Darmstadt University of Technology

The eDonkey 2000 Protocol

Oliver Heckmann, Axel Bock

KOM Technical Report 08/2002
Version 0.8
December 2002

Last major update 22.05.03
Last minor update 27.06.03

E-Mail: {heckmann, bock}@kom.tu-darmstadt.de

Multimedia Communications (KOM)

Department of Electrical Engineering & Information Technology
& Department of Computer Science
Merckstraße 25 • D-64283 Darmstadt • Germany

Phone: +49 6151 166150
Fax: +49 6151 166152
Email: info@KOM.tu-darmstadt.de
URL: http://www.kom.e-technik.tu-darmstadt.de/

1. Introduction

The Edonkey2000 Protocol is one of the most successful file sharing protocols and used by the original Edonkey2000 client and the open source clients mldonkey and EMule. The Edonkey2000 Protocol can be classified as decentral file sharing protocol with distributed servers. Contrary to the original Gnutella Protocol it is not completely decentral as it uses servers; contrary to the original Napster protocol it does not use a single server (farm) which is a single point of failure, instead it uses servers that are run by power users and offers mechanisms for inter-server communication. Unlinke Peer-to-Peer (P2P) protocols like KaZaa, Morpheus, or Gnutella the eDonkey network has a client/server based structure. The servers are slightly similar to the KaZaa supernodes, but they do not share any files, only manage the information distribution and work as several central dictionaries which hold the information about the shared files and their respective client locations.
In the Edonkey network the clients are the nodes sharing data. Their files are indexed by the servers. If they want to have a piece of data (a file), they have to connect using TCP to a server or send a short search request via UDP to one or more servers to get the necessary information about other clients sharing that file.
This documents presents our findings about the eDonkey protocol based on TCPDump, Ethereal and TCPFlow analysis.
The document in its current version it is work in progress!! We have used the information contained in this document for P2P traffic measurements. The measurement results will be published in a separate document and in the student thesis of Axel Bock. For a high-level description of the protocol and a comparison with other P2P protocols see our report TR-KOM-2002-06 at http://www.kom.e-technik.tu-darmstadt.de/publications/abstracts/HSS02-3.html (which is written in german).

After this introduction we present a general overview of the protocol in section 2, present the general message format in section 3 and conclude with a detailed description of the messages in section 4.

2. Protocol Description

Typically a client connects to one server:

- A client connects a server via TCP and stays connected
- The client sends information about itself, containing username, IP address and connection port
- The client sends a list of the files it offers to the server. This information is added to the server's files database.
- A list of other known servers is transferred from the server to the client.

After this the client is "connected" it can use the eDonkey network to search and download files, while itself shares his files by making them available for download by other users. Between the clients themselves there is no communication except the communication needed to initiate and run file transfers.

The servers do loosely communicate over UDP to keep their internal server list up-to-date. This is done by ping/pong mechanisms and server list exchanges.

Now follows a descriptions of several network communication mechanisms. The numbers in brackets are eDonkey message markers. To get detailed information about these see section 3.

2.1. Server - Server Communication

The servers communicate with other servers only via UDP messages. Upon connection to the network a server begins to send UDP "Announce" messages (0xa0) to other servers to make itself known.

The communication between servers is simple and straightforward: They can can exchange serverlists (request 0xa4 and reply 0xa1), ping each other (0x96) and announce the very own presence (0xa0). The pongs (0x97, the answers to ping queries) contain also the current users/files count. There is another UDP packet which delivers a description of the server, but this is of minor importance for the network.

Unfortunately do the clients also send ping messages to verify the correctness of their respecive internal server list, which is the reason the ping/pong messages make a very high percentage of the overall traffic volume.

2.2. Client - Server Communication

Login. The following login procedure happens when a client connects to one eDonkey server. The original client and the Emule client try to stay connected with a single server while the mldonkey client can connect to more than one server at a time.

- The client opens a TCP connection to the server it wants to connect to.
- Then the client sends a "hello" packet to the server (0x01), which contains his username, his user hash, the IP and port on which he can be reached and his protocol version.
- Now the server checks whether the client is a so-called "low-id" client by checking whether the client is firewalled or not (wether a TCP connection can be opened to his eDonkey port or not).
  A firewalled client is called "low-id" because the client's id is normally very high (in the millions, cause this is just the integer representation of the client's IP), and when it is firewalled, the server gives it an IP starting from 1.
- After this the server answers with a 5 bytes message (0x40) containing the clients ID (the marker & the ID).
- Then the server send the server greetings by sending a bunch of "write string" commands (0x38),

which are printed out by the client for the user to read.
- As the last step of the connecting procedure the server usually sends some additional information about itself: A message (0x41) containing the official server name and description (this is *not* the server greeting) and/or the number of users logged in hosting just so many files (0x34).

User interactions. Next follows a brief description of what happens when a user does certain things within the network, namely file searches and file downloads.

Text search based on filename. The most used user interaction in the eDonekey network is a file search, for file downloads happen quite automatically. File searches are always text searches; the entered text is split up into single words and all known filenames are searched for occurances of all these words. If a file has all sent words in its name, it matches. The server can be configured to do complete substring searches or just compare whole words.
On the protocol level the realization is quite simple:
- The client sends a message (0x16) containing a search string and some search parameters. The parameters can be min and/or max size, the type of the file (audio, video, image, ...), some type specific parameters (bitrate, resolution, length, ...), or just some additional search terms.
  The search string is cut down to the maximum size of four words in all lowercase on the client side.
- The server answers with a message containing the descriptions of all matching files (0x33), whose consist exactly of a (filename, hash, size) tuple and - if present server-sided - some additional information about some special file types (e.g. audio length / bitrate, video compression algorithm, image author, program format).

Download initiation. Should the user decide to download a certain file, he tells his client to retrieve this file from the net. After this point the user can lean back and wait for the file to be completed.

The client/server communication which happens after this looks like this:
- Immediately after telling the client to download the file, it sends a “query sources” message (0x19) to the server, which only contains the hash value of the desired file (the MD4 hash of a file is used instead of filenames to identify files independantly from their filename).
- The server answers with a message containing the possible download sources (0x42), which is in fact just a list of ID/port pairs of clients claiming they hold this file. Due to transmission of the ID and not the IP the receiving client can see at once whether the other part is firewalled or not and can - if neccessary - request a server-sided push command to be sent to the other one.

Download. Now the client which requested the sources connects each of the named sources and asks for the file to be transmitted to itself. Here it is possible that the connected client has only parts of the file the first client already has, for the clients already share when they have not yet finished downloading. This increases the availability of often wanted files faster, but also leads to some annoying problems when receiving such an often requested file. But more on this later.

The client/client communication mechanisms are not yet covered by this document.

2.3. Network behaviour

On our test machine (Linux, Kernel 2.4.18+) we have observed some major network performance problems. One time we tried to shut down the official eDonkey server, which caused undefinitely **more** traffic than keeping the server running. More detailed forensics came up with the following indications:
- When the server was already down, there werr still some 150+ open TCP connections, all waiting to be closed. This may be due to some very crude network programming.
- The network itself doesn’t realize the server going down, especially not the clients. The remaining incoming UDP traffic after server shutdown adressed to the eDonkey port was about 50+K/s, which

strangely increased after the server shutdown (probably because the machine was no longer sending replys the UDP messages were sent a second and third time...).

- Finally the *outgoing* traffic increased massively after server shutdown. The incoming UDP packets on port 4665 caused the kernel to send "ICMP port unreachable" packets back to the senders, which blocked the upload line (ADSL) almost completely. A new dialin was the only thing that helped. The server simply ignores the UDP packet it cannot process any more, which explains the outgoing traffic increase.

Admittingly this does not always happen if a server is shut down, but at least the open TCP connections can be observed every time the server goes down.

3. Message Format

TCP Message format. The TCP part of the protocol (the TCP streams between client and server) is divided into logical „messages“ at the application layer. These are called *messages* from now on to mark the difference to packet based protocols.

Every sent TCP message is wrapped in the following header:

```
e3 xx xx xx xx
```

e3 is the edonkey marker and xx xx xx xx is the amount of bytes following this little header (unsigned int). The first byte after this header determines the blocks content (e.g. download request, text search, publish files, etc.), so every packet’s first six bytes are:

```
e3 ss ss ss ss mm
```

Where ss stands for a part of the message size, and mm is the message type marker.

The included packet content consists mostly of two parts.
First comes a fixed part of data which always has the same layout, so that there can be no problems parsing this part.
The second part consists of a variable number of so-called *“tags”*, which can hold almost any type of data, mostly depending on the type of the packet they are in. Separated are these two parts by a tag marker, which indicates how many of these tags follow.

So this will in general look like this:

```
e3 ss ss ss ss mm       <fixed part>      <tag count>             <following tags>
message header          as said ...       how many tags follow?   the following tags ...
```

Tag format. The understanding of the tag structure is essential for the understanding of the network protocol, cause it also explains why it is possible that different clients still work together although one of them may have an extended protocol set (eMule for example).

As said a tag is just a structural representation for a certain kind of data. There are far more tag types than covered in this document, but for the core eDonkey2000 protocol there are only two of them of interest: string and integer. Both can be very easy decoded when you know how to read them.

A tag is always of the following format:

```
<data type> <data description> <data>
```

Where data description is some kind of tag itself. Some examples:

```
Tag: client port:    03 01 00 0F 36 12 00 00
Tag: filename:       02 01 00 01 05 00 .h .e .l .l .o
Tag: file size:      03 01 00 02 3D 0F 00 00
Tag: bitrate:        03 07 00 .b .i .t .r .a .t .e 80 00 00 00
```

You can see that the tags start with different numbers - 02 and 03. 02 stands for text data type, whereas 03 stands for integer data type. There are still more defined but they are to no importance within the protocol.

After the leading type there comes the description. This has always the form

<size> <description>

The size is always two bytes unsigned short. As you can see, in the examples of client port, filename and file size the size of the description is one, and the description itself some kind of special marker used for very common tag types. In the bitrate example is is different as you can see: an integer data with the description “bitrate” and the value 128.

The data itself differs from string to integer - a string is just length (two bytes unsigned short) and then the string following, while integer data is always fixed size four bytes integer.
So you have to read this as follows:

02	01 00	01	HELLO WORLD
data type: string	size of description	special tag: name (user or file)	the data - a string in this case
03	07 00	“BITRATE”	80 00 00 00
data type: integer	description length	the description: the string “bitrate”	the data: int 128 for the bitrate

This is exactly how the protocol tags work.

Now you can also guess how proprietary extensions of the protocol are easily possible: the clients just ignore all tags they don’t immediately recognize and go over to the next one. This is only possbile cause the tag sizes are exactly known in advance! The only thing you cannot do is to inject new data types into the protocol. But you could simply create a new “special tag” (a tag which length is one byte) of the string type. In the string you pack your information, and all clients which do not know this tag simply go on to the next one.

Tag list. This is a list of all relevent protocol tags. The cDonkey server is built upon these. I have not included sample data, so this is ONLY the data type and data description part, divided by string and integer types.

(User-/File-)Name:	02 01 00 01
File type:	02 01 00 03
File format:	02 01 00 04
Server description:	02 01 00 0b
Filesize:	03 01 00 02
Client version:	03 01 00 0f
User port:	03 01 00 11
Availability (files):	03 01 00 15

Now follows a rather useable analysis of the eDonkey’s network messsages, divided into client / server communication (first TCP, then UDP), and then server / server communication (again first TCP, then UDP).

Search messages. The only exception to this format are search messages. They do also contain these advanced tags, in case you search for a certain file type or a certain file size, but due to the treelike nature of these (something like (type == PRO) AND (size > 100>) AND (size < 10.000) for example) they are transmitted in another format.

UDP packet format. The UDP packets have just the same format as the TCP messages, but with no size included. There are minor differences in some packets, but we’ll point these out later.

4. Detailed communication analysis

What follows now is a rather useful analysis of the eDonkey 2000 messages. This includes the message markers, the fixed part of the packets, and the mostly sent tags. For the tags we will always give type and description when needed. So for the bitrate we will say “... an integer with the ‘bitrate’”, and for the file name only “file name tag”.

The messages will be split up into TCP and UDP messages, for there are some little differences (besides the missing size field in UDP packets). Some messages are absolutely similar, these we will not list twice.

And we will include a short description of the message and some additional information about its contexts when appropriate. Also we filled in some values in fields which are normally variable, the tag count field for example. In this case this just means *“in most cases this field takes the value given here”*.

To simplify matters we didn’t include the edonkey marker 0xe3 at the head of each message in TCP and UDP, as well as the size information in the TCP messages.

All values are hexadecimal.

4.1. TCP communication: client to server

A description of all traffic the clients can send to the server.

4.1.1.TCP 0x01: hello

Sent by a client directly after connecting. After evaluating whether the client gets a low-id or not the official server sends the clients’ ID back.

Usually included tags: user name, client version, client port.

bytes	content	meaning
1	`01`	packet type
16		user hash
4		client ip
2		client port
4	`03 00 00 00`	tag count
the tag part ...		
4	`00 00`	trailing zeroes

4.1.2.TCP 0x14: request serverlist

This one byte packets asks the server to send a serverlist to the client.

bytes	content	meaning
1	`0x14`	packet type

4.1.3.TCP 0x15: offer files

This block has exactly the same format as the „search result“ block sent by the servers upon a search request. Please stick to this for detailed information.

The format of the UDP „extend search“ *response* packets is also the same, but without the number of results. Every UDP answer contains exactly one result, if more were found by the server, more packets are sent.

The UDP search result message marker is 0x99, see below.

4.1.4.block type 0x16: do text search

The search mechanism is the only exception of the so-called meta-tag packet model. Because a search query is nothing more than a combination of AND’ed or OR’ed search queries, it can be seen as a kind of binary tree, which is exactly what is transmitted. An example is given below:

((substrings “unreal cd1”) and ((file size > 500M) or (file size < 10k)))

This search could be used to find - for example - the CD image of the first CD of the game “Unreal” as well as the corresponding *.CUE-file. (The cue files are text files which contain the neccessary information to burn a given CD image - the image is large, the .cue-file very small).
As you can see you can write this search query as a binary tree, which is exactly the format of the eDonkey text search messages: a preorder binary tree. In practice though the search flexibility is limited to some extend, because the full flexibility definitely not needed within the eDonkey network.

In the treelike search query the nodes are the operators and have the form “00 <OP>”, so every node not beginning with a leading zero is a leaf and contains an expression. There are - but this is only a guess - three operators: AND (0x00), OR (0x01) and AND NOT (0x02), whereby AND and OR are verified and AND NOT was not seen by ourselves but recorded in various public available protocol analyses.
The leafes have three possible formats, for the three possible searches. The first search is a substring search within the filename, and has the prefix 0x01. The second one is a tag search, with which you can search for file attributes, and has the prefix 0x02. The third and last one is a minimum/maximum search and is used mostly for file sizes. It has the prefix 0x03.
The format of all three is a little bit different.

Substring:	01	<len string> Unsigned short	<string> byte array		
TAG:	02	<len string> ushort	<string> string to match	<len tag> ushort	<tag> tag to be searched
Minimax:	03	<int> uint	OPERATOR 1 byte, see below	<len tag> ushort	<tag> tag to be searched

The minimax search seems to differ slightly between the versions. What we title as “operator” is one

byte which determines the match behaviour - greater or equal vs. smaller or equal. What we found is that 01/03 (odd numbers) means “up-to” and 02/04 “not-more-than”. A detailed search query out of practice will be analyzed later.
As you notice with these three search methods you can search for every string in every combination of file attributes.

4.1.5.TCP 0x19: search download sources

Files in the eDonkey 2000 network are only found by their respective hash values. With this message the client asks the server to send possible download locations for a file with a certain hash value.

bytes	content	meaning
1	`0x19`	packet type
16		file hash

4.1.6.TCP 0x1c: push request

When a clients desires a file from another, firewalled client, it tells the server to initiate a push request. Then the server sends a push command over the existing TCP connection to the other client, which tells the other one to connect the first.
The connection information is supplied in the message.

bytes	content	meaning
1	`0x1c`	packet type
4		ID of client to push

4.1.7.TCP 0x21: more results

There seems to be a message called “get more results”, used when the server finds more than the possible 255 ones (the field containing the number of found results is exactly one byte). Clients getting a file list with 255 results would be able to request the missing ones with this.
But this message was never encountered, so maybe it’s gone in current versions of eDonkey clients and servers.

4.2. Server - Client Communication, TCP

A description of all traffic the server can send to the client.

4.2.1.TCP 0x32: serverlist

bytes	content	meaning
1	0x32	tag: server list block
1		number of following servers
	number of servers *	
4		IP of a server
2		according server port

4.2.2.TCP 0x33: return search results

This is a relatively large block where the knowledge of the tag format is essential for parsing the data. This is the only message type where the "fixed part / tag count / tag part" is broken up.

bytes	content	meaning
1	0x33	tag: return search results
4		number of results (unsigned int)

Now comes *for every returned file* an information block looking like this:

	...	
16		the edonkey MD4 file hash
4		a clients IP (?) or zero*
2		the clients port - or zero*
4		tag count: num. of tags follo-wing
	the tags come here ...	

Note that after all files there comes a last trailing zero. Thus the format is: header with number of results, file information block for every found file, and afterwards a trailing zero. In detail:

bytes	content	meaning
1	0x33	tag: return search results
4		number of results (unsigned int)

... for all files		
16		file hash
6		IP / Port pair or 0x0
4		tag count: # of tags following
the file's tags (size, name, etc.)		

1	`0x00`	trailing zero

The ususally included tags are: file name, file size, file type and file availability, mostly also file format and file type

.

Included tags can also be - depending on the file type:

Strings:	“artist”, “album”, “title”, “length”, “codec”
Integers:	“bitrate”

The search query can contain exact specifications for all of them, and more, if neccessary. Theoretically you could search just for the meta tags - you didn't even have to supply a search string. But this depends on the implementation of the searching code, and all tested clients refused with “no search string” or similar messages. But searches for all mp3 files are easily possible: the file extension is included in the search word database in also every known server, so an “s mp3” in the linux client does the trick. A search for all videos would so be made of three search queries: “s avi”, “s mpg”, “s divx” (in the text mode client), to cover the most common file extensions for video files.

4.2.3.TCP 0x34: users / files online

This message is sent on a regular basis.

bytes	content	meaning
1	`0x34`	tag: users/files online block
4		users online (unsigned int)
4		files online (unsigned int)

4.2.4.block type 0x38: write string

The string sent with this command is written on the screen of the client. The welcome message is generatet with a bunch of “write string” commands, for example.

bytes	content	meaning

1	`0x38`	tag: write string block
2		length of string
x	`„...“`	the string to write

4.2.5.block type 0x40: send client ip

The server sends the client its given ID. Mostly identical to the IP address of it. Should the client be firewalled the server generates unique ID’s starting from 1 counting up.

bytes	content	meaning
1	`0x40`	tag: client ip
4		the clients IP address

4.2.6.TCP 0x41: server name and description

Notice that the “server name” tag is simply the “name” tag. This tag is often used and changes meaning with its context.

bytes	content	meaning
1	`0x41`	tag: server description
16		probably server hash
4		server IP
2		server port
4	`02 00 00 00`	tag count: mostly 2
server name & server description tag come here		

4.2.7.TCP 0x42: return download sources

The answer to a TCP 0x19 query. A list of possible download sources holding a certain file. Notice that the source does not have to have the file completely - the server just has the information sent from the client, and this information only says “have file A with hash B size C and name D (and types/formats ...)”. So it may happen that the server has the file listed from 100 clients, 99 of them could have only the first 9 megs of it.

bytes	content	meaning
1	`0x42`	tag: download sources block

16		hash of the file whose sources follow
1		number of sources
for each source:		
4		IP address of client
2		port of client

4.3. UDP communication

4.3.1.UDP 0x96: PING

This packet is used by clients as well as by servers to verify their respective serverlist.

bytes	content	meaning
1	`0x96`	tag: ping
5		challenge

4.3.2.UDP 0x97: PONG

Also called “PING response”. It includes the challenge sent in the PING, and the information about how many users and files a server holds.

bytes	content	meaning
1	`0x97`	tag: pong (ping answer)
4		the received marker
4		users currently online
4		files currently online

4.3.3.UDP 0x98: extend text search

When a client does a text search and finds nothing on its server, it can extend the search to others. To be able to do a quick search on other servers this UDP message is used. The format is basically the same as the one of the TCP text search messages, but without the message size and with another type marker (0x98 instead of 0x16).

4.3.4.UDP 0x99: text search answer

The answer on 0x98 requests. As well as with the search the packet format is similar to the TCP equivalent without the size marker and a changed type marker.

4.3.5.UDP 0x9a: query sources

When searching for download sources the search is done within the whole network. To search on servers to which it is not connected the client sends this UDP message to other servers, which (may) respond with available download sources for the searched hash value.

bytes	content	meaning
1	`0x9a`	tag: query sources
16		the hash value to be searched

4.3.6.UDP 0x9b: query sources answer

The answer to 0x9a messages. If the server has download sources and spare bandwidth it sends this list of sources as an answer to the clients search request. As always the IP of the given sources is the ID the other client got from the other server. So if this ID is a low-ID (an ID far below one million, 5 for example) it indicates the other one is firewalled and cannot be contacted. So a “UDP callback request” (0x9c, see below) would be neccessary to set up the connection.

bytes	content	meaning
1	`0x9b`	tag: download sources answer
16		searched hash
1		number of results
	number of results *	
4		ID of download source
2		port of download source

4.3.7.UDP 0x9c: callback request

If the client A gets a 0x9b answer with a low-ID source of a file, it is not able to initiate the connection to. So it sends this message to the other server on which the other client - B - is logged in. This message now tells the server holding client B to send a TCP “callback request” to this client B, with the addresss of client A. Now client B initiates the connection to client A, and A is able to download the desired file from B. If client B logged out in the meantime this will not work - the TCP connection between the other server and client B is lost.

bytes	content	meaning
1	`0x9c`	tag: callback request
4		IP to connect
2		port to connect to
4		the client ID on the remote server which should initiate the connection

4.3.8.UDP 0x9e: callback failed

If the requested UDP callback failed (i.e. the desired client with the requested low-ID dropped), the server sends back a failure notification.

bytes	content	meaning
1	`0x9e`	tag: server list packet
4		the ID which couldn't be reached

4.3.9.UDP 0xa0: server announce

When a server comes online it announces itself with this message to other servers. These send in return serverlists and varying other information.

bytes	content	meaning
1	`0xa0`	tag: server announce
4		server IP
2		server port

4.3.10.UDP 0xa1: server list

This packet has exactly the same format like the TCP server list message. Besides the other packet type marker, of course.

bytes	content	meaning
1	`0xa1`	tag: server list packet
1		number of following servers
number of following *		
4		IP of a server
2		port of the server

4.3.11.UDP 0xa2: get server description

A two-byte UDP packet which requests the server description - which is server name and the server description string. The UDP data is simply the 0xe3 marker plus 0xa2.

4.3.12.UDP 0xa3: server description

The answer on 0xa2 requests. It justs holds the two stings - server name and server description. Do not confuse the server description with the welcome message delivered to the clients logging in via TCP - this is completely independant! And the welcome message can contain several strings as well.

bytes	content	meaning
1	`0xa3`	tag: server description
2		ushort: length name (x)
(x)		string: name of the server
2		ushort: length description (y)
(y)		string: description of the server

4.3.13.UDP 0xa4: get serverlist

Another one-byte UDP packet, which requests a serverlist of the addressed server. The serverlist is also sent as an answer to 0xa0 “server-announce” messages, but servers receiving 0xa0 messages seem to enter the sender into their serverlist, which they don’t seem to do when receiving a 0xa4 request. But this is unproven and may vary between different server implementations.