the version number of the eMule client a peer is running. The following table lists possible entries for this field:

Number	Version
0	unknown
1	0.46c
2	0.47a
3	0.47b
4	0.47c
5	0.48a

The response to a hello request is the `KAD2_HELLO_RES` message. It has the same fields as the request:

0x19 [1]	Kad ID [16]	TCP Port [2]	Version [1]	0x00 [1]

Search

The `KAD2_KADEMLIA_REQ` message is used instead of the `KADEMLIA_REQ` message, to obtain closer peers. The format remains the same.
The reply is a `KAD2_KADEMLIA_RES` message with following format:

0x29 [1]	Target ID [16]	NumOfPeers [1]	Kad ID [16]	IP [4]	UDP [2]	TCP [2]	Version [1]

The only difference to the old format is the version field in the peer list. It replaces the type field and provides eMule's version number.

In the Kademlia 2.0 protocol the keyword request was split from the source request. These two requests now have their own message type. To search for sources, a KAD2_SEARCH_SOURCE_REQ is sent, having this format:

0x34	File Hash ID	Start Position	File Size
[1]	[16]	[2]	[8]

The hash ID and the files size allow to uniquely map to a specific file. The start position allows a peer to request for more than 300 entries (by default only the first 300 are returned). It holds the position from which the responding peer should start listing the results.
To search for keywords, the KAD2_SEARCH_KEY_REQ is used:

0x33	Keyword Hash ID	ST Flag	Search Tree
[1]	[16]	[2]	[*arbitrary*]

If only one keyword was entered in the search dialog, it is hashed and written to the keyword hash ID field. In this case, there is no search tree and the search tree flag is set to 0x0000. If there are more than one keywords, the first one is hashed and written to the keyword hash ID field. The remaining keywords are then stored in the search tree. The latter can also be used to express boolean search expressions, including restrictions using meta tags like file type, availability etc. Note that the first keyword is never sent in text format and therefore not included in the search tree. The search tree flag is set to 0x8000 if a tree is sent in the message. The search tree uses the following format:

```
<search tree> := <operation> | <string> | <metatag> |
                 <numeric relation>
<operation>   := <and> | <or> | <not>
<and>         := <search tree> | <search tree>
<or>          := <search tree> | <search tree>
<not>         := <search tree> | <search tree>
<string>      := strSize[2] strData[strSize]
```

The `<metatag>` and `<numeric relation>` expressions were not analyzed. All expressions are encoded using a one byte code which can be obtained from the following table:

Expression	Value
`<operation>`	0x00
`<string>`	0x01
`<metatag>`	0x02
`<numeric relation>`	0x03 or 0x08
`<and>`	0x00
`<or>`	0x01
`<not>`	0x02

When searching for comments on a file, a KAD2_SEARCH_NOTES_REQ is used:

0x35	File Hash ID	File Size
[1]	[16]	[8]

It contains the hash ID and size of the file for which comments are being searched.

The response to a source, keyword or comment request is a KAD2_SEARCH_RES message, with the following format:

0x3B	Sender's ID	Item Hash ID	NumOfResults	Result ID	Tag List
[1]	[16]	[16]	[2]	[16]	[*arbitrary*]

The sender's Kad Id is contained in the field after the message type field. The item hash ID is either a file hash ID (for source and comment responses) or a keyword hash ID. The results are sent as tuples of a result ID and a tag list, repeated as many times as specified in the forth field. In case of a sources search, the tuples are peers that share a specific file. The result ID corresponds to a peers ID and the tag list contains the IP address, the TCP and UDP port of that peer. 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. In case of a comment search, the tuples are comments. The result ID is the Kad ID of the peer that created the comment and the tag list contains the file name, a rating and the comment. The message can be split into several packets, if they are to big.

A.3 The eDonkey Server Protocol

This protocol is used by eDonkey servers to exchange information among themselves and by eMule to send search requests to servers. As there is no source code of the server application, we had to guess the meaning of the message contents. In some cases, we could not decode parts of a message. Nevertheless, it is necessary to send such messages when faking a server. Therefore we filled undecodable fields with the same data that we received while reverse-engineering these messages (through capturing packets from the real lugdunum server application).

Server Announcement

Servers announce themselves to other servers when they join the network. They also periodically check whether each one of their known servers are still running. This announce procedure consists of three request messages, followed by corresponding responses. The first message sent in this procedure is the SERVER_LIST_REQ with the following format:

0xA0 [1]	Sender's IP [4]	Sender's TCP Port [2]	*Unknown* [4]

This message requests the receiver to send a list back with all the server it knows. The fields contain information about the announcing server. The last field was not decodable. It has the value 0x479DD533.
The answer is a SERVER_LIST_RES message containing a list of servers:

0xA1 [1]	NumOfServers [1]	IP Address [4]	TCP Port [2]

The second field defines how many servers are contained in the message. A server consists of its IP address and TCP port.
The second message sent in the announce procedure is a GLOBSERVSTATREQ message with the following fields:

0x96 [1]	Challenge 1 [4]

It requests the receiver to send information about its status. A challenge is used to confirm the identity of the reply message. As its value is always 0x55AA0001, this challenge does not contribute to increase the security.
The response is a GLOBSERVSTATRES message having this format:

0x97 [1]	Challenge 1 [4]	NumOfUsers [4]	NumOfFiles [4]	MaxUsers [4]

It contains the challenge from the request and 3 additional fields with capacity information about the server (Number of users currently on this server, sum of files shared by all these users and the maximum number of users allowed on the server).
The third announce message is the SERVER_DESC_REQ message, having this format:

0xA2 [1]	Challenge 2A [4]	Challenge 2B [4]

With this message the receiver is asked to supply its description. There are two challenges included in the message.

The answer is a SERVER_DESC_RES message with this fields:

0xA3	Challenge 2A	*Unk.*	Server Name	*Unk.*	Server Desc.	*Unk.*	Challenge 2B
[1]	[4]	[8]	[*arbitrary*]	[4]	[*arbitrary*]	[8]	[4]

The description message contains the server name and description. Note that strings always have a preceding two bytes long field indicating the length of the string (not shown in the figures). There are several parts that we could not decode. These should contain, among other things, the server version and its features like the support of large files etc. Also, the two challenges from the request can be found in the message.

Search

If eMule searches on servers with the *global* method, it uses GLOBSEARCHREQ messages with the following format:

0x98	Search Tree
[1]	[*arbitrary*]

All keywords are contained in the search tree. It has exactly the same format as in the Kademlia 2.0 protocol (See the explanations for the KAD2_SEARCH_KEY_REQ message on page 49). There also exist two other message types for global keyword searches (the GLOBSEARCHREQ2 type with the same format as the first type and the GLOBSEARCHREQ3 type with another format), but they were hardly received by our tracker.

For the global search of sources for a specific file the GLOBGETSOURCES message is used:

0x9A	File Hash ID
[1]	[16]

It just contains the hash ID of the searched file. In addition, the GLOBGETSOURCES2 type can be used for the same purpose. It contains an additional field with the file size. We measured that it is used 20,000 times less than the first type.

server.met

The list of servers is stored by eMule in the *server.met* file. The format of this file is compatible with the lugdunum server application and it is also used by our eMule Tracker. It has the following structure:

0xE0	NumOfServers	IP Address	TCP Port	Tag List
[1]	[4]	[4]	[2]	[*arbitrary*]

After a header and the field holding the number of servers contained in this file, the servers are listed with tuples of their IP address, TCP port and a list of tags. The latter contains, among other things, the server name and its description.

B User Guides

We have presented several attacks and many measurements in this thesis. This section gives detailed instructions on how to set up and run each of them. First, guides for the attacks are provided, followed by instructions for running a fake server with the eMule Tracker. After that, setting up the measurements is explained. All applications used in this thesis were running on Microsoft's Windows XP operating system. The attacking or measuring computers should not be located behind a firewall or NAT (network address translation) system. Furthermore, the eMule Tracker, the Measurement Framework, and the LogAnalyzer need Microsoft's *.NET Framework Version 2.0*[18]

B.1 Attacks

This section describes the steps for setting up the three attacks. In the first two attacks, we use modified versions of the eMule clients. To be able to process their log files later, *verbose logging* has to be activated and set to *save the log to disk* (this can be done in the options dialog).

Node Insertion Attack

To run the node insertion attack a special version of eMule is needed, that we have modified for this purpose. It is based on the original eMule client in version 0.47a. Like mentioned in the Section 2.1, several clients are needed to successfully run this attack. The best way is to set up each client on a different machine, but it is also possible to run more then one client per machine. In the latter 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 using the appropriate button (see Figure 17). 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 success of the attack is measured with another modified eMule client, which is also used for the publish attack. This client has to be started and connected to the Kad network. Then, *periodic searches* can be activated in the servers tab of the graphical user interface. Which keyword is searched

[18] See http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5.

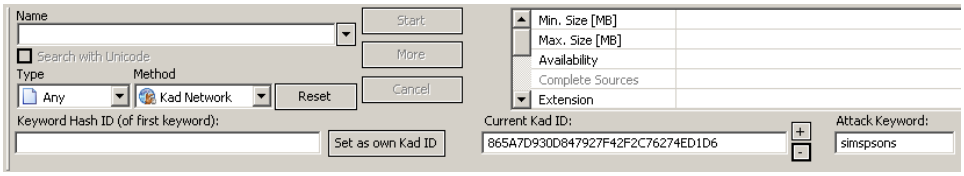


Figure 17: The graphical user interface of the modified eMule client, used to run the node insertion attack. Only the modified part is shown.

and what interval is used, has to be manually defined in the file called *dave.dat*. It is located in the config subdirectory of eMule. It contains settings for all the additional features we implemented. If opened with a hex editor the search keyword can be found at the end of the file. It is terminated with 0x0D0A like every other string in this file. The keyword can be replaced with any other keyword. The interval is specified directly before the search keyword, using an unsigned one byte integer. The number represents the interval in seconds. We recommend to use an interval of 60 seconds.
The results of the periodic search measuring the attack is logged in the verbose log called *eMule_Verbose.log*. This file is split every time it reaches a size of one MB. Therefore it has to be merged before it can be analyzed. This is done with the LogAnalyzer, which offers a function for merging files (See Section 3.1.3). Then, the merged file is analyzed with the *success measurement* method found in the attacks tab of the LogAnalyzer.

Publish Attack

As mentioned in the instructions of the previous attack, the publish attack also uses a modified version of the eMule client. Unlike the previous modification, this version is based on the original version 0.48a. After it is started and connected to the Kad network, the attack can be set up. For this purpose, the servers tab of the graphical user interface was extended with several controls (see Figure 18). To run the publish attack on a keyword, the appropriate radio button is selected. Then, the main keyword which should be attacked can be entered in the first field. In some cases additional keywords have to be specified (see Section 2.2 for more details). This can be done in the field beneath. To save the changes the set button has to be pressed. This automatically hashes the attacked keyword and shows its ID in the corresponding field. The values of this fields are stored in the file called *dave.dat* so that the user does not have to retype them after a restart. The user interface also contains a debug box, where the user can specify whether to send attacking packets or to just simulate the attack. Additionally, a fast mode can be activated which speeds up the steps of

the attack (this mode is mainly for demonstration purposes). To start the publish attack, the start button has to pressed.

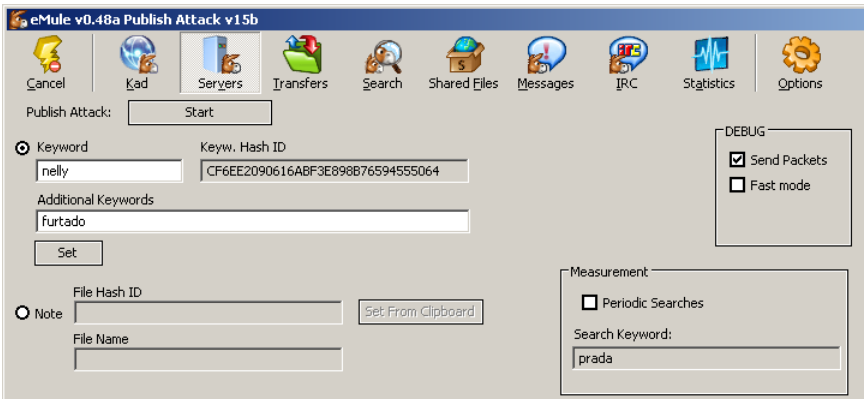


Figure 18: The graphical user interface of the modified eMule client, used to run the publish attack. Only the modified part is shown.

It is also possible to attack the comment entries instead of keyword entries. But this mode is not fully developed, as there were problems injecting more than one comment from the same IP address. In this mode, the hash ID of the file whose comments will be poisoned is needed. It is imported via the corresponding button which parses an eDonkey link from the clipboard. Such a link can be obtained in the search tab through right clicking on a search result.
The success of the attack is again measured using another running instance of this modified version of eMule. Refer to the instructions of the previous attack on how to set up the measuring client.

Eclipse Attack

The eclipse attack is performed using the Measurement Framework (as described in Section 2.3). Before using this framework to run the attack the *Windows Packet Capture Library* (WinPCAP) has to be installed[19]. We used version 4.0.1 for our tests. After starting the application, the eclipse attack extension first has to be activated using the corresponding check box in the eclipse attack tab of the user interface (see Figure 19). Then, all fields have to be filled out with the necessary information. This information is stored when the application is closed, so that it has not to be retyped on the next start.
When pushing the start button, the framework begins to listen to requests from the specified attack IP address and answers them with faked peers. To

[19]See http://www.winpcap.org/.

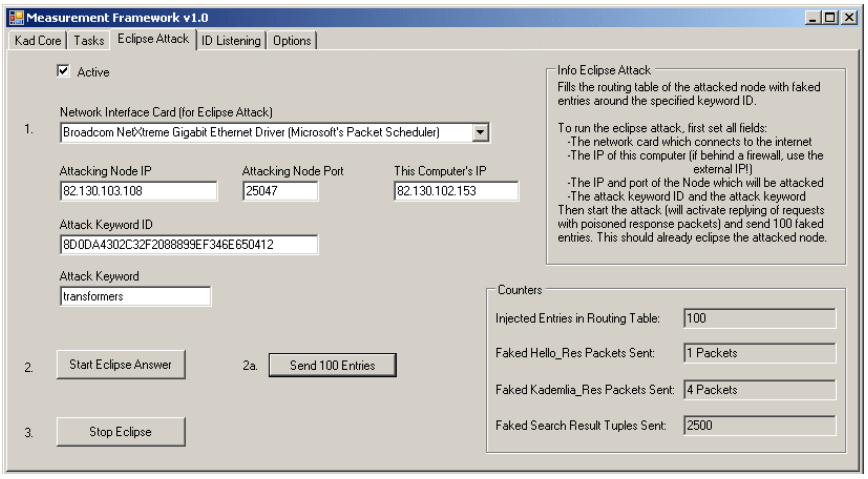


Figure 19: The graphical user interface of the Measurement Framework, showing the eclipse attack tab.

inject faked peer entries into the routing table of the peer being attacked, the appropriate button has to be pressed. The attack is stopped with the stop button. On the right side of the user interface there is a box with the instructions on how to run the attack. Beneath this box the counters box shows the actual state of the attack.

B.2 eMule Tracker

This section describes how to set up a faked server using the eMule Tracker that will spread itself in the eDonkey network and listen to various requests (see Section 3.1.1 for detailed information on the implementation of the eMule Tracker and for a screenshot of the graphical user interface). Before the tracker can be started, a few preparations are necessary. We experienced problems with personal firewalls installed on the system the tracker was intended to run on. If such a firewall is installed it should not only be deactivated but also uninstalled. Then, the configuration file of the tracker has to be adapted to the actual setting. Therefore the file called *EmuleTracker.exe.config* can be opened with a text editor. It contains settings such as the IP address and TCP port of the server, its name and description, and the faked entries for the actual number of users and shared files. Additionally, there is the possibility of filtering the source requests for three defined IDs which are then logged in a file. These IDs can also be specified in the configuration file. This has to be done *before* starting the tracker.
As soon as the eMule Tracker is started, it will log all received requests. To

announce it on other servers a list of all currently active servers has to be imported. This can be done with the corresponding button, which parses a *server.met* file that can be obtained from various web sites or from the eMule directory. Another possibility is to add the servers manually by typing in their IP address and TCP port into the corresponding fields and pressing the button to add a server. To activate the automatic announcement procedure, the corresponding check box (*Auto Notify*) has to be active. Then, this procedure is repeated every hour. It can also be started directly with the *Notify Now* button. If the tracker should also log source requests for the three specified IDs, the appropriate check box has to be activated. The tracker uses three different log files to separate the information being logged. The *tracker.log* is the main log file. All error messages that occur are written to it. The content of this log is also printed into the log window of the user interface. In addition, every minute an entry is written containing the number of search requests, unknown search requests (not parsable requests), source requests, and login trials that were received that minute. These values can be extracted from this log using the appropriate function of the LogAnalyzer. The keyword search requests are logged in the file called *searchResults.log*. This file grows very fast und reaches several hundred of megabytes after only few days. There are many analyses contained in the LogAnalyzer that process this log file. Source requests of the three specified IDs are written to the file named *sourceRequests.log*. The LogAnalyzer can also analyze this log file.

B.3 Kad Measurements

All our measurements of the Kad network can be obtained using the Measurement Framework (see Section 3.1.2 for more details). It can also be used to run the eclipse attack, as mentioned earlier. Before a measurement can be started the framework has to be set up. This includes adjusting its configuration file *MeasurementFramework.exe.config*, which can be opened with a text editor. It is not necessary to change the fields in the *userSettings* part, as these can be changed in the framework itself. Important settings are the ports (UDP and TCP), which should not be occupied by other applications including other instances of the framework. Other important settings concern the keyword search task and the ID listening extension and will be discussed later. The framework needs other peers to be able to connect to the the Kad network. It uses the same format to store its known peers to disk as eMule. They are stored in the file named *nodes.dat* and can be obtained from eMule's directory. After starting the framework its settings can be validated in the options tab. It also allows to turn of displaying debug messages in the log window of the user interface. The main log file

of the framework is called *mflog.txt*. Like in the eMule Tracker it logs error messages and its contents is also displayed in the log window of the user interface. The measurement data is stored in different log files.

Keyword Search Task

To periodically search for a specific keyword the keyword search task can be used. Before it can be started the hash ID of the keyword and the keyword itself have to be set in the configuration file. If the ID is not known for a keyword, it can be obtained using the modified version of eMule for the node insertion attack, through searching for that keyword. After the framework has been started, the interval of the keyword search task should be checked in the tasks tab. Then, the task can be enabled. From now on it will automatically start a keyword search in the specified intervals. The results are written to the file named *keywSearchTask.log*. They can be analyzed later with the LogAnalyzer.

Scan IDs Task

To periodically create a snapshot of all peers in the network, the scan IDs task can be used. It stores its results to the main log file. As already mentioned earlier, this task is premature and should be used for testing purposes only.

Implementing a new Task

The Measurement Framework can be extended with new tasks as follows. As all the task classes are located in the *TaskScheduler* folder in the source code, new task classes should be created there. Every task must implement the interface class *ITask*. As a consequence several methods have to be created in every task. The `Run()` method contains the code that should be executed periodically in the specified intervals. As soon as this method has finished executing, the task will get a new execution time. If this time is reached, the `Run()` method is executed again. To add a new task to the framework, it also has to be registered in the constructor of the *Scheduler* class.

ID Listening

To position the Measurement Framework on an arbitrary ID in the Kad network and listen to the requests, the ID listening extension can be used. After starting the framework, the specified ID has to be entered into the corresponding field in the ID listening tab (see Figure 20). Note that the *keyword* field is just used to remind the user to which keyword (or file) the

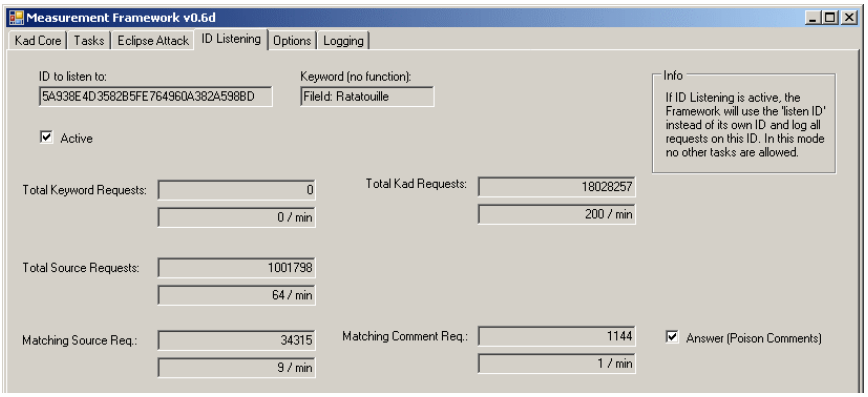


Figure 20: The graphical user interface of the Measurement Framework, showing the ID listen tab.

entered ID belongs to. Both values are stored to disk when the framework is closed.
After activating the checkbox the framework changes its ID to the listen ID and starts logging the requests to the file *idListener.log*. It logs all keyword, source, and comment requests received into this file. Therefore it does not matter whether a keyword hash ID or a file hash ID is used to listen to. To analyze the keyword requests with the LogAnalyzer it is also possible to use the analyses of the node insertion attack, as the format of the log file is the same. If the frequencies of the requests should be analyzed, the main log file has to be used, as an entry with these values is written every minute to it. To attack comment requests with faked comments, the appropriate check box has to be enabled. But before starting the comment poisoning, it should be checked that the name and size of the file whose comments will be attacked is properly set in the configuration file of the framework.

References

[1] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[2] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. ACM Special Interest Group on Data Communication (SIGCOMM)*, 2001.

[3] P. Maymounkov and D. Mazières. A Peer-to-Peer Information System Based on the XOR Metric. In *Proc. 1st International Workshop on Peer-To-Peer Systems (IPTPS)*, 2002.

[4] David Mysicka. Reverse Engineering of eMule. Semester Thesis, Swiss Federal Institute of Technology (ETH) Zurich, 2006.

[5] Moritz Steiner, Ernst W. Biersack, and Taoufik Ennajjary. Actively Monitoring Peers in the KAD. In *Proc. 6th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2007.

[6] Moritz Steiner. Private Communication. In *Proc. 6th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2007.

[7] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 299–314, 2002.

[8] Dan S. Wallach. A Survey of Peer-to-Peer Security Issues. In *International Symposium on Software Security*, 2002.

[9] Seth James Nielson, Scott A. Crosby, and Dan S. Wallach. A Taxonomy of Rational Attacks. In *Proc. 4th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2005.

[10] Eytan Adar and Bernardo A. Huberman. Free Riding on Gnutella. *First Monday*, 5(10), 2000.

[11] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *Proc. 5th Workshop on Hot Topics in Networks (HotNets)*, 2006.

[12] Atul Singh, Tsuen-Wan "Johnny" Ngan, Peter Druschel, and Dan S. Wallach. Eclipse Attacks on Overlay Networks: Threats and Defenses. In *Proc. 25th Annual IEEE Conference on Computer Communications (INFOCOM)*, 2006.

[13] John R. Douceur. The Sybil Attack. In *Proc. 1st International Workshop on Peer-To-Peer Systems (IPTPS)*, 2002.

[14] Lidong Zhou, Lintao Zhang, Frank McSherry, Nicole Immorlica, Manuel Costa, and Steve Chien. A First Look at Peer-to-Peer Worms: Threats and Defenses. In *Proc. 4th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2005.

[15] Elias Athanasopoulos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Misusing Unstructured P2P Systems to Perform DoS Attacks: The Network That Never Forgets. In *Proc. 4th International Conference on Applied Cryptography and Network Security (ACNS)*, 2006.

[16] Naoum Naoumov and Keith Ross. Exploiting P2P systems for DDoS attacks. In *Proc. 1st International Conference on Scalable Information Systems (INFOSCALE)*, 2006.

[17] Karim El Defrawy, Minas Gjoka, and Athina Markopoulou. BotTorrent: Misusing BitTorrent to Launch DDoS Attacks. In *Proc. 3rd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2007.

[18] Xin Sun, Ruben Torres, and Sanjay Rao. Preventing DDoS Attacks with P2P Systems through Robust Membership Management. *Technical Report TR-ECE-07-13, Purdue University*, 2007.

[19] Jian Liang, Naoum Naoumov, and Keith W. Ross. The Index Poisoning Attack in P2P File Sharing Systems. In *Proc. 25th Annual IEEE Conference on Computer Communications (INFOCOM)*, 2006.

[20] Baruch Awerbuch and Christian Scheideler. Towards Scalable and Robust Overlay Networks. In *Proc. 6th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2007.

[21] Andreas Haeberlen, Alan Mislove, Ansley Post, and Peter Druschel. Fallacies in Evaluating Decentralized Systems. In *Proc. 5th International Workshop on Peer-to-Peer Systems*, 2006.

[22] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. of Multimedia Computing and Networking (MMCN)*, 2002.

[23] Eytan Adar and Bernardo A. Huberman. Free Riding on Gnutella. *First Monday*, 5(10), 2000.

[24] Saikat Guha, Neil Daswani, and Ravi Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proc. 5th International Workshop on Peer-to-Peer Systems*, 2006.

[25] Daniel Stutzbach and Reza Rejaie. Improving Lookup Performance over a Widely-Deployed DHT. In *Proc. 25th Annual IEEE Conference on Computer Communications (INFOCOM)*, 2006.

[26] D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-Peer Networks. In *Proc. 6th Internet Measurement Conference (IMC)*, 2006.

Acknowledgements

This report and the implementation of the three attacks for the Kad network, the Measurement Framework, and the eMule Tracker are the results of my master thesis in the Distributed Computing Group of Prof. Dr. Roger Wattenhofer at the Swiss Federal Institute of Technology in Zurich.

During my work I gained great experience in peer-to-peer systems and network programming, especially because this thesis was a continuation of my semester thesis which also covered this field. The implementation of several applications helped me to improve my programming skills in C# and also in C++. Furthermore, the reverse-engineering of eMule messages which was necessary in order to understand its communication, brought me deep insights into the layers of network protocols.

First of all, I would like to thank my supervisors Thomas Locher and Stefan Schmid for their help in many ways and their valuable feedback which always inspired me with new ideas. I would also like to thank Professor Dr. Roger Wattenhofer for his productive and interesting discussions and for giving me the opportunity to accomplish my master thesis in his group.

My appreciation also goes to the system administrators of the Distributed Computing Group and to the Swiss Federal Institute of Technology which provided me with many computing and network resources to perform my measurements and attacks.