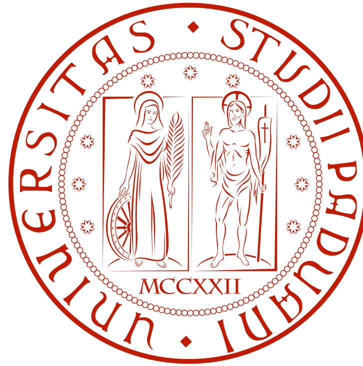
Università degli studi di Padova
Facoltà di Ingegneria



Corso di Laurea Magistrale in Ingegneria Informatica

PariMulo: Kad

Supervisor: **Chiar.mo Prof. Enoch Peserico Stecchini Negri De Salvi**

Student: **Francesco Mattia**

Academic Year 2010-2011

To myself,
Because I deserve it.

Summary

With the advent of broadband connections and computing power available in every kind of digital equipment there is a need to share resources, such as information, among people. To fulfill this need in these years we have seen an amazing growth of distributed systems: cloud computing services, web applications, peer-to-peer systems, etc.
In this context is born PariPari, a project which aims to build a modern peer-to-peer network of computers that runs various services, among which there is an eMule-compatible client, called PariMulo.
As it is well known even to less computer-savvy people, there have been some problems with the centralized server-based structure of the original eDonkey network, and it has helped the development of a new network, Kad, based upon the Kademlia protocol.
This work focuses on the implementation of Kad in PariMulo, starting by first describing the protocol and how the network works, and then providing an in-depth vision of the implementation considering security and performance issues. Finally we make some observations about future possibilities of development.
This thesis was written both for people curious about the inner workings of a modern distributed peer-to-peer network and for developers of software compatible with such networks.

Contents

1 Introduction **1**

1.1 The PariPari Project 1

1.2 PariMulo Plug-in for PariPari 3

1.2.1 Peer-to-peer networks 3

1.2.2 eDonkey and eMule 5

1.2.3 eDonkey network 6

1.3 Implementing Kad 12

2 Kademlia **14**

2.1 Distributed hash tables 14

2.1.1 Key space and Key space partitioning 16

2.1.2 Overlay network 18

2.2 Kademlia 19

3 Kad network **26**

3.1 Network basics 26

3.2 Joining the network 30

3.2.1 Routing table 31

3.2.2 Bootstrap 33

3.2.3 Firewall check 34

3.2.4 FindBuddy (Callback) 35

3.3 Content management 36

3.3.1 Lookup 38

3.3.2 Search 39

3.3.3 Publishing 42
3.4 Advanced features 44
3.4.1 Protocol obfuscation 44
3.4.2 64-bit extension support 45

4 Kad Implementation 46
4.1 Communications 48
4.1.1 UDPListener 49
4.1.2 Packets 50
4.2 Building the network 54
4.2.1 Kad ID (Int128) 55
4.2.2 KadContact 56
4.2.3 Routing table 56
4.3 Lookup 61
4.4 Search 64
4.5 Publishing 68
4.6 Firewall check and FindBuddy 69

5 Considerations and future works 72
5.1 Working on PariMulo 72
5.2 Testing 75
5.3 Future works 79
5.3.1 PariMulo as a framework 80
5.3.2 Kad for embedded systems 81
5.3.3 Developing a new network 84

Bibliography 87

Chapter 1

Introduction

In recent years there has been a lot of hype around distributed systems and peer-to-peer networks: ubiquitous computing and faster connections to the Internet provide new ways of exchanging information that require developing scalable, performant and reliable networks.
PariPari, developed at Università degli Studi di Padova mainly by students, is an example of such networks providing different services as plug-ins. PariMulo, is a plug-in that acts as an eMule-compatible client. As eMule connects to the Kad network, PariMulo plug-in offers such functionality and tries to offer a better user experience. We now describe all these projects, so if you are more interested in technical-related details, you may skip this chapter.

1.1 The PariPari Project

PariPari is a multifunctional, extensible platform for peer-to-peer applications written in Java, it is made of a kernel called *PariCore* and a number of plug-ins, which offer different services locally and over the Internet. PariPari clients can communicate between them using the homonymous peer-to-peer network, which is completely distributed, as it doesn't rely on any server.
Among the services offered there are VoIP, IRC, DNS and other modules, so called *"plug-ins"*; furthermore there are two plug-ins compatible with Bit-

Torrent and eMule[1] networks, respectively called Torrent and Mulo (italian word for donkey).

All those plug-ins rely on so called *internal plug-ins* that offer basic services, such as connectivity with other machines and storage of data, to *external plug-ins* like the ones named above. These internal plug-ins are fundamental to coordinate and control access to resources of the machine where PariPari is running: they provide resources to external plug-ins by wrapping Java objects and adding methods that may limit access to those resources in order to avoid conflicts and exploit synergies between different applications.

PariPari is meant to be launched as a Java Web Start application that anybody can run just by clicking on a web page, requiring no installation. This structure easily allows to keep the software constantly updated thus simplifying maintenance operations and avoiding problems with clients running obsolete versions.

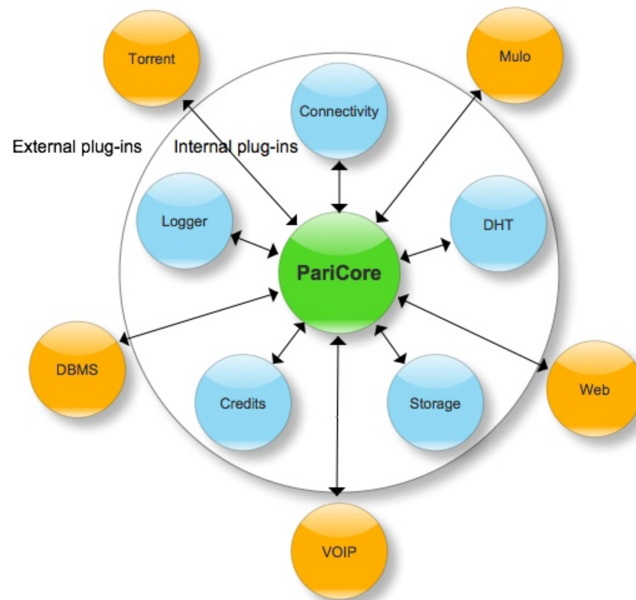


Figure 1.1: Structure of a PariPari client.

[1] More precisely eMule is just a client that connects to the eDonkey and Kad networks.

1.2 PariMulo Plug-in for PariPari

PariMulo is a module for PariPari, a plug-in that uses only services provided by the Connectivity, Local Storage and Logger internal plug-ins in order to have access to the Internet and to local data on disks, so at the moment there is no co-operation[2] among PariPari clients running PariMulo.
PariMulo was first developed as a client for eDonkey2000 peer-to-peer network, allowing for download and upload of files. Later, it has been added support to eMule extensions and the object of this work is the implementation of Kad network support, as in the latest versions of eMule. The structure of the plug-in is modular and independent from the user interface, and that allows to modify PariMulo for different uses, as it will be seen later.

1.2.1 Peer-to-peer networks

Kad network can be described as an *overlay network* with a *peer-to-peer decentralized architecture*. An overlay network is a computer network built on top of another network: in the case of Kad, an Application Layer Protocol[3] is defined to allow intercommunication between clients over TCP and UDP which rely on the IP network.
Peer-to-peer networks is a common architecture for modern networks, as the growth of computing power, data storage and bandwidth capacity of personal computer allows them to act both as client and servers to other computers that join the network, thus we prefer to refer to them as peers[4]. As stated by Wikipedia:

> *Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the*

[2] By co-operation we mean a peer helping another peer to download a content in exchange of some credit. PariMulo clients work perfectly fine with each other.
[3] As described in the ISO/OSI model. http://en.wikipedia.org/wikiOSI_model
[4] Throughout the thesis we use the terms *peer* and *node* interchangeably.

application. They are said to form a peer-to-peer network of nodes.

Many of the most popular file sharing networks rely on peer-to-peer architecture implementing an abstract overlay network: by distributing resources among the nodes generally make the network more resistant to attacks and legal issues. The term however does not define clearly the distribution of resources, thus we can discriminate different generations of peer-to-peer networks.
First generation networks were *hybrid peer-to-peer systems*, in the sense that a server was required to join the network and index the resources to be shared among peers, then peers would connect to each other in order to retrieve desired content. Famous examples of such networks are eDonkey2000 and Napster but both suffered from legal issues that resulted in teardown of original networks.
Second generation networks were *unstructured peer-to-peer systems* such as Gnutella and Freenet. Gnutella, perhaps the most widely deployed of these networks, made use of no server, thus being completely peer-to-peer. The unstructured nature of the network, however, implied that nodes would need to flood neighborhood in order to search for content. As a result, search was often unreliable and the overhead was unbearable, often saturating connections and rendering slower nodes useless. So this architecture has proven to be very unscalable and unreliable, even if modifications to the network, by implementing a tiered system of ultrapeers and leaves, tried to address these problems.
A new generation of networks are *structured peer-to-peer systems*, and being widely addressed in literature, the most common design is based on distributed hash table (DHT), with many projects like CAN, Chord, Pastry, Tapestry and Kademlia itself exploiting this design.
Key features of DHTs are decentralization: nodes do not need any central coordination to form the system; fault tolerance: the system should be reliable even with nodes' churn (nodes constantly joining, leaving or falling);

scalability: the system should be efficient even with millions of nodes.
Advantages of structured peer-to-peer systems are evident: the lack of a centralized server eliminates the single point of failure for the network, as tearing down the server would render the network unusable. However the central server provides faster and more accurate query results to peers than unstructured systems, as the flooding query model proved to be unreliable and significantly less efficient, with the undeniable advantage of being completely decentralized. DHTs should get the best of both worlds by providing a reliable and efficient over a decentralized network, anyway we'll be back on the subject to verify these claims.

1.2.2 eDonkey and eMule

eDonkey2000 (as known as *ed2k*) is a software application and an hybrid peer-to-peer network named in the same way. It featured some key advantages over Napster, such as downloading of files from multiple sources at once and indexing of files shared by file hashes instead of filenames, and acquired great popularity after the demise of Napster. eDonkey2000 was developed by a company, *MetaMachine*, which had possession of the servers managing the network and kept the source code private, thus implementing *security-by-obscurity*, which usually isn't a clever security policy. In fact a number of eDonkey-compatible clients and servers appeared, the most famous example is the eMule client. eMule is an open-source software released under the GNU General Public License. thus enabling willing developers to join the cause and enhance the application.
As eDonkey was closed-source, eMule developers had to *reverse-engineer* the protocol, which is way more difficult and error-prone than following documentation guidelines. While eMule client was open-source, the main server softwares developed by reverse-engineering, *Lugdunum eserver* and *satan-edonkey-server* are free (as in gratis) but closed-source, thus choosing again the questionable policy of security-by-obscurity, as it was declared by Lugdunum developers on their website. With the growth of popularity of eMule,

the original protocol has been modified first by eMule developers, who added the so called *eMule extensions*, and then, as the software is open-source, many third-party developers started their own fork projects of eMule adding features and protocol extensions incompatible with other clients thus leading to a jungle of protocol dialects.
In 2005, eDonkey was discontinued due to legal issues with RIAA[5], but the eDonkey network survived as servers were brought up all around the world: interconnections between servers and features added to eMule prevented the network from splitting in a multitude of smaller networks. As eMule remained the only *de-facto standard* client to connect to eDonkey servers, it was possible to develop a new completely decentralized peer-to-peer network, Kad, which implements the Kademlia overlay protocol, as described in [1]. This network has gained a lot of popularity due to the constant tear down of the several eDonkey servers, proving so far to be a valid alternative to the hybrid system, being both robust and scalable.
Currently, the main threats to the Kad network are attacks, which will be discussed later, and protocol filtering to peers trying to join and participate to the network.

1.2.3 eDonkey network

We give a brief description of the eDonkey2000 network and protocol to confront it later with Kad.

Communications

The eDonkey2000 network, as an hybrid peer-to-peer system, is an overlay network built on top of the IP network using both TCP connections and UDP datagrams exchange. So called ed2k packets are sent through the TCP streams or using UDP datagrams as wrappers. TCP connections are used mainly for reliable and durable connections to servers, and between peers

[5]RIAA - Recording Industry Association of America

to download/upload files. UDP datagrams, due to their smaller header and unreliable nature, are preferred for fast information exchange, both with peers and servers.

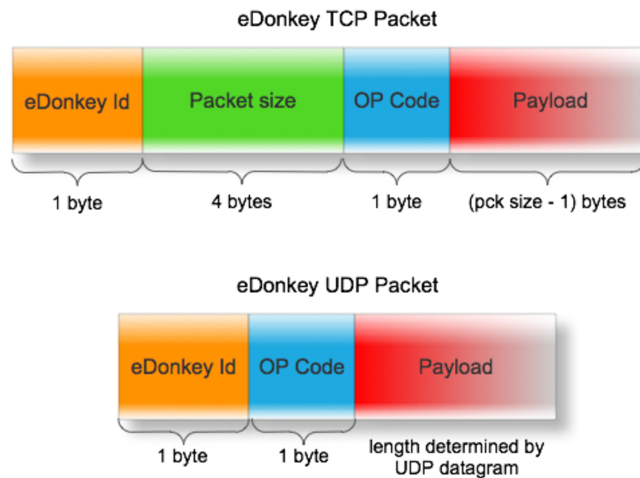


Figure 1.2: **Structure of eDonkey packets** sent through TCP or UDP. First byte is *eDonkey packet identifier* (`0xe3`), while *OP Code* determines the packet type. Packet size in TCP must be specified, and the size field is the length of the packet after the size field itself. UDP datagrams instead define the length of the packet, therefore no packet size field is needed.

Network topology

The network relies on number of servers, which have the duty to mantain an index of connected nodes and the files shared. In such a sense the content of resources shared is not sent to the servers but just a reference to the resources, and then the peers retrieve the content from other peers, so the indexing of contents is centralized while distribution is completely peer-to-peer.
A peer joining the network, it connects to a single server through a TCP

connection that is mantained for the whole session, and it publishes its shared files.

Peers Identification

Upon connection to the server a peer acquires a *ed2k ID* given by the server: if the peer is accessible through a specified TCP port (communicated at the connection to the server) it will be given a 4-bytes *High-Id* calculated from its IP address in the form $a.b.c.d$, where each letter stands for a byte, with the following formula:

$$ed2k\ id = a + b \cdot 2^8 + c \cdot 2^{16} + d \cdot 2^{24} \quad (1.1)$$

Else, it will be given a random, 4-bytes $ed2k\ id < 2^{24}$ *Low-Id*[6] by the server. The main difference is that each server mantains its own list of connected peers, so in two different servers may exist peers with the same Low-Id, while, for obvious reasons, there can be just a peer with an High-Id.
A 16-bytes user ID, persistent across sessions, is self-assigned to each peer and used for identification in order to implement a simple credit system between peers.

Resources Identification

The resources shared in the network are files. One of the main problem of Napster was that identical files with different file names were treated as different resources. The ed2k network uses a clever system: every file is uniquely identified by its file hash calculated with the cryptographic hash function MD4[7], which generates a 128-bit identifier, and its file size (4 byte integer). As the file name does not contribute to the calculation of MD4 and file size, files with different name but exactly identical have the same identifier and to the network will appear as the same resource.

[6] This way it is possible to discriminate High-Id from Low-Id clients just by looking at the *ed2k id*.

[7] http://en.wikipedia.org/wiki/MD4

Search

Searching the network for published contents is possible in two ways: through a query to the server the peer is connected to (*called local search*), or sending queries to servers the peer has knowledge of (*global search*). Local search is performed with an exchange of ed2k packets with the server using the TCP connection, while for global search ed2k packets are exchanged through UDP datagrams.
Search of a content technically is divided in two steps: first a querying client sends a *keyword search* to get a list of files published related to the keyword (more correctly a string, it can contain multiple words); then when a file is chosen by its hash and size, a *source request* is sent, and a list of sources which has the file is given, so the client can start the download straight off. Advanced keyword search features (such as specifying file type) are all dealt

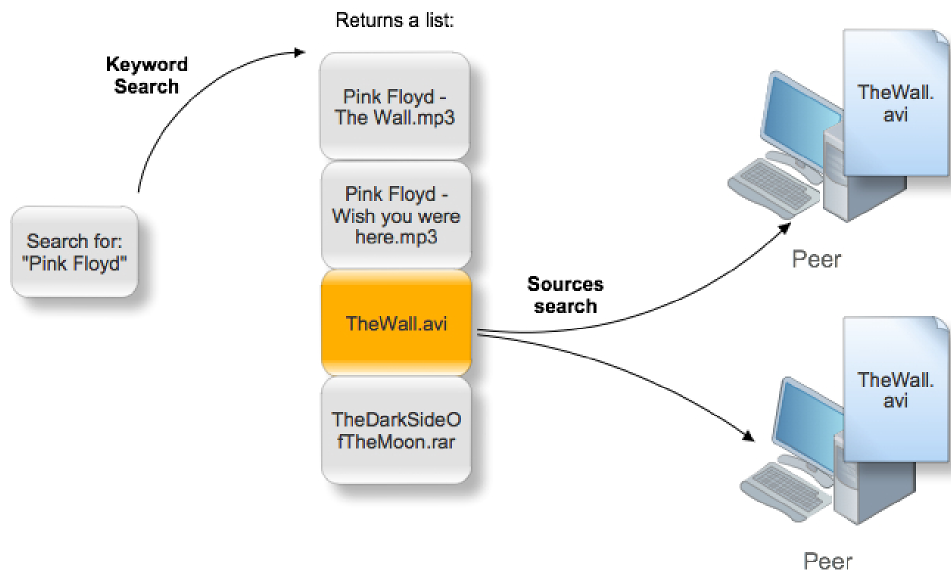


Figure 1.3: Looking for content, a peer performs a keyword search, in this case “Pink Floyd”, which returns contents related to the string. Choosen a file to be downloaded “TheWall.avi”, a source search to locate serving peers for such resource is performed.

putting options in the ed2k packets sent to servers. Servers are responsible just for contents published by peers connected to them.

Publication of shared files

The publication to the server of files shared by the peer happens sending a series of ed2k packets containing metadata referring to the files, such as file name, file type, size and possibly other information content-related. All the references are stored in the server for a period, after which they are deleted unless the peer republishes them. In order to avoid overloading the server at the beginning of publication, published files are split in packets sent with a time delay and, if the peers has many files, only a portion is published. However the limits and delays aren't strictly defined by the protocol, so different client and servers may deal the question otherwise.

File exchange

When a file to download is chosen and sources are available, the downloading peer contacts each of the source peers by mean of a TCP connection and requests parts of the file. If there are too many peers downloading, a new request will be enqueued and served later. A enqueued client does close the TCP connection, which will be established again when the uploading peer can serve the client.
The file exchange will be dealt more in-depth later, as this part of the protocol is in common with the Kad network.

Callback

The *Callback* is a mechanism that allows High-Id peers (nodes that can accept incoming TCP connections) to instantiate a TCP connection with Low-Id peers (nodes that can't accept incoming TCP connections). This works only between peers connected to the same server, as the requesting

High-Id client must send an ed2k packet indicating a *Callback request* to the server, which will forward the request to the Low-Id peer through the active TCP connection. Now the Low-Id client will actually *call back* the requesting peer initiating a TCP connection. At the moment, according to ed2k clients and servers, there is no way for two Low-Id clients to communicate. For more details on the Callback mechanism please refer to [7].

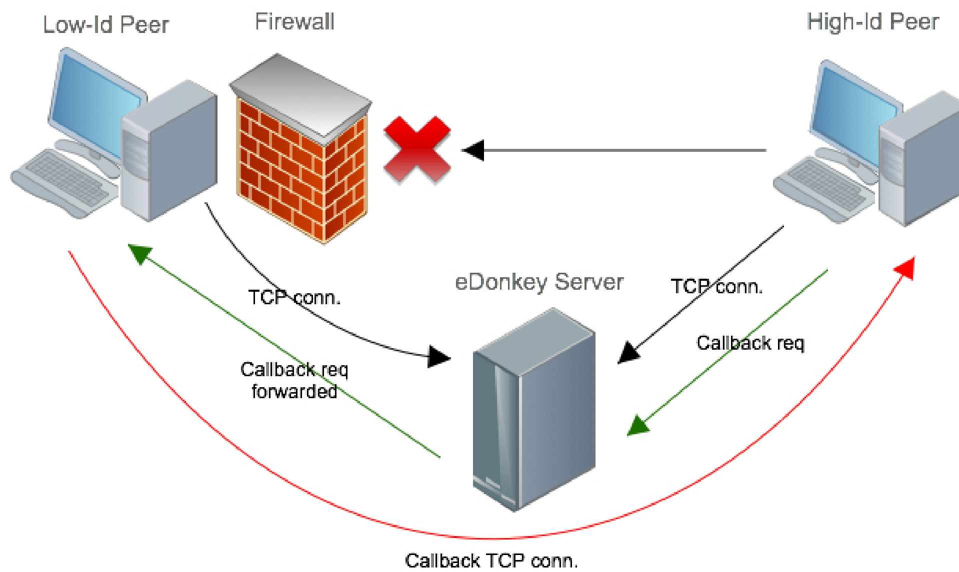


Figure 1.4: **Callback mechanism**: an High-Id peer cannot connect directly through TCP to a Low-Id Peer, therefore it sends a Callback Request to the eDonkey server which forwards the request to the Low-Id Peer (green arrows) through the estabilished TCP connections (to the same server). Then Low-Id Peer calls back directly the High-Id Peer with a TCP connection (red arrow).

Credits

Ed2k clients implement a credit system, necessary to prevent “leechers” from draining all upload capacity of peers without allowing other clients to download from them. The credit system implemented in eMule is quite simple: a downloading client rewards an uploading peer by modifying its queue rank-

ing proportionally to the amount of data uploaded. Credits between peers are calculated as follows:

$$Credit = min\left(\frac{\texttt{bytes received} \times 2}{\texttt{bytes sent}}, \sqrt{\texttt{MBytes received} + 2}\right)$$

Credit is in the range of 1 and 10, and it should be noted that is not the only queue ranking modifier considered, other parameters such as waiting time in queue and file priority can be considered. The credit system works just between pairs of peers and, being credits stored and managed on the peers, so there is no authority managing the peers and a serving peer doesn't get any credits with the whole network, but just with (honest) peers served. This, as it concerns file exchange, is another feature in common with the Kad network and will be reviewed later.

Protocol obfuscation

In 2006 was implemented in eMule clients a new feature to prevent ed2k (and Kad) communications from being filtered or eavesdropped, protocol obfuscation, which tries to obfuscate both TCP and UDP traffic by mean of RC4 stream cipher[8], using Diffie-Hellman key exchange protocol. Even if RC4 is robust, the whole system as it was implemented is not very efficient and even obfuscated communications can be filtered quite easily.

1.3 Implementing Kad

When we started working on PariMulo, in late 2007, massive attacks and tear down of ed2k servers that lead us to think that eDonkey2000 network would demise quickly. As of this writing, in the beginning of 2011, not only eDonkey2000 network still works, but the Kad network proves to be reliable

[8]http://en.wikipedia.org/wiki/RC4

so far and working, even if many attacks are possible and ISPs still try to filter peer-to-peer traffic.
As the Kad network gained popularity, counting millions of nodes connected at the same time, implementing it in PariMulo was mandatory choice. eMule (and its derivatives) implements it since 2006, while the other most famous client, aMule uses mostly the same code for the Kad network support. When we first developed PariMulo with support for eDonkey network, as stated in [2], we relied mainly on the protocol description given in [4] and reverse engineering of the protocol. Even if we achieved good results developing a client that is compatible the other clients, it has proven not to be the best choice, as perhaps directly studying and porting the eMule source code would have been a more efficent working methodology.
In fact, the real problem with this approach is that eMule and aMule are written in *C++*, an object-oriented programming language, like *Java* which is the language chosen for PariPari. Despite being both languages object-oriented, there are quite some differences in memory management (Java provides *Garbage Collecting* capabilities), networking and concurrency frameworks.
We'll be back on the subject later, for now, it suffices to say that for Kad implementation we chose to analyze eMule and study its sourcecode: Kad inner workings are much more complex and subtle than eDonkey, and relying just on documentation and reverse engineering would have been really difficult and unfeasible.

Chapter 2

Kademlia

Before starting the description of the network, we must define some key concepts that otherwise may confuse the reader. *Kademlia*, as described in the original document [1], is a distributed hash table (DHT) upon which is built the *Kad network*, which is most accurately described by the eMule source code, where the specifications of *Kad protocol* are also completely defined. This chapter give a detailed overview of DHT technology and reviews Kademlia original document, pointing out differences with other DHTs and the Kad network as it was implemented.

2.1 Distributed hash tables

As stated in Wikipedia[1]:

> A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table; (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a

[1] http://en.wikipedia.org/wiki/Distributed_hash_table

> change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

A DHT can serve many purposes, being fundamentally a tool to store and retrieve information from a distributed network of computers. However, to make the system that uses a DHT to work correctly and efficiently, many issues like distribution of information, nodes churn[2], security, scalability, redundancy, etc. should be considered thoroughly.
Research on DHT, and more generally on structured peer-to-peer systems, was boosted by the problems of unstructured peer-to-peer systems which led researches to find more efficient network designs to share information over the IP network. As a result CAN[3], Chord[4], Pastry[5], Tapestry[6] and Kademlia [1] were proposed, and the latter counts on the most widely deployed implementation: Kad.
The DHT is built, in its main incarnations, as a group of equal peers that cooperate to form an *overlay network* without coordination from a central entity. All the "intelligence" in the network resides on the peers[7]: they manage the network taking care of nodes churn and distribution of the content. The big step forward from unstructured peer-to-peer systems is that a node on average has knowledge of just $O(log\ n)$ nodes in a network of n peers. This is really helpful to improve scalability and fault tolerance, as a minimal number of messages must be exchanged between peers.

[2] The rate at which nodes join or leave the network.
[3] CAN: http://berkeley.intel-research.net/sylvia/cans.pdf
[4] Chord: http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf
[5] Pastry: http://research.microsoft.com/ antr/PAST/pastry.pdf
[6] Tapestry: http://pdos.csail.mit.edu/ strib/docs/tapestry/tapestry_jsac03.pdf
[7] We use the terms "peer" and "node" interchangeably.

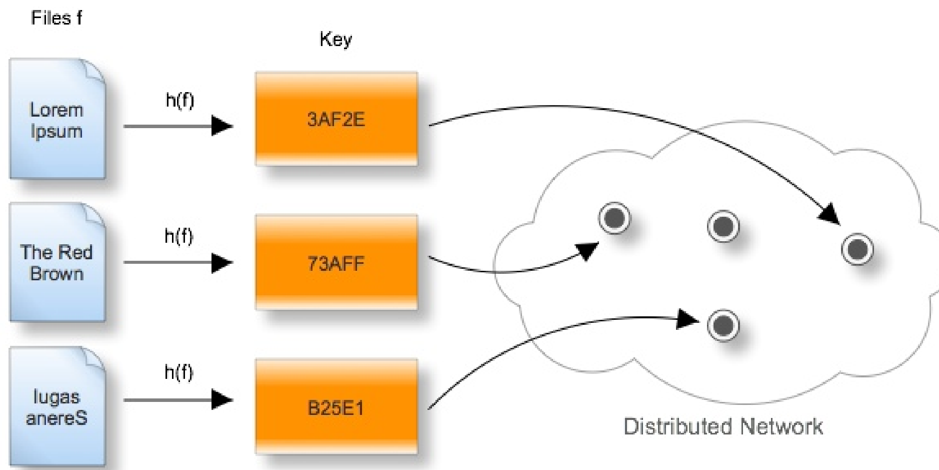


Figure 2.1: An example of DHT network: resources are stored on the network on peers responsible for keys. A key for a resource is obtained through an hash function.

2.1.1 Key space and Key space partitioning

As it was said, the network acts as an hash table, thus a *key space* must be defined, so that nodes can store content by associating a value to a key. The key space is commonly a set of bits, for instance 128-bit in Kad network[8]; this choice serves well to define the concept of *distance* between keys.
Information needs to be distributed and stored across nodes, so we must define a *key space partitioning* so that nodes can efficiently and reliably distribute and retrieve data from other nodes, knowing which nodes take care of the content they are looking for. DHTs usually use *consistent hashing* techniques over traditional hashing, so that even with the number of nodes changing, a minimal redistribution of content should be done.
To make things more clear, we make an example of traditional caching. Think about storing object o in n machines: a common way would be to store o in the machine numbered $hash(o)\ mod\ n$. While the number of n machines

[8]Kademlia original paper describes the system with a key space of 160-bit, actually Kad network has a 128-bit key-space.

is fixed this solution works well, but when n changes over time this proves to be catastrophic as the machine where all the previously stored o objects reside changes.

Consistent hashing basically solves the problem by associating each of the

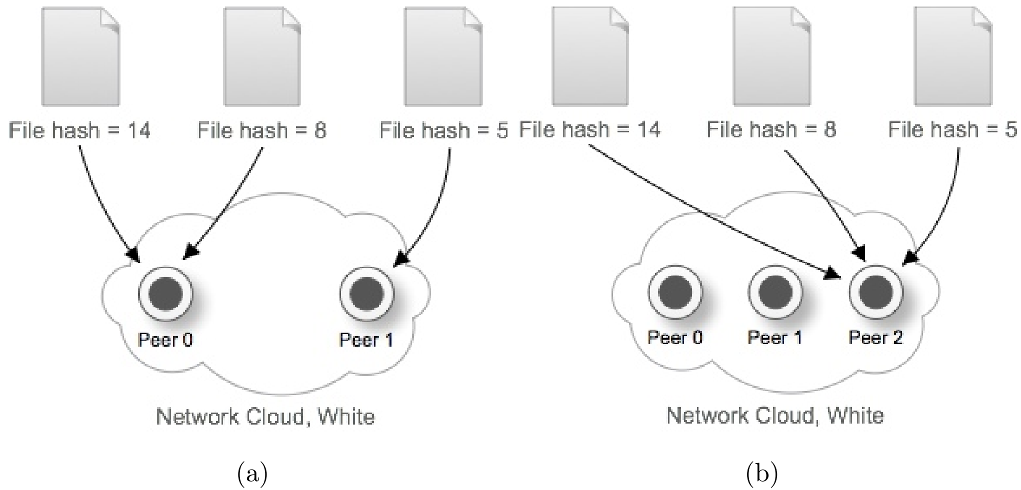
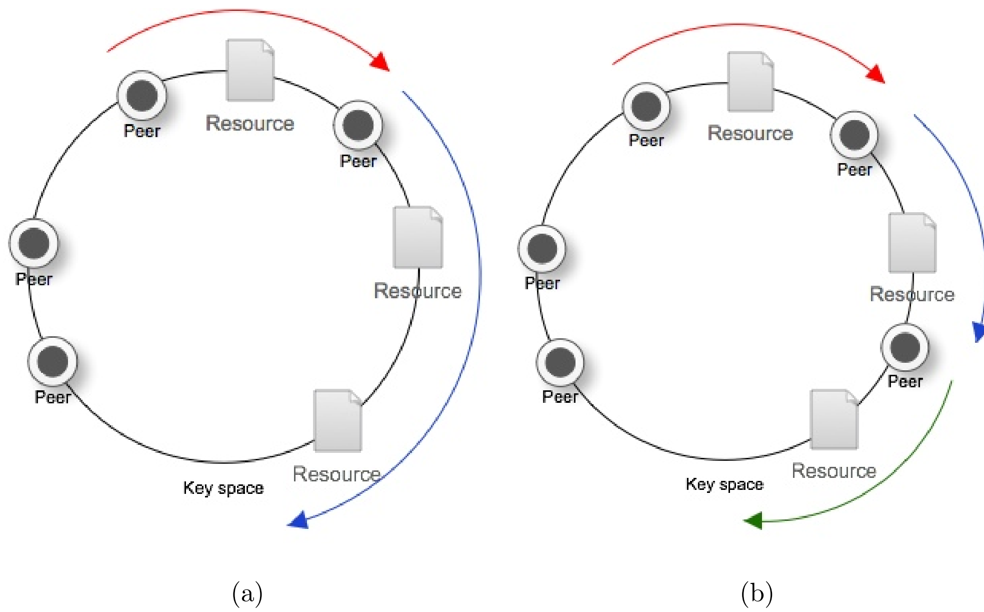


Figure 2.2: Traditional Hashing: Files in (a) are stored by their file hash in node *hash(f) mod n*. When a new node joins the network (b), resources stored need to be moved to the new node.

n machines to a key in the key space, in order to map each machine to an interval in the key space. In this way, all the objects are consistently mapped to the same machine as long as it is possible, thus in a network with K keys stored and n machines, an average of K/n items need to be redistributed when the number of machines change.

For example, we can assign each node an identifier i in the key space, and we can define a distance function $\delta(k_1, k_2)$ between each pair of keys k_1 and k_2, unrelated to network latency or geographical position. In this way we can assign all the keys k_x to the node with id i_1 for which $\delta(k_x, i_1)$ is minimal. When a new node with id i_2 where $\delta(i_2, k_y) < \delta(i_1, k_y)$ joins the network, responsibility of key k_y is shifted from node with i_1 to node with id i_2. Conversely, when node i_2 will leave the network, responsibility of k_y will be moved

back to node i_1. This mechanism allows the network to work efficiently even with high rates of nodes churn.



(a) (b)

Figure 2.3: Consistent Hashing: The circle represents the key space, on which are displayed both peers and resources taken care of. Each node has the responsibility for all the resources on the network represented clockwise on the circle until the next peer. When a peer joins the network (the one with green arrow), as depicted in (b), it takes control of resources handled by the peer with blue arrow. Other peers are not involved in the process (red arrow). Therefore in a network with K resources and n peers, upon joining or leaving of a node, an average of K/n resources need to be relocated.

2.1.2 Overlay network

Another key aspect of distributed hash tables is the *overlay network*, that is to say how the nodes connect to each other and have knowledge of network topology at application level over the lower level network (typically the IP

network). To correctly track a node that is responsible for key k, a node i_1 must either be itself the node responsible for k or have a link to a node i_2 closer (by definition of the function δ) to the key k. So, using a technique called *key-based routing*, it is possible to correctly reach the node responsible for k by forwarding, in a greedy[9] manner (which does not always provide an optimal solution but surely a correct one), the message to the closest node to the key k.

Other than ensuring the correctness of routing through the network, a trade-off between nodes degree and route length is to be considered. An high nodes degree means that each node know a lot of neighbors, thus requiring a lot of maintenance overhead (closely related to nodes churn). The other side of the medal is that knowing a lot of neighbors requires less *hops* to get to the node responsible for a key, viceversa if a small number of neighbors is known, a large number of hops will be required on average. The most common implementations let the nodes store reference to $O(log\ n)$ nodes in a n nodes network, thus requiring an average of $O(log\ n)$ hops, even if this choice proved not to be optimal considering the two aforementioned parameters but allow for more flexibility in terms of neighborhood choice.

2.2 Kademlia

Now it will be presented Kademlia distributed hash table protocol, as described in [1], on which the Kad network relies, depicting the key features that make Kademlia more suitable for the deployment of a file sharing network, related to other DHT designs. Citing the original document:

> Kademlia takes the basic approach of many DHTs. Keys are opaque, 160-bit quantities (e.g. the SHA-1 hash of some larger data). Participating computer each have a node ID in the 160-bit key space. <key,value> pairs are stored on nodes with IDs "close"

[9] Algorithms that follow the greedy paradigm at each stage make the locally optimal choice, but this does not guarantee that the global solution will be optimal.

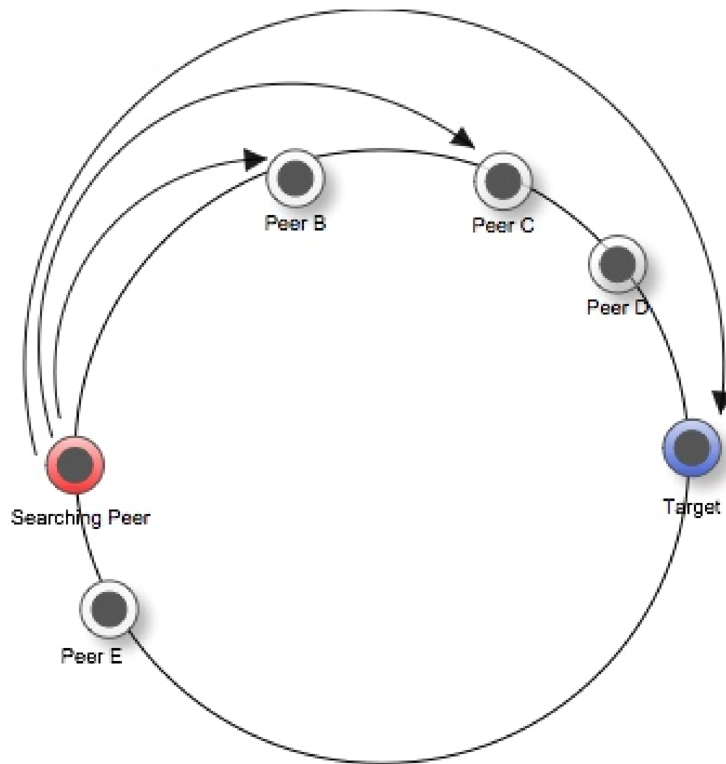


Figure 2.4: Routing in a DHT: Approaching a target key. A searching peer recursively looks for closer nodes to the target that, thanks to routing table structure, will have a better knowledge of nodes close to them. Not all the closer peers will join the procedure, it depends upon routing tables of the peers contacted.

> to the key for some notion of closeness. Finally, a node-ID-based routing algorithm lets anyone efficiently locate servers near any given target key.

Distance function

Kademlia best intuition is to use a *XOR metric* to compute the distance function δ between two identifiers in the key space. The distance between two identifiers x and y, $\delta(x, y)$ is defined as $\delta(x, y) = x \oplus y$. We note that this operations shows three interesting properties:

- $\delta(x, x) = 0$
- Symmetry: $\delta(x, y) = \delta(y, x)$
- Triangular property: $\delta(x, y) + \delta(y, z) \geq \delta(x, z)$

Differently from Chord[10], Kademlia exploits the *symmetry* of the distance function to let the nodes learn useful routing information from queries they receive. Symmteric routing tables also allow to send a query to any node within an interval, not requiring to contact a precise node that stores the needed value. Moreover, Kademlia uses a single routing algorithm, called lookup to locate nodes near any ID. XOR metric exhibits another useful property: it is *unidirectional.*

- $\forall$ keys k_1, $\Delta \geq 0$, there is exactly one key k_2 such that $\delta(k_1, k_2) = \Delta$

Because of this feature all lookups towards a key converge along the same path, regardless of the originating node.

Identifiers and Routing tables

Each node is assigned a 160-bit opaque ID. Nodes are seen as leaves in a binary tree, and the tree is divided in 160 successively lower subtrees not containing the node, with a progressively longer prefix in common with the node.

> **Ex.** If our key space was 4-bit long, a node *0011* will have its tree divided in 4 subtrees: the first one, with prefix *1* representing the other half of the network not containing *0011*; a second one, with prefix *01*, sharing the first bit with *0011* thus being closer and containing half of the nodes of the first subtree (1/4); the third one has prefix *000*, containing 1/8 of the nodes; and the

[10]Chord is another popular protocol for distributed hash table, which models a ring network topology, with nodes referencing successor nodes, thus being asymmetric.

last one *0010* containing just one node. Of course nodes with some prefixes may not be connected to the network, so any of the subtrees could be empty, with a major probability of lower trees.

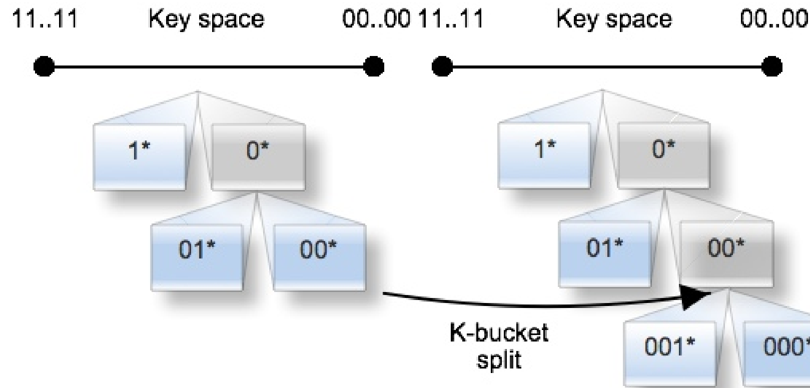


Figure 2.5: Routing table for a node with ID *00..00*. Blue squares (leaves) are buckets containing nodes (*k-buckets*), while grey squares form subtrees that can split (up to the 160th level). Therefore a node with time has a deeper knowledge of its neighborhood.

Kademlia ensures that each of the subtrees contains at least one node, allowing the lookup process to locate nodes by IDs to work correctly. Each of the subtrees in a node's tree, according to the distance function defined above, keeps nodes of distance between 2^i and 2^{i+1}, where i is subtree's level, from the node itself.

The tree is called *routing table* and the subtrees, each containing not just a node but a maximum of k nodes, are called *k-buckets*. Each k-bucket is saved as a list of nodes ordered chronologically, while each of the nodes' reference contains the IP address, UDP port and of course the node ID.

k is a parameter chose such that any given k nodes are unlikely to fail within an hour. Kademlia uses a mechanism to update the k-buckets whenever a node receives any message from another node. If a new contact is received and the k-bucket it belongs is not full the contact is inserted. Else, if the

k-bucket is full, the node will ping the least-recently seen contact in the k-bucket, in order to decide whether to discard the new contact or the old one, if it does not reply.
This mechanism that favors older contacts is used on the assumption, based on studies found on the Gnutella network (more details in [1]), that older nodes are more likely to stay connected longer. Moreover, it provides a resistance to DoS attacks, as malicious entities cannot flush a node's k-bucket so easily with bogus nodes.

Protocol

Kademlia original document defines four RPCs[11] to enable communication between peers: *PING*, *STORE*, *FIND_NODE* and *FIND_VALUE*.
PING is used to probe if a node is connected, sent directly to the node enquired, which is required to respond. *STORE* is used to store on a node key-value pair for later retrieval. *FIND_NODE* is used to locate nodes near a specific 160-bit ID, as the receiving node will respond sending a list of the k closest nodes to the requested ID, with references containing the IP address, UDP port and node ID, so that the requesting node can use them in its routing table. *FIND_VALUE* behaves similarly, but if the receiving node has previously received a *STORE* RPC for the keyword requested, it will respond with the stored value.
A simple *challenge-response* mechanism to prevent address forgery is also implemented in all the RPCs, in fact, RPCs must contain a random ID which will be echoed by the replying node.
The lookup procedure, used to locate nodes close to an ID in the network, relies on the aforementioned RPCs. The algorithm exploits concurrency by defining an α parameter that is the number of nodes initially chosen to send multiple *FIND_NODE* requests at-once, instead of sending just one and waiting for the response. The concurrency provides some robustness to the

[11] Remote Procedure Call: activation of a subroutine or procedure by a program, where the activating computer is different from the one running the procedure.

algorithm in case of stale nodes and faster lookups. Once an answer is received, a new request is sent to the α closest node to the target ID, which hopefully should be nodes received from previous RPC, as, according to routing table maintenance policies, nodes should have a good knowledge of their neighborhood. The lookup terminates as soon as the querying node receives responses from the k closest contacts it has seen. Lookup is used to imple-

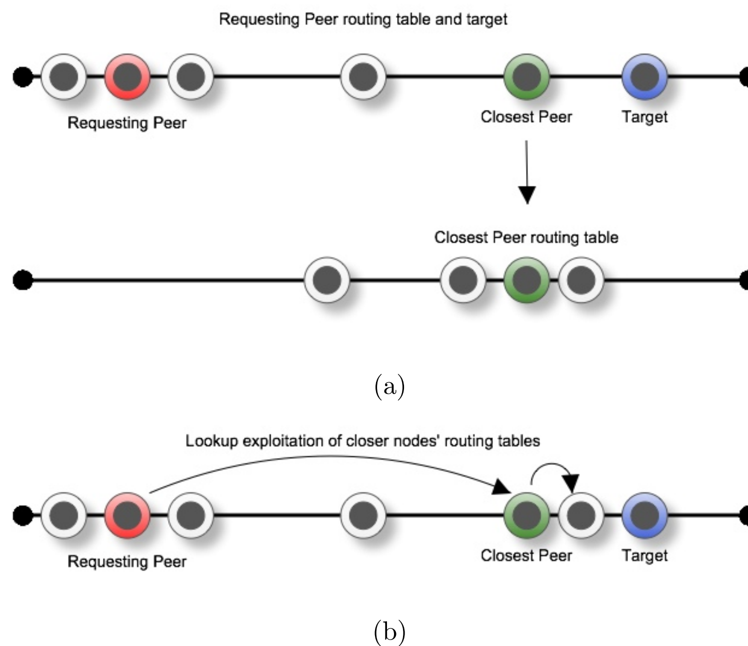


Figure 2.6: Simplified Lookup procedure: the requesting peer asks to contacts close to the target for closer nodes. They reply with the nodes and requesting peer will recursively approach the target.

ment most operations such as store and retrieval of a value: the algorithm converges recursively towards the target ID, where nodes with minimal distance from the target are responsible for holding the value of a certain key. When the k closest nodes to a target are known, *STORE* or *FIND_ VALUE* RPCs are used.

In the case of storing a value, in order to limit the stale index information, a publication of a key-vaue pair expires after 24 hours, and a republishing is

needed.
When performing a value lookup, a *caching mechanism* was implemented to help managing more popular keys: whenever sending *FIND_ VALUE* requests, the requester would be responsible of storing the value found in the closest node to the target that didn't provide back the value. By the way, to prevent "over-caching", expiration time is set exponentially inversely proportional to the number of nodes between the current node and the node whose ID is closest to the key ID. Even if messages exchanged with other nodes refresh the routing table, there is a risk of stale nodes in the k-buckets, which can lead to lookup failures, so a node is expected to do some routing table maintenance by performing lookups towards buckets not recently refreshed.
A node u joining the network must have at least a contact w to start a lookup procedure towards itself (its own node ID). Then, it should refresh all the k-buckets further away than its closest neighbor. This way, node u populates its own routing table but also inserts itself in other nodes routing tables.

Chapter 3

Kad network

Kad network was built upon the Kademlia protocol [1], but it shows many differences from the original document. Furthermore Kad clients provide a complete protocol, while the document doesn't explain many of the mechanisms implemented.
In literature there are many documents describing more or less in detail the inner workings of the Kad network, the most notable is probably René Brunner master thesis [3]. However, we advise the reader to rely for accuracy on the eMule source code, as it may change over time rendering any documentation obsolete.

3.1 Network basics

Kad network is mostly run by clients connected also to the eDonkey2000 network, so Kad protocol retains some features of ed2k protocol.

Topology

As seen in the previous chapter, the network does not rely on any other entity other than the peers. Anyway, a connecting peer needs to know at least another peer to join the network, therefore lists of peers for a first-time

bootstrap are available on the web[1].

A node has knowledge and exchanges messages with only a really small portion of the network, typically $O(log\ n)$ in a network of n nodes, and the contacted nodes vary depending on where the resources requested are located (in the keyspace, not physically). In figure 3.2 is shown a peer that accesses a resource in another part of the network, connecting directly with nodes from that part using previously known nodes to approach them.

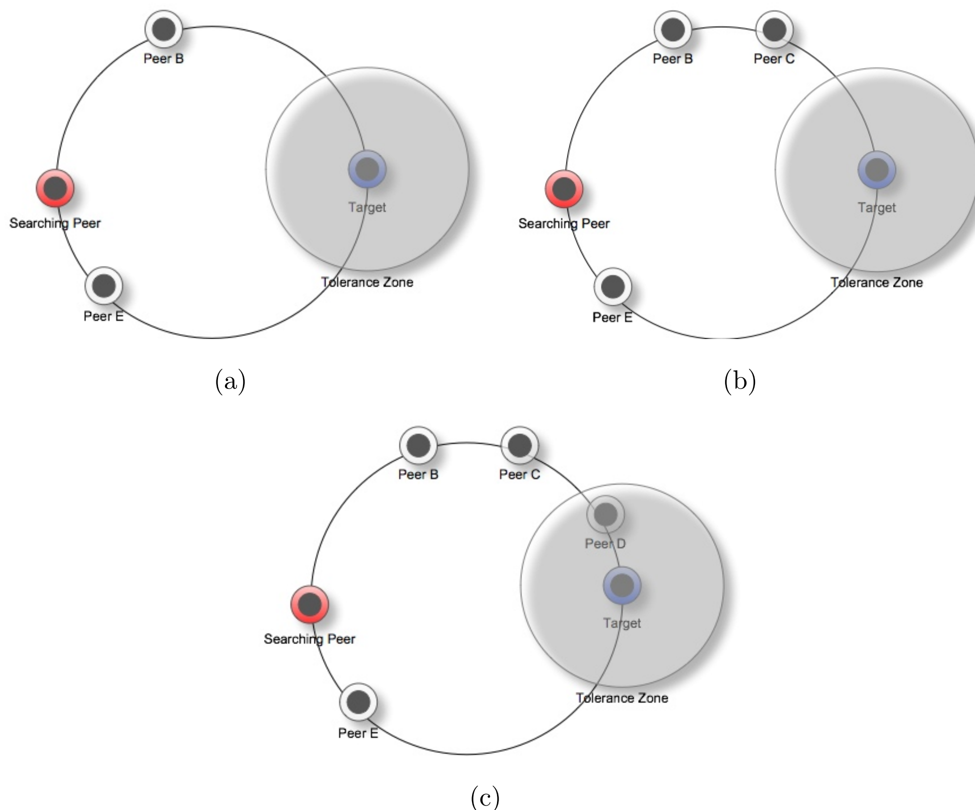


Figure 3.1: **Lookup iterations**: Searching peer sends requests to known nodes that will reply with closer nodes to the target. Once enough peers in tolerance zone are known, the process can terminate.

[1]This poses some security problems, as we should be concerned of who manages those lists. If we downloaded a list of malicious nodes we could join a Kad fake network without noticing!

Communications

Nodes need to communicate quickly to other nodes, sending short messages that even if lost once in a while don't really jeopardize the network functionality, therefore UDP datagrams are used to carry those messages: they offer shorter header, no handshake, congestion or flow control that would cause a useless overhead. As we have seen concurrency in lookups, as described in [1], offers some robustness to packet loss.
TCP connections are used only in communications between peers that exchange files, as file exchange requires reliable data transfer, congestion and flow control that TCP provides. These communications work exactly as in eDonkey, in fact there is no distinction during file exchange with peers known from Kad or eDonkey network.

Messages over UDP datagrams are sent as Kad packets, with the same

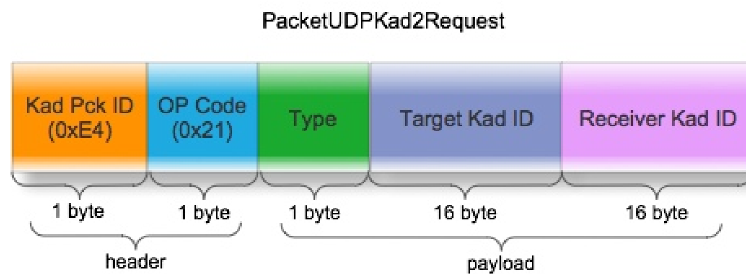


Figure 3.2: Structure of `KAD2_REQUEST` packet. All UDP sent packets have a 2-byte header and a payload as big as the rest of the datagram itself.

structure of ed2k/eMule ones. As no connections are established, usually there is an exchange of UDP datagrams: all response timeouts, packet loss and stale nodes must be managed at *application level*, that is to say by the client.

Peers Identification

Peers and resources are both identified by means of a 128-bit ID[2] called *kad ID*. As mentioned in subsection ??, it is useful to associate peers in the same key space of resources (exploiting consistent hashing advantages), in order to exploit consistent hashing advantages. A 128-bit key space lets allocate up to 2^{128} different objects, a very large number compared to the number of peers in the network. So the second reason is that, even choosing a random ID, the probability of ID collision is negligible. To demonstrate that, we first calculate the probability that in a group of n users, all the n IDs are different:

$$\bar{p}(n) = 1 \times \left(1 - \frac{1}{2^{128}}\right) \times \left(1 - \frac{2}{2^{128}}\right) \times ... \times \left(1 - \frac{n-1}{2^{128}}\right) \tag{3.1}$$

$$= \frac{2^{128} \times (2^{128} - 1) \times ... \times (2^{128} - n + 1)}{(2^{128})^n} \tag{3.2}$$

$$= \frac{2^{128}!}{(2^{128})^n (2^{128} - n)!} = \frac{n! \cdot \binom{2^{128}}{n}}{(2^{128})^n} \tag{3.3}$$

The event of at least two of the n users having the same ID is complementary to all n IDs being different:

$$p(n) = 1 - \bar{p}(n) = 1 - \frac{n! \cdot \binom{2^{128}}{n}}{(2^{128})^n} \tag{3.4}$$

We find an approximation that is easier to calculate, using $1 - x < e^{-x}$ inequality:

$$\bar{p}(n) = \prod_{k=1}^{n-1} \left(1 - \frac{k}{2^{128}}\right) < \prod_{k=1}^{n-1} \left(e^{-k/2^{128}}\right) = e^{-(n(n-1))/(2\times 2^{128})} \tag{3.5}$$

We now can calculate an upper bound for $p(n)$, and find a value for a network with 10^8 users (which is a relax bound).

$$p(n) = 1 - e^{-(n(n-1))/(2\times 2^{128})} \tag{3.6}$$

$$\tag{3.7}$$

[2] In the original Kademlia document, a 160-bit identifier is proposed.

$$p(10^8) < 10^{-18} \quad (3.8)$$

Rare, not arranged collisions, do not cause the network to fail much more than stale nodes do, so, small but greater than zero probabilities are acceptable. Peers at first launch of the client application randomly choose a kad ID, that in theory should not be changed across sessions. Studies found that is not true: some peers change kad ID and most importantly some peers do not choose randomly the ID, a way to exploit some vulnerabilities of the network.

Resources Identification

As said above, it is useful to have both peers and resources mapped on the same key space. Even if Kademlia original document proposed a 160-bit key space, for backward compatibility with eDonkey, a 128-bit identifier was chosen.
In fact, the most important resource in the Kad network, files, are indexed by their file hash. Contrary to eDonkey network, no file size is required, but the file identification on the network relies just on the *MD4* hash as described in subsection 1.2.3, that associates a 128-bit identifier to the file.

3.2 Joining the network

For a peer to join to the network, it needs at least to exchange UDP datagrams with other peers connected to the Internet. It has to retrieve in some way at least a peer connected to the Kad network, thus a *bootstrap mechanism* is implemented.
Nodes known during the session are saved in a particular structure similar to the one described in the Kademlia original document, called *routing table*, which requires maintenance to prevent stale nodes thus ensuring efficiency. Differently from eDonkey network, there is no discrimination of firewalled clients looking at the *Kad ID* and there is no server to check if a TCP port

can receive incoming connections: therefore a Kad client must start *firewall checks* on its own and set its state accordingly. However a client should be able to tell even if incoming UDP messages are blocked, a matter that will be discussed in-depth later.
As there is no server, firewalled clients that cannot accept incoming TCP connections need a *callback mechanism* similar to the eDonkey one. In this case a peer, called *buddy*, acts as a server to forward callback requests.

3.2.1 Routing table

The routing table, contained in every node, can be considered the core of the Kad network, as it manages links between peers across the network, thus defining the topology. It can be seen as set of contacts that the node has knowledge of, maintaining it by adding nodes, deleting the stale ones and keeping a structure that makes it efficient when used to access resources all over the network.
Contacts are stored by their kad ID, with a *type* variable (see table 3.1 for types, as defined in *eMule*) that evaluates their reliability based on how long they are known. eMule manages the routing table through three classes:

Table 3.1: Kad contact types

Type	Description
0	Contact active more than 2 hours
1	Contact active more than 1 hour, less than 2
2	Contact active less than 1 hour
3	Contact was just created
4	Contact is to be deleted

`RoutingZone`, `RoutingBin` and `Contact`. The routing table is made of different RoutingZones, which correspond to nodes in the tree and the trees leaves are also RoutingZones containing a RoutingBin each.
RoutingBin corresponds to k-buckets as described in section 2.2, where $k =$

10, where contacts are stored in `m_listEntries` which keeps contacts sorted by time they were last seen: older contacts are on the top, while newer ones or recently contacted are on the bottom. Contacts on the top are periodically contacted to determine if they are still active or they should be removed from the list. The routing table is a full binary tree: all nodes that are not leaves have two children; but it isn't a perfect binary tree with all leaves at the same depth. `RoutingZone` class represents a node in the routing table and it provides procedures[3] to maintain the routing table.
Contacts are inserted in the routing table according to their XOR distance from the node own Kad ID, contrary to Kademlia document that stores nodes by their Kad ID, thus simplyfing the implementation of source code and the illustration of the routing table. Nodes are inserted in the routing table starting from a routing table with only one routing zone (k-bucket) that is splitted respecting some constraints, in fact not all routing zones can split.

Listing 3.1: eMule CanSplit() method in RoutingZone

```
bool CRoutingZone::CanSplit() const
{
        // Max levels allowed.
        if (m_uLevel >= 127)
                return false;

        // Check if this zone is allowed to split.
        if ( (m_uZoneIndex < KK || m_uLevel < KBASE) && m_pBin->GetSize
            () == K)
                return true;
        return false;
}
```

As it is shown in listing 3.1 a routing zone can't split if we reached the last possible level (number 127, the 128th); also current routing zone must be a

[3]We should call them more appropriately methods, as eMule is written in *C++*.

leaf containing already k contacts, thus really requiring splitting. Last requirements are more interesting: all levels less or equal than 4 (so, till the 5th level) can split to form a perfect binary tree. The other requirement is more subtle: a zone on levels $5-127$ can split only if their position from the right hand of the tree on the level is less than 5 (only the first 5, from 0 to 4 allowed). To make things more clear we give an illustrated version of the routing table as it should be when it is completely filled with contacts. It is straightforward to see that a node with its routing table completely filled has a better view of his neighborhood than other parts of the network. Besides, we can calculate that the maximum number of contacts stored in the routing table is theoretically 6360, thus not requiring large amounts of memory and providing an efficient solution for the network needs.
However routing table requires maintenance to refresh stale nodes, eMule presents two different methods: `OnSmallTimer()`, that is run frequently and concurrently on more routing zones, while `OnBigTimer()` is run less frequently on just a routing zone at the time.
`OnSmallTimer()` is run, more often on k-buckets near own node's kad ID and it checks if contacts in the bucket are still active sending `HELLO` packets. `OnBigTimer()` is run every 10 seconds and its objective is to populate the routing table with new contacts, and it can run only on almost empty k-buckets or routing zones with k-buckets that can split (respecting the constraints defined above).

3.2.2 Bootstrap

To join the network properly, a node must populate its routing table by inserting peers and veryfing they are not stale, so that by contacting them, other peers can also insert the node in their routing tables.
Bootstrapping implies getting a list of nodes from somewhere outside the Kad network (most preferable from some trusted website) and use the list to populate the routing table. eMule reads from `nodes.dat` files, containing contacts, which comes in two flavors. Normal nodes.dat files have just

around 50 contacts which are directly inserted in the routing table, so if it was a file distributed to many clients for bootstrapping, it would lead quickly to *DDoS* because many nodes would rely on those 50 nodes for all the initial lookup procedures. A Bootstrap nodes.dat file instead contains a list of approximately 500-1000 contacts, that are not directly inserted in the routing table, but in a `bootstrapList` used to populate the routing table. Not all the nodes are chosen for the bootstrap, but just the 50 closest to the node's ID, therefore providing some randomness between different users, useful if many clients use the same file.
Nodes are saved at the end of a session for later use in a nodes.dat file where are stored maximum 200 contacts chosen from the routing zone using `GetBoostrapContacts()` method from RoutingZone class. The methods recovers contacts starting from the top routing zone and going down max 5 levels down the tree.

3.2.3 Firewall check

Joining the network, a node needs to find out if it is able to accept incoming TCP connection, and if it can exchange UDP datagrams with other peers. The initial checkup is started upon connection to the network and then repeated periodically every hour. A node requesting a firewall check sends a `KADEMLIA_FIREWALLED2_REQ` packets to known nodes, waiting for four `KADEMLIA_FIREWALLED_RES` packets, containing the IP address of the requesting client: a first check based on matching the received IP address and the one known, determines if the client is behind NAT[4]. Contacted nodes will try to estabilish a test TCP connection with the requesting node. If the connection is successful, a `KADEMLIA_FIREWALLED_ACK` packet is sent, and after receiving two of these packets, the requesting node sets its state not firewalled.

[4]Network Address Translation: a technique that modifies IP addresses and ports in IP packets transiting through a router.

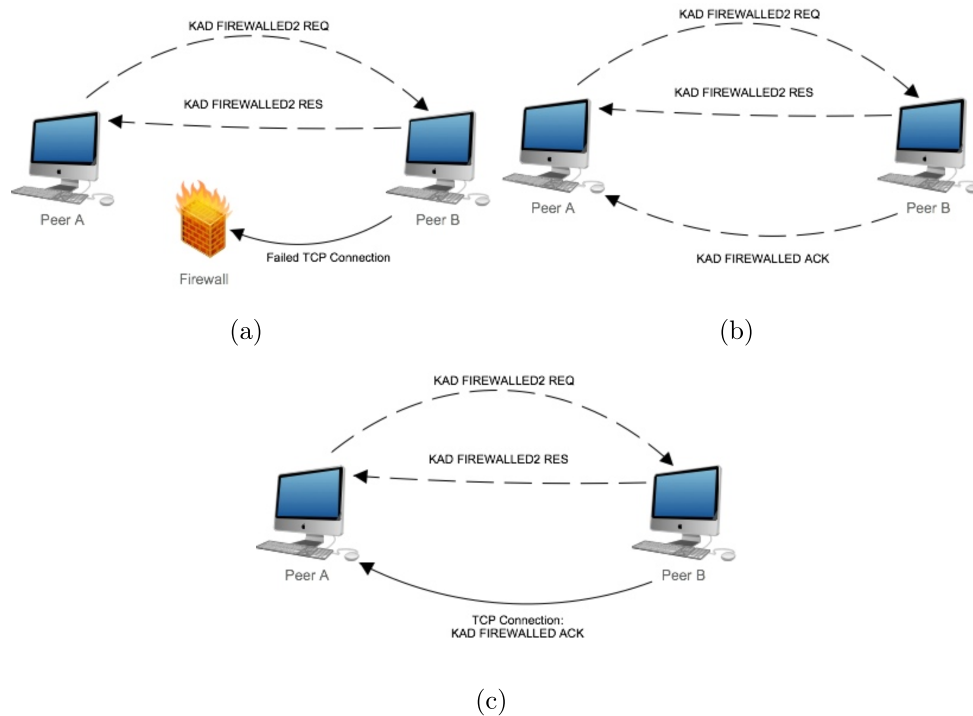


Figure 3.3: Firewall check mechanism: UDP datagrams are exchanged between firewall-checked peer (Peer A) and the peer checking for firewall (Peer B), then checking peer tries to estabilish a TCP Connection. If it fails, as in (a), Peer A knows it is firewalled. If it succeeds as in (b) and (c), an acknowledge packet is sent through UDP datagram or the estabilished TCP connection, it depends upon clients' versions.

3.2.4 FindBuddy (Callback)

A node that did a firewall check and resulted in a `firewalled` status, thus not being able to accept incoming TCP connections, needs a way to be contacted by other nodes that want to establish a connection. As there is no server as in eDonkey network, the Callback mechanism relies on buddies: each firewalled or non-firewalled node is allowed one buddy, acting as a client (firewalled nodes) or a server (non-firewalled nodes).
Finding a buddy consists in sending 3 `KADEMLIA_FINDBUDDY_REQ` packets

to the nearest peers, which will respond with a KADEMLIA_FINDBUDDY_RES packet if they are not firewalled. Once the buddy is found, when a node wants to establish a TCP connection with the firewalled client, it will send a callback request to the peer, which will in turn forward it to the firewalled client. The firewalled client will then establish the connection with the requesting client, with no further interaction with the buddy.

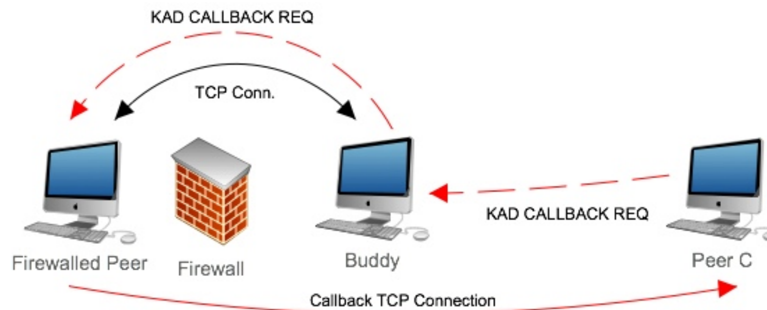


Figure 3.4: Callback Mechanism in Kad: Firewalled Peer can't receive incoming TCP connections, therefore relies on already estabilished TCP connection with its buddy to receive forwarded callback request (in red) and then calls back the requesting peer (Peer C).

3.3 Content management

Being Kad a file sharing peer-to-peer network, the main resource exchanged across peers are files. But, in order to be able to look for a file, clients should be able to find first which file they are looking for, and then they need to locate it in order to retrieve it. As files in the Kad network (as in eDonkey) can be downloaded from multiple sources at once, they are indexed by their content, to be sure that data we are downloading from different peers belongs to the same file.

A node searching for some content first starts a *keyword search* (e.g. "ubuntu

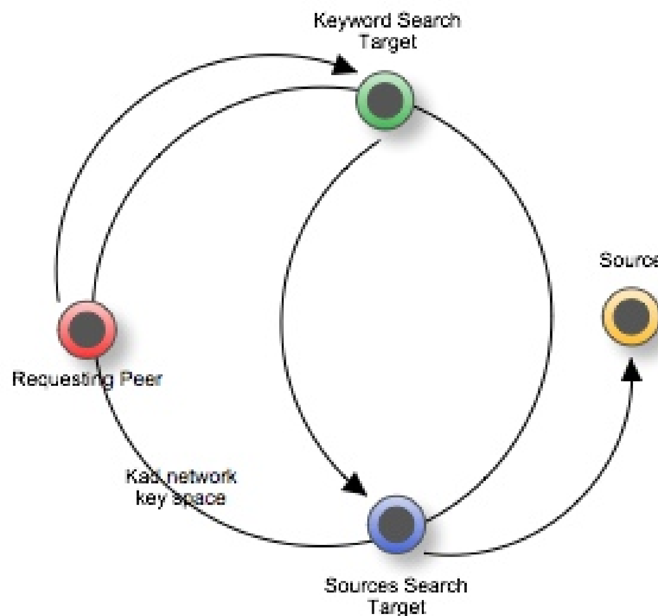


Figure 3.5: A requesting peer (red node) performs a *keyword search* on a target in the key space (an hash of a search string). Then, chosen a file to download, it performs a *source search* on another target (blue node, hash of the file) and finally it contacts sources to download the content (yellow node). Notice that sources, even if they are connected to the Kad network, are not required to, as file exchange is not a Kad related activity.

iso"), which will return a list of files related to keywords[5] searched. A list of files is then returned from peers responsible for the keywords, with different 128-bit kad ID that identify them. Chosen a file to download, a *source search* is performed and nodes responsible for the indexing of the file will return a list of peers that serve the file. At last, the node can contact directly those peers to download content.
A file has a 128-bit *kad ID* and is also referenced by a *ed2k ID* that is the same, so files are referenced by the same identifier in both networks: this allows to search for some keyword on the Kad network and then find sources

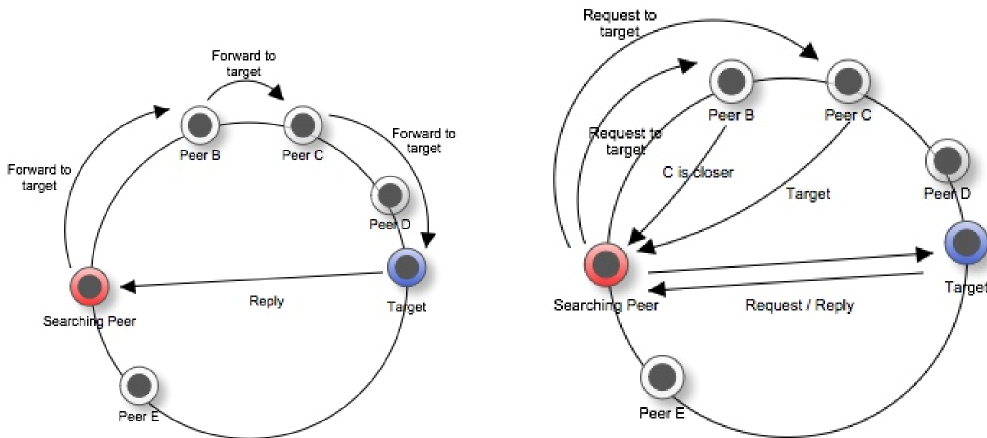
[5]For now we can consider the keyword a string of alphanumerical characters.

on both networks, as long as the file was published in both networks. Viceversa, it is possible to do a keyword search against an ed2k server and lately search for sources on Kad network.
So the resources shared on the Kad network are: *file references* found with keyword searches, and *sources* found with source searches. As in eDonkey, peers are allowed to "comment" files, so in Kad too a third resource is represented by *notes*, found with note searches. In order to retrieve all those resources someone must have indexed them on the nodes before: nodes sharing those resources *publish* them onto responsible peers, so there are keyword, file and note publications.

3.3.1 Lookup

Search, publication and even routing zone population tasks all share a common procedure that locates in the network nodes responsible for resources: the *lookup*.
Lookup is a procedure that iteratively locates closer nodes to a target kad ID: closer nodes have a better chance of being responsible for resources indexed by the target ID, thus through the lookup we should get a list of nodes close to the resource we want to ask, so that we may then perform *search actions* or *publishing actions* on those nodes. We may also just want to locate some nodes somewhere in the network, as done in routing table maintenance tasks: in that case only the lookup phase is performed, and nodes added to the routing table.
Lookup works similarly to the protocol for *FIND_NODE* and *FIND_VALUE* RPCs described in section 2.2, it is an iterative process in the sense that our node iteratively approaches the target asking for closer nodes to the target and contacting the closest at each step. A recursive way, where nodes would be in charge of forwarding the request through closer nodes (in their routing tables), would be more efficient in terms of message overhead and time, but less reliable as the node originating the request has no mean to control the

process, and the reliability of the process should be managed by all nodes taking part. Lookup terminates when closest nodes do not return any closer



(a) **Recursive lookup** forwards packets through nodes towards the target. It is not reliable as responsibility for lookup process relies on all the nodes, if one node doesn't forward the message, the process fails.

(b) **Iterative lookup** is handled by the requesting peer, responsible for the whole process. It requests information to closer nodes but mantains control of the system. It is robust to node failures.

node to the target. Kademlia original document [1] proposed an α parameter to exploit concurrency in the lookup process: sending requests to more nodes at once can protect from stale nodes and reduce considerably lookup time. The document proposed to manage the lookup as α independent threads, but eMule uses a smarter approach, defined *loose concurrent node lookup* in [3]: using a single thread, lookup starts choosing 50 *possible contacts* with minimum distance from the target and sending the request to α closest contacts, and once received the first reply, sends a new request to the new closest contacts, thus maintaing α active requests at all time.

3.3.2 Search

As we said, it is possible to locate different resources on the network using *keyword search*, *source search* and *note search*.

Keyword search

A keyword search is more exactly a search for files published on the network that are referred to some string that is object of the search. As all the resources we look for in the network must be mapped on the 128-bit key space, keywords follow this rule too: the target of the lookup is the *MD4 hash* of the longest word in the search string.

> **Ex.** If we search string *"sigur ros hoppipolla"*, eMule starts a lookup with target `D9902A5F0B69C73E2BA3E767BE20C95F`, that is MD4 hash of "hoppipolla" word.

Once the lookup terminates, we have a number of nodes close to the target ID, those nodes are probably responsible for the keyword and a `KADEMLIA2-_SEARCH_KEY_REQ` packet is sent to them. This packet not only contains the keyword, but all the search string and eventually parameters to filter search. The receiving peer, if it is responsible for some content published by a third party, will look for the keyword in its indexed contents, and then reply with a `KADEMLIA2_SEARCH_RES` packet, which is generic for all kinds of searches, the differences reside in the `TagList` contained in the packet, a list of tags describing each file result, complete with information like the kad ID, filename, file type, size and other more format specific tags (such as artist or album for music). The list of tags is processed by the instance of `CSearch` that manages the search, through `ProcessResultKeyword()` method. Moreover, as the request and response packets contain kad ID referring to the search, it is possible to launch more concurrent searches, while in eDonkey network only a search is allowed at once towards a server.

Source search

Searching for sources for a file is more straightforward, as the target of the search is the file hash of the file. Hypotetically speaking, we could calculate

the MD4 hash of a file and try to look for the same file on the Kad network just by running a source search targeting the 128-bit hash, this is a stand-alone process, independent from keyword search.
Source search will start with a lookup and, exactly as keyword search, when a list of nodes that should be responsible for the file sources is known, the requesting client will send `KADEMLIA2_SEARCH_SOURCE_REQ` containing the file size (which is returned during keyword search in the tag list). As expected, the receiving nodes will respond, if responsible for the file asked, with `KADEMLIA2_SEARCH_RES` containing a tag list that will be processed by the method `ProcessResultFile` of the relative `CSearch` instance, and results added to the download queue for the file (as source search typically is performed during file download).

Note search

As for source search, note search is targeted to a specific file identified by its MD4 hash in the same way. Once the lookup is done `KADEMLIA2_SEARCH-_NOTES_REQ` packet (containing the file size as `KADEMLIA2_SEARCH_SOURCE-_REQ`) is sent to close nodes, which will respond with `KADEMLIA2_SEARCH_RES` packet, where the `CSearch` instance that takes care of the search will process the results through the method `ProcessResultNotes`. The tag list in this case is a list of comments and ratings for the file.

A search terminates when enough results are retrieved or it goes on for too long, according to constants defined for each kind of search: keyword search stops after 300 results (`SEARCHKEYWORD_TOTAL`), source search stops after 300 results too (`SEARCHFILE_TOTAL`) and notes search stops after just 50 results (`SEARCHNOTES_TOTAL`) while all kinds of search have a maximum duration of 45 seconds (`SEARCHKEYWORD_LIFETIME` and other constants named accordingly).

3.3.3 Publishing

In order to index resources in the nodes for later retrieval, nodes that want to share files and related resources must publish them. In the eDonkey network, a client joining the network sends an *OfferFiles* packet to the server[6], with all the information needed to retrieve the file. The server itself will index both the keywords for the files shared and the source for the referenced file, it can also be aware of when the source will disappear in case the connection with the offering client drops.
In Kad, as there is no server, the publishing scheme is a bit more sophisticated. A Kad client joining the network must publish the keywords for its own files and publish itself as a source in two different processes. As if it was not enough, it should take care of grouping references to the same keyword as to avoid sending many messages to the same nodes for the same keyword.

> **Ex.** If a client wants to share three different files "kernel_linux.tgz", "slackware-linux.iso" and "linux_for_dummies.pdf" it must first find out all the keywords that it will publish. In this case it will publish references for keywords: "kernel", "slackware", "dummies", etc. as keywords with a single file entry, and then the "linux" keyword will be published with references to the three different files. Notice that a single file may be referenced in many different keyword publishing.

As it is possible to search for notes, it is also possible, as an owner of a file, to publish comments. Publishing of resources does not last forever: as written in the original Kademlia document contents need to be republished because publishing expires. eMule source code accordingly to the document set expiration for keywords and notes after 24 hours (see constants `KADEMLIAREPUBLISHTIMEK` and `KADEMLIAREPUBLISHTIMES`), while source publishing expires just after 5 hours (see constant `KADEMLIAREPUBLISHTIMEN`).

[6] To be more precise, if the client shares more than 150 files, a number of different OfferFiles packets is sent, delayed to avoid overloading the server.

eMule also defines how many resources can be published before rejecting pub-

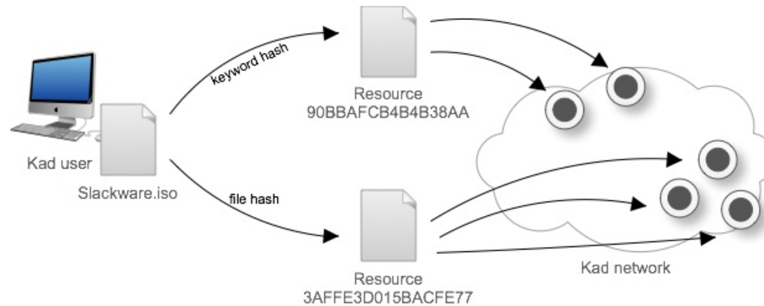


Figure 3.6: Publishing: when a file is to be published by a *Kad user*, he first hashes both keywords and the file itself, and then proceeds to separately publish those resources (all the hashed keywords related to the file) and the hashed file on the network, storing references on a set of peers that will then be responsible for the resources.

lishing requests: a client can index up to 50000 keyword (`KADEMLIAMAXINDEX`) and for each file published a maximum of 1000 sources (`KADEMLIAMAXSOUCE-PERFILE`) and 150 notes (`KADEMLIAMAXNOTESPERFILE`). Publishing can be seen as a kind of search, where the final action is a *store* request, different in case we are publishing a keyword (`KADEMLIA2_PUBLISH_KEY_REQ`), a source (`KADEMLIA2_PUBLISH_SOURCE_REQ`) or a note (`KADEMLIA2_PUBLISH_NOTES-_REQ`). In each case a lookup is performed and then, once a list of nodes close to the target are known, requests will be sent. Nodes receiving the requests will first check if they are entitled to be responsible for such resources, as only contacts in the *tolerance zone* can control a resource. If so, they check other contraints and will eventually index the requested content, replying with a `KADEMLIA2_PUBLISH_RES` packet containing the load of the node for the kind of resource that needs to be indexed. Load is a value between 0 (empty) and 100 (full, request rejected) calculated as shown in table 3.2 The publishing process terminates when enough nodes have successfully stored our resources, that is to say replying with a $load < 100$. The number of nodes is 10 for keywords, notes and files (as defined by constants `SEARCHSTORE-`

Table 3.2: Load calculation

Resource type	Equation	Max stored
Keyword	$\frac{\texttt{uIndexTotal}}{\texttt{KADEMLIAMAXINDEX}} * 100$	50000
Source	$\frac{\texttt{uSize}}{\texttt{KADEMLIAMAXSOUCEPERFILE}} * 100$	1000
Note	$\frac{\texttt{uSize}}{\texttt{KADEMLIAMAXNOTESPERFILE}} * 100$	150

KEYWORD_TOTAL, SEARCHSTORENOTES_TOTAL and SEARCHSTOREFILE_TOTAL). Furthermore, if the publishing process takes to long, it is terminated after 140 seconds for keyword and source publishing (SEARCHSTOREKEYWORD_LIFETIME and SEARCHSTOREFILE_LIFETIME), while notes terminate after 100 seconds (SEARCHSTORENOTES_LIFETIME).

3.4 Advanced features

This section briefly describes protocol obfuscation for Kad traffic and 64-bit support.

3.4.1 Protocol obfuscation

Kad network supports encryption of packets exchanged between clients in order to obfuscate messages to prevent filtering by routers and protect privacy of users. Obfuscation works in different ways in TCP connections and UDP datagrams exchange. Peers have a 32-bit KadUDPKey that they do exchange following Diffie-Hellman[7] key exchange protocol and they use to communicate between them through UDP datagrams.

[7] http://www.lsi.upc.edu/ diaz/diffie.hellman.pdf

3.4.2 64-bit extension support

Original eDonkey network supported files sized up to 4 GB (2^{32} bytes). In eMule was later added 64-bit file-size support, and obviously Kad network supports 64-bit file-size too. Only `KADEMLIA2_SEARCH_SOURCE_REQ` and `KADEMLIA2_SEARCH_RES` packets contain file size: in the first case as a packet field 64-bit long, and in the second case as a tag.

Chapter 4

Kad Implementation

In the previous chapter we presented the Kad network, going deeper analyzing eMule implementation[1]. As eMule was developed in *C++*, tightly integrated with the user interface, it required us practically to write a client from scratch, in fact our software needs to be written in Java to be compatible with PariPari structure. Another implementation of a eDonkey/Kad client written in Java exists, it is called *jMule* but due to license incompatibilities with our software, the possibility of a merge or collaboration has been so far considered.

Even if our client is far from being complete, it is quite reliable and performs good especially on the eDonkey network. Kad support has been developed first as a stand-alone client, that has been later integrated in *PariMulo* merging some classes, while eMule maintains a separate folder with Kademlia source code. A number of classes are used both to support compatibility with eDonkey and Kad networks, while only a few are network-specific, and some classes are used to interface with PariPari. In table 4.1 there is a list of *Java classes* fundamental for supporting connectivity to the Kad network, while table 4.2 lists and describes briefly other classes used for Kad, but not strictly Kad-related as they are used for eDonkey network support too.

In this chapter we describe the inner workings of PariMulo plug-in for what

[1] Which, to a certain extent, coincides with aMule implementation.

Table 4.1: Kad-only PariMulo classes

Class	Description
`Bootstrap`	Manages bootstrapping process when joining the network.
`ClientList`	Manages a list of peers, used for FindBuddy mechanism.
`Int128`	Takes care of all address related stuff, calculate distances, does conversions.
`Kad`	Starts the Kad network support in PariMulo.
`KadContact`	Represents a single node in the Kad network.
`KadIndexed`	Stores contents published by other peers.
`KadPrefs`	Contains static functions to set Kad firewall state.
`KadSearch`	Manages all kinds of search in Kad network.
`KadUDPKey`	Represents key used for UDP datagrams obfuscation.
`Lookup`	Used for KadSearch, handles lookup mechanism.
`RoutingBin`	Represents a leave in routing table, contains a list of KadContacts.
`RoutingZone`	Represents a node in routing table, keeps static reference to tree's root.

it concerns Kad network support. As it was described in the previous chapter there are many features to implement, and we tried our best to achieve a rational design exploiting object-oriented nature of Java and its advanced functionalities (*e.g* concurrency, data structures, networking framework, etc.). Looking at table 4.1 and 4.2 we may divide classes into groups. First, classes that manage connectivity to the network and defines the message protocol on top of UDP (`Packets`, `PacketsUDP`, `Tags`, etc.). Then, there are classes that build, manage and maintain the overlay network over the IP network (`RoutingZone`, `RoutingBin`), these classes are crucial and even little bugs could spread and modify the behavior of all the network. A third group is represented by classes used for lookup procedure and classes that use

lookup, that is to say classes that implement search and publishing (`Lookup`, `KadSearch`, etc.). Lastly, there are classes that support indexing of files published by other nodes (`KadIndexed`), FindBuddy mechanism and protocol obfuscation for UDP communications.

Table 4.2: PariMulo classes used for Kad-related activities.

Class	Description
`Config`	PariMulo configuration, contains some Kad parameters.
`Connection`	Handles TCP connections to nodes and peers.
`Download`	Manages the download of a file.
`DownloadManager`	Manages all the downloads.
`DownloadSession`	Manages the download of a file from a single peer.
`Hashes`	Represents various hash functions (MD4, MD5, SHA-1).
`Packets`	Handles packets decoding both for TCP and UDP.
`PacketsUDP`	Contains all classes representing UDP packets.
`Peer`	Represents a peer that can serve us a file.
`RC4Engine`	Used for protocol obfuscation, handles encryption / decryption.
`Tags`	Encodes and decodes tags in search results and other packets.
`TaskManager`	Manages tasks that are performed periodically.
`UDPListener`	Listens to incoming UDP traffic and provides methods to send packets.
`Utils`	Contains a set of methods used by various classes (e.g. conversion methods).

4.1 Communications

Communications between peers in Kad are sent as *kad packets* over UDP datagrams. `UDPListener` class is in charge of sending and receiving UDP

datagrams, thus hiding the management of the socket, buffers and the thread, with upper level classes working directly on kad packets.
`Packets` is a fundamental class to decode kad packets from UDP datagrams, building objects that can be easily manipulated. It also decodes compressed packets and obfuscated packets (when protocol compression and obfuscation are used). `PacketsUDP` actually contains the definition of all the packets, including eDonkey and eMule packets, with methods to encode and decode them from raw bytes.

4.1.1 UDPListener

The class starts a thread (more correctly a PariPari class that wraps a `Thread`, called `PariPariThread`), that handles a socket bound to an UDP port (`Config.udpPort`). This class manages generic UDP datagrams, dividing eDonkey/eMule/Kad packets if more packets are contained in a single datagram (by matching the same 2 byte header multiple times, as only same kind packets are sent in the same datagram), and providing *static send methods* that are called directly while for reception a more subtle mechanism is used.
The most important part of the class resides in the `parseDatagram` method that illustrates, as shown in listing 4.1, how received datagrams are managed, first decoded through `PacketUDP.decodePacket` and then if packet was decoded successfully, its `onReceive` method is called and according to the return boolean, the decoded packet is added to `receivedPackets` list for later use. Returning `true` should be used carefully, if a large number of packets is received and never discarded flushing the list, could lead to a great waste of memory.

Listing 4.1: Portion of parseDatagram() in UDPListener

```
packet = PacketUDP.decodePacket(Utils.arrayCopy(buffer, previous, length −
    previous), ip, port, false);
if (packet != null) { // otherwise it is seriously malformed
```

```
        PPLog.printIncoming("Received " + packet.getClass().getSimpleName() +
            " (0x" + Utils.byteToHex(buffer[previous + 1]) + ")", packet, buffer, ip
            .getHostAddress() + ":" + port);
        Mulo.udpBytesIn += packet.size + PacketUDP.HEADER_SIZE;
        Mulo.udpPacketsIn++;
        if (!packet.onReceive()) { // if it's not elaborated on the fly...
                receivedPackets.add(packet); // it is put on the list of unprocessed
                    packets
        }
}
```

4.1.2 Packets

Packet decoding and encoding, for what it concerns UDP communications, is managed by `Packets.java` and `PacketsUDP.java`, while `PacketsTCP.java` is used to define packets exchanged over TCP streams, therefore it doesn't concern Kad.

Packets.java

`Packets.java` contains the abstract class `Packet` which is extended in the same file by classes `PacketUDP` and `PacketTCP` where `decodePacket()` method takes the array of bytes and is asked to call the right constructor choosing from all the classes that extend `PacketUDP` in `PacketsUDP.java`.

Listing 4.2: Portion of decodePacket() from PacketUDP in Packets.java

```
final static PacketUDP decodePacket(byte[] originalPacket, InetAddress ip, int port,
    boolean calledByObfuscation) {
        if (originalPacket == null || originalPacket.length == 0) {
                return null;
        }
        PacketUDP decodedPacket;
```

```
boolean compressed = false;
if (originalPacket.length >= HEADER_SIZE) { // long enough to get
    protocol and type
        // get the right subclass constructor for this protocol & type
        Constructor<?> constructor; // = null;
        byte[] packet = unobfuscatePacket(originalPacket, ip);
        if (packet[0] == Protocol.COMPRESSED.id || packet[0] ==
            Protocol.KAD_COMPRESSED.id) {
                packet = unpackPacket(packet);
                compressed = true;
        }
        if (packet[0] == Protocol.ED2K.id) {
                constructor = ed2kTypeConstructors[packet[1] & 0xFF];
        } else if (packet[0] == Protocol.EMULE.id) {
                constructor = eMuleTypeConstructors[packet[1] & 0xFF];
        } else if (packet[0] == Protocol.KAD.id) {
                KadContact.setLastContact();
                constructor = kadTypeConstructors[packet[1] & 0xFF];
        } else {
                // No constructor found <- unreachable point...
                    theoretically
```

As it is shown in listing 4.2, an array of bytes representing an eDonkey / eMule / Kad packet received as an UDP datagram first checks if it can be a valid packet (being non-null and sufficient length), then `unobfuscatePacket()` method tries to decode the packet if it is obfuscated (*encrypted*), returning in both cases an array of bytes containing the *plaintext message*. Now, as we are dealing with the array of bytes `packet` containing the plaintext message, we can examine the header's first byte (`packet[0]`) to determine if the packet is compressed, if so `unpackPacket()` is called, which returns the uncompressed array of bytes and examining again the first byte it is chosen whether the packet concerns eDonkey, eMule or Kad communications.
Finally, the right constructor to build the object representing the packet type

that extends `PacketUDP` can be chosen examining the header's second (and last) byte (`packet[1]`) and if a constructor is found, it will be called using *Java reflection* package.

PacketsUDP.java

`PacketsUDP.java` contains the definition of all the UDP packets, including eDonkey / eMule ones, with methods for decoding, encoding, printing and creating them. As an example we show a single packet type `PacketUDPKad-BootstrapRequest` in listing 4.3. There are a list of *instance variables* that define the contect of this type of packet: it contains an IP address, a Kad ID, two ports, etc.
There are two constructors: the one accepting as parameters variables that will be set as instance variables (`PacketUDPKadBootstrapRequest(Int128 id, ..., byte version)` for instance) is used to create an *outgoing packet*. A packet created with this method can be sent using `UDPListener` *send()* method, which will in turn call the class *toBytes()* method that has the duty of creating an array of bytes with the content of the packet in the order described by the protocol.
The second constructor (`PacketUDPKadBootstrapRequest(byte[] packet, ..., int port)`) is used to decode *incoming packets* as it reads directly from the array of bytes the instance variables in the order described by the protocol, using superclass `Packet` read methods.
The *onReceive()* method is useful to execute some code upon receiving a specific packet type: it is called just after the creation of the packet from `UDPListener` and if the method returns `false`, the packet is stored for later use. The common *toString()* method offers instead a nice way of printing the packet contents: it should be used for logging and debugging purposes.

Listing 4.3: Class PacketUDPKadBootstrapRequest in PacketsUDP.java

```
class PacketUDPKadBootstrapRequest extends PacketUDP {
        InetAddress ip;
        Int128 clientID;
```

```
int UDPPort;
int TCPPort;
byte kadVersion;

PacketUDPKadBootstrapRequest(byte[] packet, InetAddress ip, int port) {
        super(KadType.KAD_BOOTSTRAP_REQUEST);
        this.clientID = this.readInt128(packet);
        this.ip = this.readIPAddress(packet);
        this.UDPPort = this.readPort(packet);
        this.TCPPort = this.readPort(packet);
        this.kadVersion = this.readByte(packet);
}

PacketUDPKadBootstrapRequest(Int128 id, InetAddress address, int
    udpPort, int tcpPort, byte version) {
        super(KadType.KAD_BOOTSTRAP_REQUEST);
        this.clientID = id;
        this.kadVersion = version;
        this.ip = address;
        this.UDPPort = udpPort;
        this.TCPPort = tcpPort;
}

@Override
boolean onReceive() {
        return false;
}

@Override
ByteBuffer toBytes() {
        ByteBuffer buffer = super.toBytes();
        buffer.put(this.clientID.toBytes());
        buffer.put(Utils.arrayReverse(this.ip.getAddress()));
```

```
        buffer.putShort((short) this.UDPPort);
        buffer.putShort((short) this.TCPPort);
        buffer.put(this.kadVersion);
        return buffer;
    }

    @Override
    public String toString() {
        return super.toString() + " | Client ID: " + this.clientID + " | IP:
            " + this.ip.getHostAddress() + " | UDP/TCP Port: " + this.
            UDPPort + "/" + this.TCPPort + " | Kad Version: " + Utils.
            byteToHex(this.kadVersion);
    }
}
```

4.2 Building the network

Kad is an overlay network over the IP network, where each peer knows a set of other peers selected accurately and put in a convenient structure so that even with millions of peers connected to the network it works well. Kad IDs are used to identify peers and resources in the network and the network itself is based on *XOR metric*, `Int128` is useful to represent IDs and do necessary calculations. All the peer inserted in the routing table are stored as `KadContact` objects (while peers used for file exchange are `Peer` objects). *Routing table* doesn't exist as a class itself, it is instead made of many `RoutingZone` objects (with a static reference to the root of the tree), representing "nodes" in the Routing table tree, while `RoutingBin` objects represent leaves.

4.2.1 Kad ID (Int128)

Int128, as the name suggests, can be seen simply as a 128-bit integer, but Java does not support directly this as a *primitive type*. The class has a single instance variable, a `BitSet` set final, and constructor and methods to handle easily these objects. Probably the most interesting part of the class is the constructor, as shown in listing 4.4. Int128 are sent in Kad packets as four 32-bit integer in *Little Endian*[2] format, these movements to store the BitSet are done just for conveniency so that other operations can be easily written in Java.

Listing 4.4: Int128 constructor

```
Int128(byte[] bytes, boolean isInLittleEndianByteOrder) {
	if (bytes.length > SIZE) {
		throw new IllegalArgumentException("array di max 16 bytes = 128
			bits");
	}
	this.bitSet = new BitSet(SIZE * 8);
	if (isInLittleEndianByteOrder) {
		byte[] reversedArray = new byte[16];
		for ( int i = 0 ; i < SIZE ; i += 4 ) {
			reversedArray[i + 3] = bytes[i + 0];
			reversedArray[i + 2] = bytes[i + 1];
			reversedArray[i + 1] = bytes[i + 2];
			reversedArray[i + 0] = bytes[i + 3];
		}
		bytesToBitSet(reversedArray, this.bitSet);
	}
	else {
		bytesToBitSet(bytes, this.bitSet);
	}
}
```

[2]TODO spiega endianness

4.2.2 KadContact

`KadContact` objects represent peers known through the Kad network. Among the main instance variables there are two `Int128`: *id* and distance, that are respectively the kad ID and the *XOR distance* from our own node. Then obviously there is a triple made of IP address (*ip*), UDP port (*UDPPort*) and TCP port (*TCPport*) to contact the node. A byte represents *type* as described in table 3.1 of the previous chapter and *kadVersion* byte discriminates only two versions of Kad protocol support. Lastly, there is a `KadUDPKey` used for obfuscation of UDP datagram exchanges with the contact.

4.2.3 Routing table

Routing table is a very delicate part of a Kad client, and eMule developers went as far as writing comments in eMule source code to point out that no modification should be made unless absolutely necessary. The problem is that even if a single client modifies how routing table stores nodes it does not affect the network, and in the worst case our client would not work and be treated as a malicious peers, if many clients behave badly, the network is affected.
However, as of this writing the number of nodes running our client is very limited thus even buggy classes would not harm the network, anyway our implementation is very similar to eMule one for full compatibility.

RoutingZone

As it was said, *routing table tree* is made of routing zones that represent nodes, and the only way to traverse the tree is to get the root node using *private static variable root*. As it is shown in figure 4.1, each `RoutingZone` has a *parent* `m_pSuperZone` and either zero or two children (`m_pSubZones[0]` and `m_pSubZones[1]`). The class contains a static reference to own node `Int128` *uMe* and all the contacts are sorted in the tree by their *XOR distance* from *uMe*.

Each routing zone has also two instance variables that define its position in the routing table: *m_uLevel* that indicates the level from the top of the tree, and *m_uZoneIndex* that contains the distance from the zone at this level that contains the center of the system. `RoutingZone` provides methods

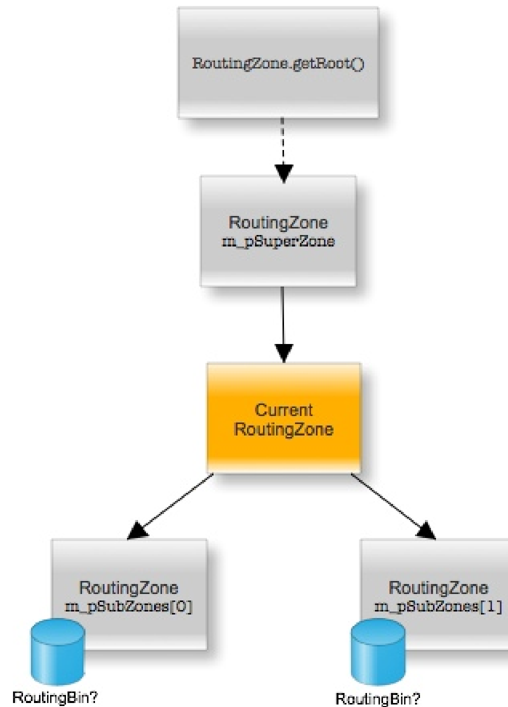


Figure 4.1: RoutingZone relationships: it is highlighted that only leaves may have `RoutingBin` objects, while a reference to the root RoutingZone is provided to traverse the tree.

to insert new contacts and get contacts, with the ability of automatically splitting and join (*consolidate*) routing zones when needed. In fact, each leaf can contain a maximum number of ten contacts, then it should be splitted when trying to add an eleventh contact (calling *addToZone()*). Routing zones are instead checked for and eventually *consolidated* periodically (every 45 minutes, `m_tConsolidate_millis`). We show in listing 4.5 the *split()* method, that together with *consolidate()* define the structure of the routing

table, as only some routing zones are allowed to split.

Listing 4.5: split() method in RoutingZone.java

```
private void split()
{
        this.stopTimer();
        this.m_pSubZones[0] = this.genSubZone(0);
        this.m_pSubZones[1] = this.genSubZone(1);
        List<KadContact> listEntries = this.m_pBin.getEntries();
        this.m_pBin = null;
        for(KadContact pContact : listEntries) {
                assert(pContact!=null);
                int iSuperZone = pContact.distance.getBit(this.m_uLevel);
                if (!this.m_pSubZones[iSuperZone].m_pBin.addContact(pContact)
                    ) {
                        this.m_pBin.removeContact(pContact);
                        PPLog.printError("Error in RoutingZone split() method!");
                }
        }
}
```

The method does not include *canSplit()* method that is run before, but just generates children routing zones and moves contacts to the new leaves.

RoutingBin

Each `RoutingZone` has a `RoutingBin` *m_pBin* instance variable that is used only when the routing zone represents a leaf in the tree, and in case of splitting, contacts in its routing bin are added to children's routing bins and parent's bin is nulled. On the contrary when consolidating leaves, routing bin's contacts are merged in a single bin on the parent, children routing zones (and thus their bins) are nulled.
Each routing bin is just a holder for a `LinkedList` of `KadContacts`, that contains up to ten contacts sorted chronologically.

Maintenance tasks

Maintaining the routing table is an important duty that is done when adding and refreshing contacts obtained from normal lookup and search procedures, but also with specific *maintenance tasks* run periodically. Just like eMule, our client provides two different methods, `onSmallTimer()` and `onBigTimer()`, that remove *stale contacts*, check old contacts and add new ones.

Listing 4.6: onSmallTimer() method in RoutingZone.java

```
public void onSmallTimer() {
if (!this.isLeaf()) {
        return;
}
long tNow_millis = System.currentTimeMillis();
List<KadContact> listEntries=this.m_pBin.getEntries();
if(listEntries==null || listEntries.size()==0) {
        return;
}
// Remove dead entries
for(KadContact aContact: listEntries) {
        if ( aContact.type ==4) {
                if (((aContact.expirationTime_millis > 0) && (aContact.
                    expirationTime_millis <= tNow_millis))) {
                        this.m_pBin.removeContact(aContact);
                        KadContact.removeContactGeoLocation(aContact);
                        KadContact.totalNumOfContacts--;
                        continue;
                }
        }
        if(aContact.expirationTime_millis == 0) {
                aContact.expirationTime_millis=tNow_millis;
        }
}
// Check if the oldest contact of the list is still alive
```

```
KadContact oldContact = this.m_pBin.getOldest();
if( oldContact != null ) {
        if ( oldContact.expirationTime_millis >= tNow_millis || oldContact.type
            == 4) {
                this.m_pBin.pushToBottom(oldContact);
                oldContact = null;
        }
}
if(oldContact != null) {
        oldContact.checkingType();
        PacketUDP pck_KAD2=new PacketUDPKad2HelloRequest();
        UDPListener.send(oldContact.ip, oldContact.UDPPort, pck_KAD2);

}
```

`onSmallTimer()` as it shown in listing 4.6 is called for a single `RoutingZone`, and assuming it should be called just on leaves (routing zones containing a bin), it remove old contacts (type 4) and contacts the oldest one in the bin, which will eventually respond and refresh itself.

Listing 4.7: onBigTimer() method in RoutingZone.java

```
public boolean onBigTimer() {
        if ( this.isLeaf() &&
                ( this.m_uZoneIndex.isAllowedToSplit(K_POSIZIONE) ||
                this.m_uLevel < K_LIVELLO || this.m_pBin.getRemaining() >=
                    8 )) {
                        this.randomLookup();
                        return true;
        }
        return false;
}
```

`onBigTimer()` tries to populate routing zones that have few nodes and can split by launching a lookup towards a random node in the current zone. This process is called every ten seconds on one routing zone at once, and every one hour on each of the routing zones. That is to say that if we have just one routing zone (a leave), the process won't run every ten seconds, but just once in an hour.

4.3 Lookup

Lookup is an important procedure to locate a number of nodes close to a kad ID, and its correctness, robustness and efficiency greatly affects all the Kad network experience. For instance, if routing table fails, a node can use other nodes' routing tables and work correctly, however, if a lookup fails, a node can't obtain resources from the network, thus rendering the network useless. That is to say that for a Kad client it is crucial to implement correctly lookup procedure.
Differently from eMule, our implementation defines a class `Lookup` that is tightly bound to `KadSearch`, used for *search* and *publishing*, while eMule handles lookup in the same classes of search (*Search.cpp* and *SearchManager.cpp*). Our choice was to give a more elegant and clean structure to code, and avoid repeating the same code in different methods. Even if lookup procedure vary slightly for different kind of searches, all the differences can be dealt including a `lookupType` instance variable.
It is somehow surprising for novices that lookup works even if the *start()* method, that manages all the procedure, just sends packets and never receives them or wait for them. In fact, packets are received by `UDPListener` and processed by the corresponding packet class in `PacketsUDP.java`. The *onReceive()* method that is executed upon packet arrival, will call back the `Lookup` class through *addContacts()* method, which will find the corresponding lookup, add the contacts and wake up the probably sleeping lookup thread. This mechanism looks exactly like the one implemented in eMule,

and it's a great way to simplify the code, to the point that we could even think of running just one thread for all the lookups.

Listing 4.8: addContacts() method in Lookup.java

```
public static void addContacts(Int128 target, InetAddress sender, LinkedList<
    KadContact> contacts) {
        Lookup l = findLookup(target);
        Request r = null;
        if (l != null) {
                r = findRequest(l, sender);
        }
        if (r != null && l != null) {
                if (l.lookupType == KadSearch.KadType.FILE ||
                l.lookupType == KadSearch.KadType.KEYWORD ||
                l.lookupType == KadSearch.KadType.NOTES ||
                l.lookupType == KadSearch.KadType.STOREKEYWORD ||
                l.lookupType == KadSearch.KadType.STOREFILE ||
                l.lookupType == KadSearch.KadType.STORENOTES) {
                for (KadContact c: contacts) {
                        long distance = Utils.distance(c.id, target);
                        RoutingZone.add(c, true);
                        if (distance < Config.toleranceZone) {
                                l.toleranceContacts.add(c);
                                l.possibleContacts.add(c);
                        } else {
                                l.possibleContacts.add(c);
                        }
                }
                }
                else if (l.lookupType == KadSearch.KadType.NODE) {
                        for (KadContact c: contacts) {
                                RoutingZone.add(c, true);
                        }
```

```
            }
            MuloThread.notify(l.target);
        }
}
```

In listing 4.8 is shown how new contacts are dealt by the `Lookup` class: first it is checked if a lookup was active towards the target or the packet received is related to an old lookup and thus useless, or more probably a *malicious node* wants us to store *bogus nodes*. For the same reason a check, to match the sender of the received packets with a previously sent request, is done and only if both checks pass, some steps are taken based on the type of lookup: contacts received from a *node lookup* (`KadSearch.KadType.NODE`), called to populate some routing zone, are directly stored in the routing table. Other kind of lookups, the ones used to locate some resources on the network, need to go closer to the node: given an `Int128` *target* a *tolerance zone* is defined as a range where peers probably know resources located at *target*.

Listing 4.9: distance() method in Utils.java

```
    static long distance(Int128 target, Int128 source) {
            Int128 xorValue = Int128.XOR(target, source);
            return xorValue.get32BitChunk(0);
    }
```

`Config.toleranceZone` is defined as 2^{24}, and distance between `Int128` objects, as shown in listing 4.9, concerns only the first 4 bytes, thus an x texttttInt128 is in the tolerance zone of *target* `Int128` if and only if the first byte coincides (*XOR* sets the first 8 bit to zero).
Contacts received for *resource locating lookups* like `KadSearch.KadType.-KEYWORD` are added to the routing zone for later use and also added to *possibleContacts* list, which contains nodes that can be asked for closer nodes to the target. The lookup terminates depending on the type, with *resource lo-*

cating lookups ending when we received at least ten contacts in the tolerance zone or a time-limit has been hit (50 seconds).

4.4 Search

As it was described in the previous chapter, there are various kinds of search: *keyword search*, *source search* and *note search*. These are search in a strict sense, in fact *node search* and *publishing* differ slightly from traditional search. We present in table 4.3 a full list of all types of search possible. All these search types are implemented in `KadSearch` class, that stands for both *Search* and *SearchManager* classes in eMule. As it is shown in listing 4.10, a search is actually a lookup where *tolerance contacts* (contacts in tolerance zone distance from the target) are sent a request, called *action*, and should reply with the desired resources, as being in *tolerance zone* they are probably responsible for those resources.

Listing 4.10: Portion of start() method in KadSearch.java

```
Lookup lUp = new Lookup(this.targetID, this.type);
LinkedList<KadContact> toleranceContacts = lUp.start();
        if (toleranceContacts != null) {
        for ( KadContact c : toleranceContacts ) {
                PacketUDP pck;
                switch (this.type) {
                case FILE:
                pck = new PacketUDPKad2SearchSourceRequest(this.targetID, ( (
                    KadFileSearch)this ).fileSize); break;
                case KEYWORD:
                pck = new PacketUDPKad2SearchKeyRequest(this.targetID, ( (
                    KadKeywordSearch)this ).tags); break;
                case NOTES:
                pck = new PacketUDPKad2SearchNotesRequest(this.targetID, ( (
                    KadNotesSearch)this ).fileSize); break;
                case STOREKEYWORD:
```

Table 4.3: Kinds of search

Type	Description
File	Search for sources that can serve the file.
Keyword	Finds files shared on the network related to the string searched for, targets the longest word in the string.
FindBuddy	Search for a buddy, using our own client ID as target.
FindSource	Finds the buddy in the network.
Node	Search for nodes, performed targeting a random ID.
NodeComplete	Search for nodes, performed targeting our own client ID.
NodeSpecial	Search for an exact node, given its ID, if we don't have its contact information.
Notes	Search for comments (notes) about a specific file shared on the network.
NodeFwCheckUDP	Targets a random ID to search a node to perform UDP firewall check.
StoreFile	Used to store a file reference in the network, that is to publish a file.
StoreKeyword	Used to store a keyword referencing a file shared on the network.
StoreNotes	Used to store a comment referencing a file shared on the network.

```
pck = new PacketUDPKad2PublishKeyRequest(this.targetID, ( (
    KadStoreKeywordSearch)this ).filesInfo); break;
case STOREFILE:
pck = new PacketUDPKad2PublishSourceRequest(this.targetID,
    Config.kadID, ( (KadStoreFileSearch)this ).tags); break;
case STORENOTES:
pck = new PacketUDPKad2PublishNotesRequest(this.targetID,
    Config.kadID, ( (KadStoreNotesSearch)this ).tags); break;
default:
```

```
            pck = null;
            }
        UDPListener.send(c.ip, c.UDPPort, pck);
        }
}
```

As for lookup, responses are not dealt in *start()* method that just sends requests, but they are managed through *onReceive()* methods in the various classes corresponding to different packets. Each kind of search in this case has its own packet for requests (if you recall lookup packets were all the same), while responses are sent as `PacketUDPKad2SearchResponse` for all search types[3], so it is necessary to discriminate first the search type upon reception of a response packet, then do the normal checks (as for lookup) and add the results to the corresponding search through specific methods (*addFileResults*, *addKeywordResults*, etc.), as seen in listing 4.11.

Listing 4.11: addResults() method in KadSearch.java

```
static int addResults(Int128 target, LinkedList<SearchResult> resultsList) {
        for ( KadSearch search : kadSearches ) {
        if (search.running) {
        if (target.equals(search.targetID)) {
                search.responses++; // Update responses count
                switch (search.type) {
                case KEYWORD:
                        return KadSearch.addKeywordResults((KadKeywordSearch
                            )search, resultsList);
                case FILE:
                        return KadSearch.addFileResults(target, resultsList);
                case NOTES:
                        return KadSearch.addNotesResults(target, resultsList);
                default:
```

[3]In reality this is true only for Kad2 packets, Kad1 packets used in older clients use different packets for all kinds of search.

```
                    PPLog.printError("Unknown or wrong Kad search type.");
                }
            }
        }
        PPLog.printWarning("Search for target " + target.toString() + " not found
            ", kadSearchesTargets());
        return 0;
}
```

Even if the response packet is the same for all the searches, they contain different information written as *tags* that are handled by `Tags.java`, and as we know already which tags are contained in a packet, it is easy to parse the results. Search terminates when at least one response is found or time limit is exceeded, but later results can be added even if no more search requests are sent. This way is possible to present the results to the user (useful in case of *keyword search*) while the response packets arrive, thus providing a better user experience.

As we said, even publishing is a kind of search: a lookup procedure is performed and final action is sent to tolerance contacts, but it terminates on different conditions. A publishing node that wants to store a keyword, a file or a note on a contact in the tolerance zone sends the corresponding *publsh packet request* and waits for `PacketUDPKad2PublishResponse` that contains the node load (as described in the previous chapter). Response count is incremented if node load is less that 100, that is to say that the resource was successfully stored on the node. Once at least 10 responses are obtained, the search terminates. Publish response packets are processed using *onReceive()* method in the response packet class and then *processPublishResponse()* does the usual checks.

4.5 Publishing

Peers store our resources, and in the same way our client is expected to accept incoming publish requests. This situation is handled by `KadIndexed` class that stores keywords, sources for files and notes. In the case of keywords, *onReceive()* method calls *addIncomingKeyword()* when receiving `PacketUDPKad2PublishKeyRequest`, other cases work the same way.
Constraints are defined on new keyword, sources and notes that can be stored on our node, in order to provide *overload protection* as for eMule. In listing 4.12 it is shown how the load, included in reply to contacts requesting to publish some content, is calculated and how keywords are stored in the *keywords* data structure.

Listing 4.12: Portion of addIncomingKeyword() method in KadIndexed.java

```
if (key == null) { //...if not, we create a new keyword entry
        key = new KeyEntry(keyword);
        keywords.add(key);
}
if (totalIndexedKeywords <= MAX_INDEXED_KEYWORDS) { //total indexed
    keywords <= 60000
        if (key.getSourceNum() != 0) { //Keyword has already sources
                if (key.getSourceRNum() > INDIVIDUAL_KEYWORD_LIMIT)
                        return 100; //Keyword has already more than 50000
                            sources
                //else
                if (key.getSourceNum() <= POPULAR_KEYWORD_LIMIT) {
                        key.addSource();
                        return ( ( key.getSourceNum() * 100 ) /
                            INDIVIDUAL_KEYWORD_LIMIT );
                }
                //else
                return 100;
        }
```

```
        //else
        key.addSource();
        return 1;
}
//else
return 100;
```

The current implementation of `KadIndexed` lacks many features and should be tested thorougly. It presents efficiency issues and also it doesn't work correctly. For instance, take a look at listing 4.13, where *keywords* is a `LinkedList<KeyEntry>`, it is obviously a poor choice and an `HashSet` would have suited the purpose better. For this reason the source code is still not merged in PariMulo SVN trunk code[4].

Listing 4.13: Efficiency issues in addIncomingKeyword() in KadIndexed.java

```
for ( KeyEntry k : keywords ) { //find out if the keyword already exists in our list...
        if (k.KEYWORD_ID.equals(keyword)) {
                key = k;
                break;
        }
}
```

4.6 Firewall check and FindBuddy

Firewall check and FindBuddy features are still experimental and yet to be tested for merge in PariMulo SVN trunk code.

[4]SVN (Subversion) is a repository where we store the code, mulo-trunk contains the most recent code with all stable features integrated.

FirewallCheck

As of this writing, only TCP firewall check is implemented. FirewallCheck is triggered when receiving `PacketUDPKad2HelloRequest`, `PacketUDPKad2-HelloResponse`, `PacketUDPKadRequest` and `PacketUDPKadResponse` packets through the *onReceive()* method.

Listing 4.14: onReceive() method that starts a firewall check

```
// Check if firewalled
if(KadPrefs.getRecheckIP()) {
        PacketUDP pck_FW2 = new PacketUDPKadFirewalled2Request();
        UDPListener.send(this.senderIp, this.senderUDPPort, pck_FW2);
}
```

In listing 4.14 is shown that *firewall check requests* are sent only if the client received less than 4 (*KAD_FIREWALL_CHECKS*) `PacketUDPKad-FirewalledResponse` packets from clients it sent a firewall request to. The packet contains an IP address that let us check if it corresponds known IP address.
A peer receiving a firewall check request will try to connect to us through TCP. If it succeeds on older client versions it will send a `PacketUDPKad-FirewalledAckResponse` packet, while more recent clients will send `Packet-TCPKadFirewallCheckAck` through the successfully estabilished TCP connection, both methods will call `KadPrefs.incFirewalled()` that when it's called twice sets our status as non-firewalled.

FindBuddy

FindBuddy mechanism, which was described in section 3.2.4, is implemented: firewalled peers can request our client to act as buddy and if we are behind firewall we can ask some peer to act as buddy. In the case our client is TCP firewalled and can't receive incoming TCP connections it needs to find a buddy to forward callback requests. A `PacketUDPFindBuddyRequest` is

sent to peers that can possibly act as buddy, and a `PacketUDPFindBuddy-Response` packet in expected. When a peer responds with such packet, a TCP connection can be established and the peer is set as buddy. Periodically `PacketTCPBuddyPing` are sent and answered by `PacketTCPBuddyPong` over the TCP connection in order to maintain it active. Whenever a third node needs to connect to our client, it sends a `PacketUDPKadCallbackRequest` packet to our buddy, that will forward us the request so that we can call back the requesting peer.

Chapter 5

Considerations and future works

This last chapter wants to make some considerations about working in a group of developers, point out some interesting aspects that should be considered in such context and give some advice to possible new *PariMulo* developers. Various tools to work efficiently in a team projects as *PariPari* are presented. Lastly, we give some hints about possible uses of PariMulo.

5.1 Working on PariMulo

Writing a file-sharing peer-to-peer client compatible with eMule[1] is a challenging task: translating *C++* to *Java* isn't as easy as it sounds and it is ultimately the best choice. Java provides powerful features and libraries that the developer should take advantage of, and of course *object-oriented programming paradigm* should be exploited, with concepts like objects and inheritance.

Working on *PariMulo* for *PariPari* project means also cooperating in a team of up to eight people that works with other teams forming a group larger than fifty people. This people works on the project for time-span that ranges from a few months to two or three years (some people spent more time in *PariPari*). This means that as the project becomes more complicated, both

[1] In the sense that is should behave like eMule as much as possible.

PariPari as a whole and PariMulo as a single plug-in, the time required for newly coming developers to be productive increases. To make an example, while when started working on *PariMulo* we knew all of the inner workings of the plug-in and know exactly where to put hands whenever needed, right now the details of some classes are obscure and need in-depth studying.

Work methodology

As it was said, documentation on the eDonkey / Kad protocols is incomplete and often obsolete, therefore is preferable to study directly the *eMule* source code instead of trying to reverse engineer the protocol analyzing network traffic. Anyway, if analyzing the traffic was the only option, there are some tricks that make it easier.

A great tool to analyze network traffic is *Wireshark*, available on all major platforms, that listens to a network interface and gathers data at different levels, including OSI network and transport layers which interest us mostly. If the eMule client is run on default ports (TCP: 4662 and UDP: 4672), it also detects most common eDonkey/eMule packets as eDonkey packets. Apart from detecting all the incoming and outgoing traffic on TCP and UDP ports, that let us check for erroneously discarded or incorrectly decoded packets, it also has a nice feature of following a *TCP stream* by selecting a single packet in a TCP connection, right-click and choosing "Follow TCP Stream" option.

Although many packets are correctly decoded by Wireshark, some may appear as large arrays of bytes that make no sense. An easy way to get a hint of their content is to guess what is the purpose of the packets and then try to get eMule to send (or receive) them with the desired content. To make things more clear we see an example:

> **Ex.** We send an ed2k *PacketSearchRequest packet* by making a search of the word "prova". The packet sent should contain such word as a sequence of bytes. As expect, the outgoing packet cap-

tured by Wireshark: `0xe3 0x14 0x0 0x0 0x0 0x16 0x0 0x0 0x1 0x5 0x0 0x70 0x72 0x6f 0x76 0x61 0x2 0x3 0x0 0x44 0x6f 0x63 0x1 0x0 0x3` contains “prova” word, in the sequence `0x70 0x72 0x6f 0x76 0x61`. By making a lot of tries, capturing as many forged packets as possible, one can understand accurately how a packet is built.

In the case of eMule a great help comes from eMule code. Even if the code is quite messy, it is possible to compile and run eMule in *debug mode*, with lot of log messages that help us understand what the program is doing.
eMule prints out all packets exchanged both via UDP and TCP, and it is also possible to modify some methods in order to print out the content of a packet. This is a great way to *test* our classes that encode and decode packets, as it is possible to forge *ad-hoc* packets or know exactly the content of a packet received from eMule client and confront it with information decoded from PariMulo. The test in listing 5.1 shows a packet received and printed by eMule (it was modified to print both the packet in a Java byte array format and the content as you can read in the comments). It is checked whether the infomration collected from eMule corresponds to data extracted from the packet using *assertEquals()* method.

Listing 5.1: Decoding test in PackingDecodingTest.java

```
@Test
// 12:29:41 KADEMLIA2_HELLO_RES from 77.194.173.139:64309
// 12:29:41 ClientId: 67E2610143DDE28E97208F8761DA8E87 TCP: 5820 Version: 8
public void Kad2HelloResponseTest3() throws UnknownHostException {
        byte[] packet = {
        (byte) 0xE4, (byte) 0x19, (byte) 0x01, (byte) 0x61, (byte) 0xE2,
        (byte) 0x67, (byte) 0x8E, (byte) 0xE2, (byte) 0xDD, (byte) 0x43,
        (byte) 0x87, (byte) 0x8F, (byte) 0x20, (byte) 0x97, (byte) 0x87,
        (byte) 0x8E, (byte) 0xDA, (byte) 0x61, (byte) 0xBC, (byte) 0x16,
        (byte) 0x08, (byte) 0x01, (byte) 0x08, (byte) 0x01, (byte) 0x00,
        (byte) 0xFC, (byte) 0x35, (byte) 0xFB};
```

```
        PacketUDP pck = PacketUDP.decodePacket(packet, InetAddress.
            getByName("85.85.47.85"), 44125, false);
        if (pck.getClass() == PacketUDPKad2HelloResponse.class) {
                PacketUDPKad2HelloResponse pack = (
                    PacketUDPKad2HelloResponse) pck;
                System.out.print("Id: "+pack.clientID.toString());
                assertEquals(pack.clientID, new Int128("67
                    E2610143DDE28E97208F8761DA8E87"));
                System.out.print("TCP: "+pack.TCPPort);
                System.out.print("Version:0x"+Utils.byteToHex(pack.kadVersion));
        }
}
```

It is easy to collect a lot of packets in this way and test them easily, it is even possible to write a tool that automatically saves the packets and tests them in PariMulo.

5.2 Testing

Writing complex programs written by many developers results in a lot of bugs, and the most recurring errors are due to poor thinking of how code will be used by others. Think about method *doubleInt* in listing 5.2. It works perfectly when called with input 1 or 2, but it fails in other cases. Now, when we write both the method and the callers we can be sure that none will call such method in an unexpected way, but it would be better to make the method error-proof just in case.

Listing 5.2: Bad doubleInt() method

```
public int doubleInt(int a) {
        int toReturn = 0;
        switch(a) {
                case 1: toReturn = 2; break;
                case 2: toReturn = 4; break;
```

```
                default: break;
        }
        return toReturn;
}
```

In the previous example the method could have been written better easily, but we should at least report an error when the program doesn't behave correctly and we know already, in this case we could have modified default case with `throw new IllegalArgumentException()` statement so that the caller would know that something went wrong.
In PariPari, *Extreme Programming* software development methodology is pushed, among the practices it advocates there is *Test-driven Development*: this process demands short development cycles where first tests are written and then code complying with the tests is developed. Even if this devel-

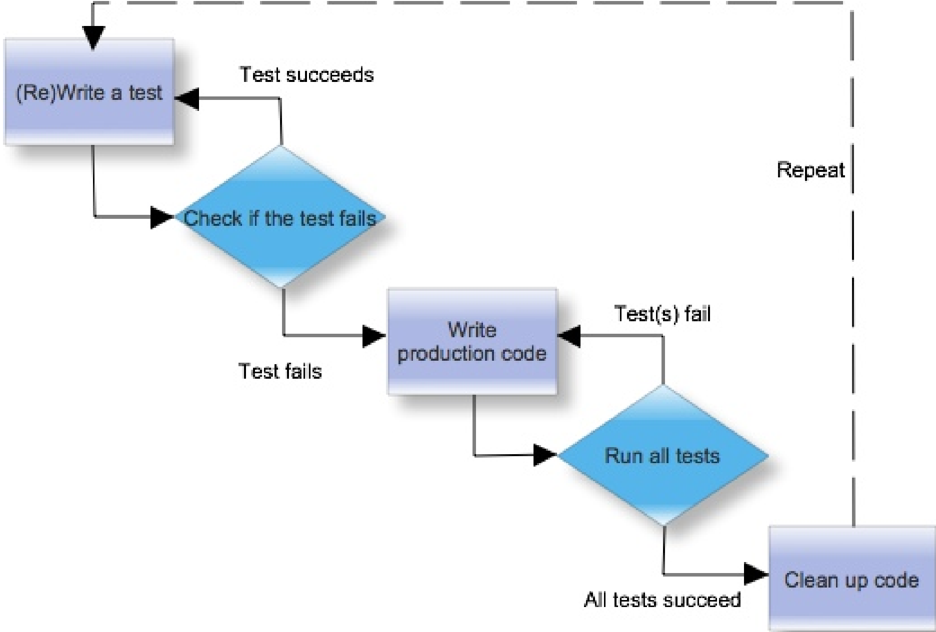


Figure 5.1: Test-driven Development

opment methodology proved to be quite difficult to achieve, probably due

to the fact that PariMulo is just a complete rewrite of a complex software written in another language, testing code is a good practice that should be seen as a useful tool for developers.
JUnit is a powerful test framework for *unit testing*: that is to say testing small parts of source code, verifying that it works correctly and individuating bugs early. From a Java perspective writing a unit test means writing a method with some special statements that verifies the functionality of a part of code. As an example, listing 5.3 shows a JUnit test that checks, throught *assertEquals()* [2] statement, the correctness of *doubleInt()*.

Listing 5.3: Sample test method for doubleInt()

```
@Test
public void provaTest() {
        assertEquals(doubleInt(1),2);
}
```

In this case, even if the test succeeds, the test proves that the method works correctly for input *1*, but it does not prove anything about other parameter's values. In fact, if we write a second improved version of the test, as shown in 5.4, an `IllegalArgumentException` is thrown, and test fails.

Listing 5.4: Improved test method for doubleInt()

```
public void provaTest() {
        for(int i = Integer.MIN_VALUE; i < Integer.MAX_VALUE; i++)
                assertEquals(doubleInt(i),2*i);
}
```

Testing has also been defined as a challenge between the tester and the programmer, where the tester should find ways to let the unit tested to fail. In reality it's up to the sensibility of the programmer to find a trade-off between neat and robust source code.

[2]assertEquals(expected, actual) takes as parameters expected value and actual value, if they do coincide, test succeeds.

Testing packets

In the case of PariMulo, where compatibility must be assured with the network, it is important to check whether the packets are encoded and decoded correctly from and to byte arrays that are sent through the network: even a wrong byte could result in a disconnection from the server we are connected to or a ban from a peer we are exchanging files with. With the help of *eMule* and traffic analysis depicted in section 5.1, it is possible to collect packets to be tested against, and verify if PariMulo correctly encodes a packet with the same content or decodes the packet collected by eMule in the same way.

Listing 5.5: A test method in PacketEncodingTest.java

```
@Test
public void requestKeyTest2() {
byte[] originalPck = {(byte) 0xE4, (byte) 0x33, (byte) 0x52, (byte) 0x6B, (byte) 0
    x30, (byte) 0x39, (byte) 0xD4, (byte) 0x44, (byte) 0xD7, (byte) 0x32, (byte)
    0x04, (byte) 0x9B, (byte) 0x9F, (byte) 0x34, (byte) 0x7E, (byte) 0xCC, (byte
    ) 0xA8, (byte) 0x01, (byte) 0x00, (byte) 0x00};
Search src = new Search(Type.KAD, "enya");
KadSearch ksrc = new KadKeywordSearch(src);
PacketUDPKad2SearchKeyRequest builtPck = new
    PacketUDPKad2SearchKeyRequest(ksrc.targetID);
System.out.println("eMule original:\t"+Utils.bytesToHexString(originalPck));
System.out.println("PariMulo built:\t"+Utils.bytesToHexString(builtPck.toBytes().
    array()));
assertArrayEquals(originalPck, builtPck.toBytes().array());
}
```

`PacketEncodingTest` class contains a number of methods that test if packets, created as objects and then put in `ByteBuffer` structures to be sent, coincide with packets retrieved with eMule that contain the same information. In listing 5.5 it is shown a test for `PacketUDPKad2SearchKeyRequest` packet,

and test succeeds only if the array of bytes are identical. Of course one test is not enough for this kind of packets, as it can contain many more options and informations. It doesn't either verify decoding correctness, that is handled by `PackingDecodingTest` class.

Listing 5.6: A test method in PackingDecodingTest.java

```
@Test
public void KadFirewalledRequestTest() throws UnknownHostException {
byte[] packet = {(byte) 0xE4, (byte) 0x50, (byte) 0x8F, (byte) 0x1B};
PacketUDP pck = PacketUDP.decodePacket(packet, InetAddress.getByName("
    120.32.40.9"), 23528, false);
if (pck.getClass() == PacketUDPKadFirewalledRequest.class) {
        PacketUDPKadFirewalledRequest pack = (PacketUDPKadFirewalledRequest
            ) pck;
        System.out.println("Port: "+pack.TCPPort);
        System.out.println("Size: "+pack.size);
        assertEquals(pack.TCPPort, 7055);
}
```

In listing 5.6 the test method passes an array of bytes repulsing the packet to `PacketUDP.decodePacket()` decoding method, which is usually called by `UDPListener`, with a fake sender IP address and port (as it does not matter in this test). The decoded object has its instance variables tested against information we know the packet contains (in this case a `TCPPort short`), the test succeeds only if the field is equal to the expected value.

5.3 Future works

In this section are presented a few unconventional ways of using *PariMulo* code, that is to say running its code without all the PariPari infrastructure. PariMulo relies on PariPari classes[3] for storage, connectivity, logging, threads

[3]Those classes act as wrappers of normal `java.net`, `java.lang` and `java.io` classes.

and other stuff, but if PariPari system is missing it falls back on Java classes provided by `java.net`, `java.lang` and `java.io`. It is therefore possible to use PariMulo *stand-alone* and run it even on devices that cannot run PariPari. As no *Graphical User Interface* was integrated (differently from eMule where GUI is tightly integrated with the rest of the code) it makes PariMulo perfect for use as a framework or platforms with different user interfaces.

5.3.1 PariMulo as a framework

Thanks to the architecture of PariMulo, with independent objects handling different aspects of the client, it is possible to use only parts of the program. PariMulo therefore can be used as a framework for various tasks, from analysis of traffic to network-wide distributed attacks.
Kad / eDonkey networks support are independent, therefore it is possible to discard eDonkey support classes, like `Server`, if not needed. To demonstrate modular architecture, by starting just an `UDPListener` instance (as it is done for unit tests on packet encoding/decoding), packets described in `PacketsUDP` can be received or sent, calling actions upon packets arrival. Packets are objects, so they can be directly created and forged as needed, and by inserting some specific values it's possible to behave like many different clients connected to the Kad network. To prepare attacks to the Kad network that require our client to pretend to be many nodes at once, only one instance of `UDPListener` needs to be running, but some minor modifications should be done for handling responses for requests sent by the client acting as many different nodes.
Launching many instances of `UDPListener` or Routing table makes little sense as, for Kad network, they can be shared by many nodes on the same machine and therefore running many nodes does not take a lot of system resources.

5.3.2 Kad for embedded systems

Nowadays a number of embedded system provide *Software Development Kits (SDK)* to develop applications for these platforms. Most notably, *Android platform* runs software compiled to execute on *Dalvik Virtual Machine*, with programs written in *Java language*.
Android operating system runs on many smartphones and tablet computers, with a variety of hardware specifications: processor frequencies are in the *500 MHz - 1.4 GHz* range, RAM memory from less than 256 MB to over 1 GB, and devices sport all kinds of storage capacities. The point of writing a Kad client for such systems is that these devices are powerful enough to run the client and most of the time they are not just connected to the Internet through *3G networks*, but also through more reliable *802.11 networks*. Even if running a peer-to-peer file-sharing client may drain the battery quickly, running such clients can be effective to download small files.

Limiting tasks and traffic

With this goal in mind we want to modify PariMulo to run on a low-resources devices as a leecher client for Kad network. A client is called a "leecher" when it downloads contents from the network but does not return the favor to other peers, exploiting poor credit system. We therefore start by choosing which classes are needed and which can be discarded.
There is no need to store and maintain a routing table, it requires maintenance tasks to be run regularly and it is mainly used by peers to keep the network topology working, for this reason even if some peers behave badly (having no routing table means that the peer can't answer to node requests), the network still works fine. With no routing table our client just relies on other peers' routing tables, and that means that lookup requires $O(log\ n)+1$, thus still $O(log\ n)$ steps. When the average number of hops to get to tolerance zone is around 3, it has no impact on client performance.
Our leecher client does not need to respond to other peers' requests: such as

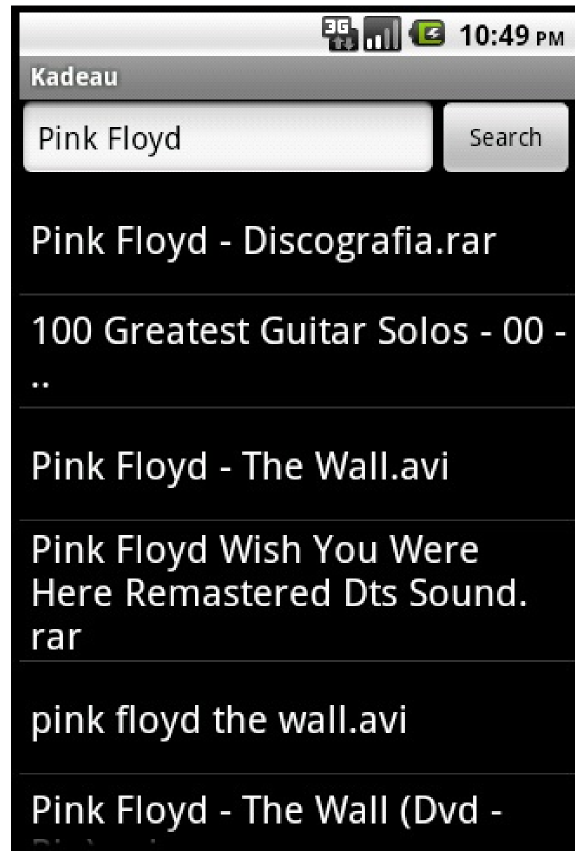


Figure 5.2: Screenshot of a first experimental version of Kad client running on Android 2.2 platform, while performing a search request.

route requests, publish requests, search requests; eventually some peers will ban our client, but our tests have shown that even with our client misbehaving, our requests are fulfilled flawlessly. This makes our client less involved in the network, economizing on network traffic.
All the classes that manage uploads can be discarded, while incoming TCP connections should be handled as they are useful for callback connections from firewalled clients that will serve us files.

Limiting threads

While computer nowadays run on *multi-core systems* that take advantage of running many threads concurrently, embedded systems are mostly *single-core systems*, and due to some operating system's constraints, the number of threads running should be dealt carefully. It is therefore important to limit the number of threads that our client launches when performing various tasks: one thread is launched for each lookup and search, while downloads start many threads, as each one handles a single serving peer. As concurrent lookups are rare and generally they are run once in a while (especially if no periodic maintenance task is run), it makes no sense to overcomplicate the `Lookup` and `KadSearch` classes to perform all lookups in one thread. Instead, speaking of downloads, it is common to have several hundreds of serving peers at once, thus it's mandatory to limit the number of running threads.
One solution could be using a single thread managing all downloads through `DownloadManager` class and a limited number of threads handling all incoming packets and triggering immediate responses upon packet arrival. Java provides in `java.util.concurrent` *thread pools*, which consist of *worker threads* (they commonly handle background tasks such as incoming requests), that minimize overhead due to thread creation. `java.util.concurrent-.Executors` offers different options when dealing with thread pools: it is possible to choose a different number of threads depending on the system that runs the application. While on an embedded system we could limit the number of threads to only one, on a more powerful machine this system is still more convenient than unlimited threads creation: if we could create an unlimited number of threads, threads would grow in number until the system becomes unresponsive to all requests when the overhead of the threads exceeds capacity of the system. On the other hand, a *fixed thread pools* let the system degrade gracefully: requests exceeding the number that the system can sustain are not handled until a threads becomes available.

5.3.3 Developing a new network

While implementing Kad for PariMulo the aim was full compatibility with eMule clients connecting to Kad network, our implementation is a good starting point to develop a new peer-to-peer network based on Kad. The key aspects one should take care of modifying PariMulo to build a network are bootstrap nodes, resources shared and network limits imposed by Kad implementation.

Bootstrapping

Choosing how to retrieve nodes for bootstrapping is fundamental: if nodes retrieved are stale or insufficient to populate the routing table and perform successful lookups our client can't join the network and access any resource. Bootstrapping nodes could be chosen by a trusted peer and published on a nodes.dat file on the web for retrieval, and updated regularly to prevent staleness of the nodes. Better yet, to prevent DoS of such nodes, a larger list could be published to the web by many trusted peers that joined the network, and a dinamically generated list may be presented upon request of clients that want the bootstrap nodes.dat file. Thus, only minor modifications are needed on `Bootstrap.java` file, hardcoding the right locations of new nodes.dat files. Hypotetically speaking, a parallel Kad network could be created by setting up a group of nodes and putting them in a nodes.dat. As these nodes have no knowledge of existing Kad peers in the original network, a new network would be created with new peers joining if they bootstrap from new nodes.dat file: this explains how important is choosing the right bootstrap nodes.dat when accessing the Kad network.

Resources

Kad network is a file sharing peer-to-peer network where files aren't stored directly on the network, just indexing of the resources is stored. Nothing prevents from storing resources or information on peers, but referencing in

this case works best for backward-compatibility with eDonkey network. Resource expiration and indexing limits on nodes can be modified, allowing less republishing overhead.
Resources should be mapped with hash functions on a 128-bit kad id (represented by `Int128`) to take advantage of consistent hashing. Of course all communications related to resources need to be modified, in particular `KadSearch` and `PacketsUDP` classes need heavy modification.

Implementation limits

Kad network was designed to support a number of peers similar to eDonkey network, that is to say a few millions users connected at once. Routing table and lookup processes are designed to perform well in such conditions, but in the case of a new network with different purposes other than file-sharing and relevantly different number of users, some parameters defined as constants should be reviewed. For instance, while 128-bit key space for resources and peers works well because there is a low probability of collision and collision would not be critical for the network, *tolerance zone distance* as defined in Kad, allows only 256 (2^8) tolerance zones, which presents scalability issues. In the case of a too small network, some zones may be empty with no nodes responsible for resources. In a network expanding with several millions of peers it may be difficult to locate nodes responsible for a resource even in the tolerance zone, it depends on resource popularity and replication of publishing (currently a content is published on 10 nodes in the tolerance zone).
The best choice for a scalable network would be to define variable parameters instead of constants, but this clearly increments the complexity of the software, and in some cases it mandates for peers agreements that are difficult to achieve in a distributed context. Some constants, as α parameter for lookup concurrency or limit of references stored in a node, act only on local tasks and therefore can be changed during network development as they have no effect on the whole network when a few nodes are affected. It should be considered however, that even changing α parameter can affect the network

in terms of traffic overhead and therefore while designing a new network all those parameters should be considered carefully.

Bibliography

[1] Petar Maymounkov and David Mazìeres. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric.*

[2] Mattia Francesco. *Reverse engineering e analisi del protocollo eDonkey2000.*

[3] René Brunner. *A performance evaluation of the Kad-protocol*

[4] Yoram Kulbak and Danny Bickson. *The eMule Protocol Specification* `http://www.cs.huji.ac.il/labs/danss/p2p/resources/emule.pdf`

[5] Ampezzan Roberto. *PariMulo 2009.*

[6] Piccolo Christian. *PariKad.*

[7] Marzo Alessio. *PariPari: Callback eDonkey.*

[8] M. Steiner, W. Effelsberg, T. En-Najjary, E. Biersack *Load Reduction in the KAD Peer-to-Peer System.*

[9] M. Steiner, E.W. Biersack, T. En-Najjary *Exploiting KAD: Possible Uses and Misuses.*

[10] D. Carra, E.W. Biersack *Building a Reliable P2P System Out of Unreliable P2P Clients: The Case of KAD.*

[11] M. Steiner, T. En-Najjary, E.W. Biersack *A Global View of KAD.*

[12] M. Steiner, D. Carra, E.W. Biersack *Faster Content Access in KAD.*

Listings

3.1 eMule CanSplit() method in RoutingZone 32
4.1 Portion of parseDatagram() in UDPListener 49
4.2 Portion of decodePacket() from PacketUDP in Packets.java . . 50
4.3 Class PacketUDPKadBootstrapRequest in PacketsUDP.java . 52
4.4 Int128 constructor 55
4.5 split() method in RoutingZone.java 58
4.6 onSmallTimer() method in RoutingZone.java 59
4.7 onBigTimer() method in RoutingZone.java 60
4.8 addContacts() method in Lookup.java 62
4.9 distance() method in Utils.java 63
4.10 Portion of start() method in KadSearch.java 64
4.11 addResults() method in KadSearch.java 66
4.12 Portion of addIncomingKeyword() method in KadIndexed.java 68
4.13 Efficiency issues in addIncomingKeyword() in KadIndexed.java 69
4.14 onReceive() method that starts a firewall check 70
5.1 Decoding test in PackingDecodingTest.java 74
5.2 Bad doubleInt() method 75
5.3 Sample test method for doubleInt() 77
5.4 Improved test method for doubleInt() 77
5.5 A test method in PacketEncodingTest.java 78
5.6 A test method in PackingDecodingTest.java 79