# PARIMULO: AUTOLOGIN E ANNOTAZIONI DEGLI UTENTI

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
CORRELATORE: Ing. Paolo Bertasi
LAUREANDO: Alessandro Di Pieri

Corso di Laurea Triennale in Ingegneria Informatica

A.A. 2009-2010


UNIVERSITA DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

*TESI DI LAUREA*

# PARIMULO: AUTOLOGIN E ANNOTAZIONI DEGLI UTENTI

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: Alessandro Di Pieri

A.A. 2009-2010

*Ai miei genitori.*

# Indice

# Sommario

Nella seguente tesi di laurea verrà illustrato il lavoro di implementazione di nuove funzionalità all'interno del modulo Mulo del progetto PariPari.
La relazione inizierà con una breve descrizione del progetto PariPari, seguita da una descrizione sul protocollo eDonkey e da una sull'organizzazione del modulo Mulo.
In seguito verranno trattati gli argomenti sviluppati ed implementati all'interno di questo progetto. Per prima cosa si tratterà della richiesta delle liste dei server ai server a cui ci si connette, illustrando, oltre alla sola implementazione del problema, anche la funzione generale di alcune classi di Mulo che vengono utilizzate e il modo di procedere che verrà tenuto per i compiti successivi. Successivamente si tratterà dell'algoritmo ideato per ottenere una classificazione dei server conosciuti e del suo utilizzo soprattutto per creare un metodo per connettersi automaticamente al server col maggior punteggio non appena si avvia Mulo, chiamato autologin intelligente.
Infine si tratterà dell'implementazione della possibilità di leggere e creare commenti sui file che si scaricano e che si condividono, in entrambi i protocolli supportati da Mulo, ossia quello eDonkey e quello kad.

# Capitolo 1

# Il progetto PariPari e il modulo Mulo

In questo primo capitolo verrà introdotto il progetto PariPari, specificando le sue caratteristiche e le motivazioni che hanno portato alla sua creazione.
Successivamente si spiegheranno le caratteristiche principali della rete eDonkey2000 e, infine, quelle del modulo Mulo.

## 1.1 Il progetto PariPari

Figura 1.1: Logo di PariPari

PariPari è una rete Peer-to-Peer serverless e multifunzionale, il cui software è scritto in linguaggio Java. Essendo la rete basata su una variante di Kademlia essa garantisce l'anonimato dei suoi nodi.

La caratteristica fondamentale di questo progetto è la multifunzionalità: esso si propone di rendere disponibili all'utente sulla rete tutti i più comuni servizi che oggi vengono utilizzati su reti diverse. L'utente potrà poi scegliere di volta in volta quale servizio utilizzare, grazie alla struttura altamente modulare che contraddistingue PariPari.

Un'innovazione importante è anche il sistema di crediti, che è più intelligente di quello delle reti eDonkey e, soprattutto, è multifunzionale.

Un'altra caratteristica rilevante è il fatto che PariPari sia scritto in Java: questo rende il progetto portabile su tutte le piattaforme, visto che i programmi scritti in Java non necessitano della ricompilazione del codice sorgente.

La modularità di PariPari si verifica tramite la sua suddivisione in plugin, programmi non autonomi che interagiscono con altri programmi per ampliarne le funzioni. Quindi grazie ad essa si riescono a creare servizi molto diversi, ma allo stesso tempo ben integrati e scalabili. I plugin permettono che il codice che viene modificato in uno di essi non vada ad interferire con il resto del progetto. Inoltre per ogni plugin viene definita una API[1], che permette agli altri plugin l'accesso alle sue funzionalità.

Il plugin centrale, attorno al quale tutti gli altri ruotano, è detto Core; per ottenere una risorsa (come ad esempio file, banda, RAM) ogni plugin deve inviargli una specifica richiesta, che esso andrà a concedere in base a particolari meccanismi di priorità gestiti dal plugin Credits.

1 API – Application Programming Interfaces

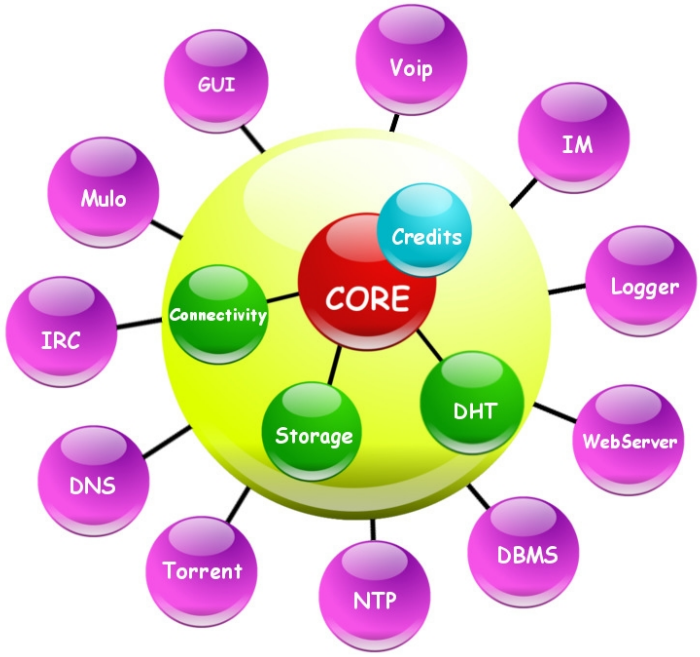


Figura 1.2: Architettura del progetto PariPari

Attorno al Core vi sono altri 3 plugin, detti “plugin della cerchia interna”, che dialogano direttamente col Core e sono i responsabili della fornitura delle risorse della rete: Connectivity instaura i socket[2] necessari per la comunicazione tra i peer della rete, Storage gestisce gli accessi alla memoria di massa, fornendo i file da utilizzare, DHT[3] permette di utilizzare una rete decentralizzata, stabile ed immune da guasti.

Infine vi sono i plugin che interagiscono direttamente con l'utente fornendo i vari servizi citati all'inizio, tra i quali Mulo, Torrent, Voip[4], IRC[5], IM[6], DBMS[7] e WebServer.

2 Socket – Astrazione software progettata per utilizzare delle API standard e condivise per la trasmissione e ricezione di dati attraverso una rete oppure come meccanismo di comunicazione tra processi
3 DHT – Distributed Hash Table
4 Voip – Voice Over IP
5 IRC – Istant Relay Chat
6 IM – Istant Messaging
7 DBMS – DataBase Managment System

## 1.2 La rete eDonkey2000

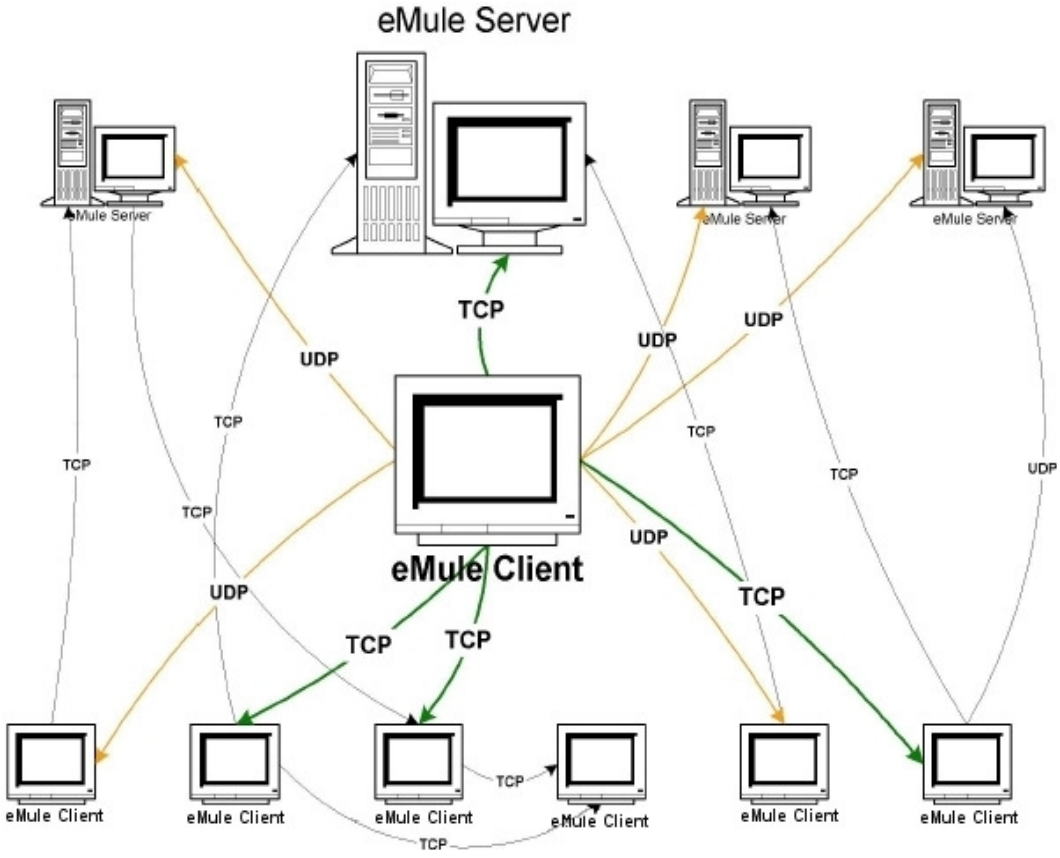


Figura 1.3: Comunicazioni client-server eMule

eDonkey2000 (detta anche eDonkey o ed2k) è una delle principali reti peer-to-peer. L'applicazione più utilizzata che si serve di questo protocollo è eMule, i cui sviluppatori, per fornire qualche caratteristica in più, hanno creato un'estensione al protocollo eDonkey, che porta proprio il nome di quest'applicazione.
I client eDonkey si connettono alla rete per condividere i file che si posseggono a livello locale e per ricercare ed eventualmente scaricare i file che vengono messi a disposizione da altri client.

I server hanno invece la funzione di rendere disponibili i file messi in condivisione da ciascun utente attraverso l'utilizzo di parole chiave che identifichino il file e la sua estensione e, inoltre, fungono da comunicazione per i client e permettono agli utenti di localizzare i file all'interno della rete.

Quando un client instaura una connessione con un server (via TCP[8]), quest'ultimo gli fornisce un identificatore a 4 byte, chiamato client ID e calcolato sulla base dell'indirizzo IP del client. Esistono 2 categorie di ID: high ID e low ID. I low ID vengono assegnati ai client che non accettano comunicazioni in entrata, perché, ad esempio, si trovano dietro un firewall che blocca tutte le porte. I client high ID non hanno restrizioni nell'utilizzo della rete e mantengono il loro ID fino a quando non cambieranno l'indirizzo IP. I client low ID, invece, sono molto limitati nell'utilizzo della rete, in quanto non possono connettersi a un altro client con low ID e anche alcuni server possono rifiutare la loro connessione.

Dopo la ricerca di un file da parte di un client (tramite parole chiave), il server gli risponde con una serie di risultati. Quando l'utente sceglie quello desiderato il server gli invia una lista di sorgenti che condividono questo file o parte di esso. Nel protocollo eDonkey2000, infatti, i file sono divisi in blocchi da 9500KiB, per ognuno dei quali viene calcolato un checksum MD4; questa è una funzione hash che si applica anche ad ogni singolo file ed è stata resa necessaria per distinguere i file diversi con lo stesso nome e per riconoscere i file uguali con nomi diversi.

Ogni client può scaricare da più fonti contemporaneamente e allo stesso modo, può far scaricare i propri file a più persone. Se le richieste ad un certo client sono in numero elevato si forma una coda d'attesa e solo una volta che si raggiunge la sua cima si può iniziare a scaricare. La velocità con cui si scala di posizione nelle code di attesa fino ad ottenere il diritto a scaricare, dipende dai crediti che il client da cui si scarica ha assegnano al client che vuole scaricare. Questi crediti dipendono da vari fattori tra cui: il tempo di attesa nella coda stessa, la priorità del file che si vuole scaricare e la quantità di byte che il client ha condiviso con gli altri utenti della rete, detto upload.

Non appena un blocco viene scaricato, esso viene automaticamente condiviso nella rete ed2k in modo da diffondere il più rapidamente possibile le fonti per quel file.

8 TCP – Trasmission Control Protocol

## 1.3 Il plugin Mulo

Figura 1.4: Il logo di Mulo

L'obiettivo del plugin Mulo è quello di creare un'applicazione per la condivisione di file, che utilizza le reti eDonkey e Kad e che possa competere con eMule, il più famoso client che utilizza queste due reti.
Per ottenere questo risultato, non essendo presente una documentazione eDonkey ufficiale, è stato necessario basarsi sul codice open-source di eMule per capire come un client debba funzionare e cercare di riportare ciò in linguaggio Java. Tuttavia bisogna sottolineare che Mulo non è una semplice copia di eMule (scritto in linguaggio C++) tradotto in Java, anche perché esso esiste già ed è chiamato jMule, ma presenta dei miglioramenti che possono fargli fare concorrenza al famoso eMule.

Le migliorie presenti fino a questo momento in Mulo rispetto a eMule sono le seguenti:

- presenza di un efficace motore di download che gestisce al meglio le fasi iniziali (hanno precedenza nell'essere scaricati i chunk[9] necessari alla visualizzazione di file multimediali) e le fasi finali (i chunk assegnati a fonti lente vengono riassegnati ad altri peer, in modo da evitare un eccessivo rallentamento alla fine)
- assenza della verifica finale dell'hash dell'intero file, che non è indispensabile e può richiedere molti secondi;
- presenza di code di upload più rapide e che hanno la possibilità di rifiutare l'upload a singoli peer;
- possibilità di connettersi in parallelo a più server in modo tale da riuscire ad ottenere un numero maggiore di fonti;

Le connessioni di Mulo avvengono sia via TCP che UDP. I pacchetti scambiati attraverso il protocollo eDonkey viaggiano soprattutto attraverso TCP e hanno una struttura ben precisa: tutti i pacchetti hanno lo stesso header, seguito poi da altri campi, diversi per ogni tipo di pacchetto. I campi più lunghi di un byte sono ordinati secondo l'ordinamento LittleEndian[10]. La struttura tipica di un header è illustrata nella tabella 1.1.

| **Campo** | **Size (in Byte)** | **Descrizione** |
|---|---|---|
| Protocol | 1 | Indica il protocollo del pacchetto; 0xe3 per il protocollo eDonkey, 0xc5 per l'estensione eMule, 0xd4 per i pacchetti compressi |
| Size | 4 | Indica la dimensione del pacchetto senza contare i sei byte dell'header |
| Type | 1 | Identifica in modo univoco il tipo di pacchetto |

Tabella 1.1: struttura tipica dell'header dei pacchetti TCP

9 Chunk – parte nella quale viene diviso un file. In eMule sono da 9.28MB tranne l'ultimo che è della grandezza restante

10 LittleEndian – ordinamento dei byte che prevede che la cifra meno significativa sia situata nell'ultimo indirizzo di memoria. I byte successivi sono inseriti per indirizzi crescenti.

Per i pacchetti UDP l'header è lo stesso senza la presenza, però, del campo Size.

Mulo supporta anche il protocollo Kademlia oltre a quello eDOnkey2000. Kademlia è una rete completamente distribuita (DHT) che viaggia su pacchetti UDP (con protocollo 0xe4). Le sue funzionalità principali sono quelle di cercare nella rete altre fonti per i file dato un certo codice hash, cercare file tramite parole chiave, cercare commenti e ratings (valutazioni) di file e allo stesso tempo pubblicare sulla rete i file, le parole chiave dei loro nomi e i commenti e le valutazioni che si danno loro, in modo che possano essere trovati da chi li cerca.

Attualmente Mulo supporta quasi completamente il protocollo eDonkey, mentre ancora incompleto è il protocollo eMule, infatti, per esempio, manca ancora l'implementazione del protocollo a 64 bit che non permette a Mulo di condividere file superiori ai 4GB.

Altre funzionalità che ora Mulo implementa sono supporto dell'AICH[11], supporto dei multipacket, supporto dei pacchetti compressi, supporto dell'identificazione sicura, supporto dei commenti, banning intelligente, login intelligente, ranking intelligente dei server, hashing parallelo e supporto dei crediti per gestire le code di upload.

Tutte le configurazioni di Mulo vengono salvate in dei file XML creati proprio per questo scopo: in essi si salvano, per esempio, nickname, porte UDP e TCP utilizzate, limiti di banda in upload e download, oltre che la lista di server conosciuti, la lista dei file condivisi e la lista dei download completati e in completamento.

Attualmente (alla release di Maggio 2010) Mulo è composto da 52 file per un totale di oltre 35000 righe di codice suddivise in 228 classi Java, ma a breve è prevista una nuova release che porterà questi dati ad aumentare ancora.

11 AICH – Advanced Intelligent Corruption Handling, serve per controllare che le parti scaricate siano prive di errori e per scaricare quelle corrette in caso di errore

# Capitolo 2

# Richiesta delle liste dei Server

In questo secondo capitolo si tratterà dell'implementazione della possibilità di richiedere l'invio delle liste dei server conosciuti da un server a cui ci vogliamo connettere nel momento in cui ci connettiamo ad esso.

## 2.1 Descrizione generale

In eMule esiste la possibilità di selezionare l'opzione “Aggiorna la Lista Server quando ti connetti a un Server”, che consiste, ogniqualvolta ci si connette a un diverso server, nell'inviare una richiesta specifica per ottenere tutti i server conosciuti da quel server, in modo tale da poter ampliare la propria gamma di scelta riguardo al server a cui connettersi per poter scaricare i file desiderati.
Poiché, come specificato all'inizio, eMule si basa soprattutto sul protocollo eDonkey2000, queste richieste vengono effettuate seguendo tale protocollo, mediante l'invio e la ricezione di pacchetti via TCP.

## 2.2 I pacchetti

Per creare i pacchetti nel progetto di Mulo in Java è stata sfruttata l'ereditarietà di Java, con la creazione di una superclasse Packet.java che memorizza le caratteristiche generali dei pacchetti, quali, ad esempio, protocollo (ed2k, eMule,

compressed, kad), tipo (TCP o UDP), grandezza del pacchetto, un puntatore per leggerlo e tags (una sorta di “minipacchetti” che servono ad aggiungere informazioni opzionali e sono contenuti all'interno di un pacchetto, spesso in una lista).

La classe Packet.java ha due sottoclassi, la classe PacketTCP e la classe PacketUDP, che estendono la superclasse. In entrambe sono dichiarati degli elenchi (in java rappresentati tramite la struttura enum) che associano a un valore di opcode[12] un tipo di pacchetto.

Infine esistono le classi che rappresentano i singoli pacchetti che estendono o la classe PacketTCP o la classe PacketUDP a seconda che i pacchetti viaggino via TCP o UDP. Il loro costruttore richiama quello della loro superclasse, specificando come parametro il protocollo che utilizzano e il tipo di pacchetto che rappresentano.

Nel caso della richiesta e risposta delle liste dei server sono stati creati due nuove classi: PacketTCPServersListRequest e PacketTCPServersListResponse.

Il pacchetto PacketTCPServersListRequest viene inviato dal client al server subito dopo la riuscita dell'handshake[13] solo nel caso in cui il client sia configurato in modo tale da abilitare l'espansione della sua Server List.

| **Name** | **Size in bytes** | **Default Value** | **Comment** |
|---|---|---|---|
| Protocol | 1 | 0xE3 | Protocol eDonkey2000 |
| Size | 4 | | The size of the message in bytes not including the header and size fields |
| Type | 1 | 0x14 | The value of the SERVERS_LIST_REQUEST opcode |

Tabella 2.1: struttura tipica dei pacchetti PacketTCPServersListRequest

Il valore del campo Size del pacchetto PacketTCPServersListRequest è 1, in quanto la lunghezza totale del pacchetto è di 6 byte, ma di questi non devo considerare i 4 del campo Size e quello del campo Protocol.

12 Opcode – Operation code: è il valore che identifica univocamente il tipo di pacchetto. Esistono opcode uguali per protocolli diversi.

13 Handshake – Procedura attraverso la quale due computer stabiliscono le regole comuni, ovvero la velocità, i protocolli di criptazione, di compressione, di controllo degli errori.
Prima di iniziare la connessione vera e propria tra due computer si crea questo tipo di connessione che consiste nella trasmissione dei pacchetti per regolare i parametri di connessione

Il pacchetto PacketTCPServersListResponse è inviato dal server al client e contiene informazioni riguardanti altri server eMule utilizzati per espandere la Server List del client.

| Name | Size in bytes | Default Value | Comment |
|---|---|---|---|
| Protocol | 1 | 0xE3 | Protocol eDonkey2000 |
| Size | 4 | | The size of the message in bytes not including the header and size fields |
| Type | 1 | 0x14 | The value of the SERVERS_LIST_RESPONSE opcode |
| Entry Count | 1 | NA | The number of servers described in this message |
| Server entries | (Entry Count)*6 | NA | Server descriptor entries each entry size is 6 bytes and contains 4 bytes IP address and then 2 byte TCP port |

Tabella 2.2: struttura tipica dei pacchetti PacketTCPServersListResponse

Il valore del campo Size di questo pacchetto varia in base al numero di server trasmessi.

## 2.3 La configurazione

Come scritto precedentemente in eMule esiste l'opzione per richiedere o meno l'invio della lista dei server dei server a cui ci connettiamo. Anche in Mulo quindi si è voluta creare questa opportunità.
In Mulo la classe contenente tutte le configurazioni del client che lo utilizza è Config.java. In questa classe sono presenti tutti i path da cui ricavare le informazioni salvate nei vari file xml, tra i quali anche quello in cui si salva la lista dei server, varie costanti (ad esempio costanti temporali) e alcune variabili booleane che indicano l'attivazione o meno di alcune opzioni disponibili.

Tra queste opzioni sono presenti quella che permette di attivare l'autologin e quella che permette di richiedere la lista dei server. Ma per cambiare le impostazioni non si può pensare di dover andare ogni volta a cambiare il valore della variabile sul codice sorgente: per questo è stata creata la classe Executor.java che fornisce numerosi comandi con le più svariate funzioni, tra le quali anche quella dell'attivazione o disattivazione della richiesta della lista dei server.
Il comando da utilizzare è “server.list y/n”

## 2.4 L'implementazione

Il protocollo eDonkey impone che i pacchetti vengano inviati in momenti ben precisi e secondo un preciso ordine. Se si sbagliasse il momento in cui inviarlo probabilmente si potrebbe non ottenere risposta, in quanto il ricevente, che può essere un server o un altro client, non se lo aspetta in quel momento. Per questa ragione è necessario studiare attentamente in che momento inviare la richiesta e in che momento attendere la risposta di un pacchetto quando si deve scrivere il codice per utilizzarlo.

In questo caso la richiesta della lista dei server e la relativa risposta avvengono subito dopo che la connessione tra client e server ha avuto successo ed è terminato l'handshake. Per prima cosa, dopo la connessione, il client offre al server tutti i file che mette in condivisione e subito dopo effettua al server la richiesta della sua Server List. Il server poi risponde inviando il suo stato e la sua versione, nonché un messaggio (di solito di benvenuto); subito dopo invia anche la sua lista dei server e fornisce altri dettagli per l'identificazione. Solo successivamente il client inizia a richiedere le fonti (altri client) da cui scaricare.

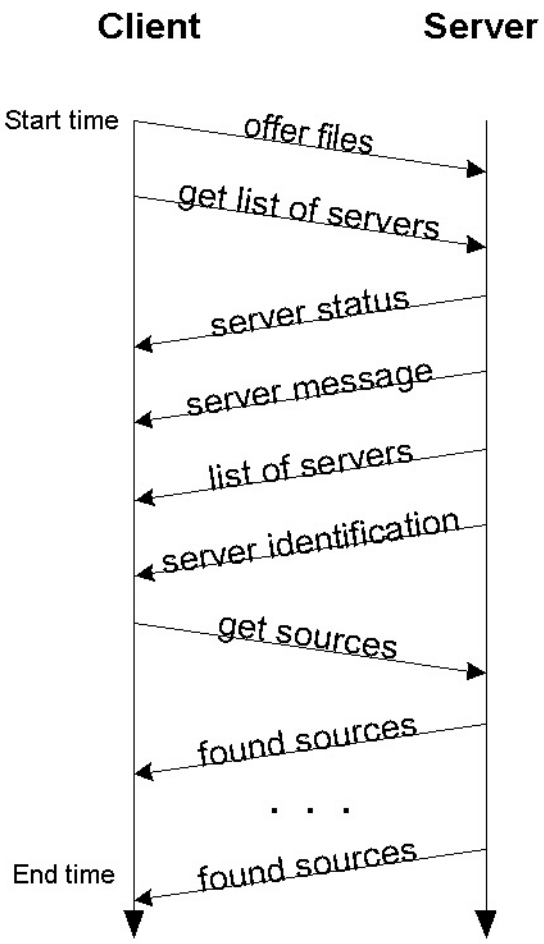


Figura 2.1 sequenza d'inizio della connessione (dopo l'handshake)

In Mulo la lista dei server è una hashtable che al primo avvio di PariPari carica i server da un file .met in cui è presente una lista dei server più importanti.

```
/** List of known servers. */
static Hashtable<Ed2kID,Server> list =
new Hashtable<Ed2kID,Server>(1);
```

Ed2kID è l'identificativo univoco per ogni server.

```
if (Config.serversListRequest) {
      int additions = Server.list.size();
      this.send(new PacketTCPServersListRequest()); //send a request to a
server for his server list
      PacketTCPServersListResponse serversList =
this.waitFor(PacketTCPServersListResponse.class); //server list
      for ( PacketTCPServersListResponse.Server server :
serversList.servers ) {
            Server.add(null, server.id, server.port);
      }
      additions = Server.list.size() - additions;
      Out.print("The server remote list contained " +
serversList.servers.size() + " servers, " + additions + " of which have been
added.");
}
```

Codice 2.1 frammento del metodo connect(boolean main) in Server.java

Osservando il codice si può notare che esso viene eseguito solo nel caso la variabile serversListRequest è true, cioè solo nel caso che l'opzione di richiesta delle Server List sia attiva.

Per inviare un pacchetto si utilizza il metodo send(PacketTCP), mentre per aspettare una risposta si usa il metodo waitFor(PacketTCP.class) che aspetta fino allo scadere di un breve timeout l'arrivo del tipo di pacchetto specificato.

Successivamente si aggiungono i server che sono stati inviati alla lista che già si possiede utilizzando il metodo add(String, Ed2kID, int), che aggiunge solo i server che non sono già presenti (riconoscendoli in base all'Ed2kID).

## 2.5 Test

Una volta scritto il codice, per verificare che esso sia corretto bisogna testarlo.

Nel plugin Mulo, dato che il suo sviluppo è a buon punto, c'è la possibilità di effettuare una doppia tipologia di test: i test jUnit e i test veri e propri utilizzando la versione di Mulo presente nel proprio pc in locale e verificando che essa funzioni a dovere.

I test jUnit servono a verificare soprattutto che il codice che viene scritto si comporti esattamente come si vuole. Essi sono unit test, ossia procedure utilizzate per verificare singole parti di un codice sorgente. Nella programmazione a oggetti, quale è Java, la più piccola unità è il metodo.

Nel caso dei pacchetti relativi alla richiesta e invio delle liste dei server, per verificare la correttezza della struttura dei pacchetti scritta, si sfrutta anche un altro strumento utile per l'analisi dei pacchetti di eMule: Wireshark. Quest'applicazione serve per catturare tutti i pacchetti trasmessi dal nostro computer alla rete e dalla rete al nostro computer in un dato periodo di tempo. In questo modo utilizzando eMule contemporaneamente a Wireshark si riescono a catturare i vari pacchetti eDonkey che vengono inviati, tra i quali quello di richiesta delle Server List e quello del loro invio.

Quindi in questo caso l'unità da verificare è il costruttore dei pacchetti, confrontando tramite il metodo assertEquals la lunghezza del pacchetto catturato con la lunghezza del pacchetto creato e in caso di errore esso restituisce la stringa che viene passata come primo argomento.

In caso di errore, inoltre, si ripete il procedimento per ogni byte del pacchetto in modo da trovare la posizione esatta dell'errore e poterlo così correggere più facilmente.

```
@Test
public void testServersListRequest() {
      byte[] pWireshark = new byte[] { (byte)0xe3 , 0x01 , 0x00 , 0x00 , 0x00
, 0x14 };
      this.p = new PacketTCPServersListRequest(pWireshark, null);
      byte[] pToBytes = this.p.toBytes().array();
      assertEquals("The reconstruct packet ServerList has a wrong length, ",
pWireshark.length, pToBytes.length);
      for ( int i = 0 ; i < Math.min(pWireshark.length, pToBytes.length) ;
i++ ) {
            assertEquals("Wrong byte in the reconstruct packet ServerList in
the position " + i + ", ", pWireshark[i], pToBytes[i]);
      }
      System.out.println("Packet ServersListRequest:");
      System.out.println(this.p.toString());
}
```

Codice 2.2 testServersListRequest() in PacketsTCPTest.java

```
@Test
public void testServersListResponse() {

	byte[] pWireshark = new byte[] {(byte)0xe3 ,0x25 ,0x00 ,0x00 ,0x00 ,
0x32 ,0x05 ,(byte)0xcf ,(byte)0xab ,0x3e ,(byte)0xf0 ,0x35 ,0x12 ,0x26 ,
0x6b ,(byte)0xa4 ,0x0a ,0x26 ,0x25 ,0x26 ,0x6b ,(byte)0xa4 ,0x15 ,0x35 ,
0x12 ,0x48 ,(byte)0xac ,0x59 ,(byte)0x81 ,0x35 ,0x12 ,0x48 ,(byte)0xac ,
0x58 ,(byte)0xbe ,0x35 ,0x12 ,(byte)0x00 ,0x00 ,(byte)0x00 ,0x00 ,
(byte)0x00 , };

	this.p = new PacketTCPServersListResponse(pWireshark, this.peerTest);
	byte[] pToBytes = this.p.toBytes().array();

	assertEquals("The reconstruct packet ServerList has a wrong length, ",
pWireshark.length, pToBytes.length);

	for ( int i = 0 ; i < Math.min(pWireshark.length, pToBytes.length) ;
i++ ) {
		assertEquals("Wrong byte in the reconstruct packet ServerList in
the position " + i + ", ", pWireshark[i], pToBytes[i]);
	}

	System.out.println("Packet ServersListResponse:");
	System.out.println(this.p.toString());
}
```

Codice 2.3 testServersListResponse() in PacketsTCPTest.java

# Capitolo 3

# Autologin intelligente e ranking dei server

In questo capitolo si tratterà dell'algoritmo scelto per fornire un punteggio ai vari server (detto ranking) e dello scopo principale per cui si effettua questa classificazione: il potersi collegare automaticamente all'avvio di Mulo al server migliore tra tutti quelli disponibili nella lista.

## 3.1 Ranking dei Server

### 3.1.1 Primo calcolo

La prima volta che si lancia Mulo, come detto precedentemente, vengono aggiunti una serie di server alla Server List di Mulo. Nella lista dei server presente nel file .met sono presenti vari elementi che contraddistinguono il server, come il suo indirizzo IP, la porta alla quale connettersi, il nome del server, il numero di utenti connessi e il numero di file condivisi dagli utenti connessi ad esso.
Per formare una prima classifica dei server dando loro un punteggio si è deciso che gli elementi fondamentali su cui basarsi sono il numero di file condivisi e il numero di utenti collegati. Questo punteggio potrà essere successivamente modificato in base ai tentativi di connessione andati a buon fine e a quelli falliti.
Il motivo per cui si sono scelte queste componenti per il calcolo del ranking è che

se si fosse considerato solo il numero di file questi sarebbero potuti essere molti, ma con un alto numero di fake e di file corrotti, così considerando anche il numero di utenti collegati si riesce ad avere un maggior controllo: presumibilmente un server con molti utenti collegati dovrebbe essere buono, soprattutto se ha anche molti file.
Quindi la decisione finale è stata quella di considerarli entrambi dando un peso leggermente maggiore al numero di file condivisi nel server e cercando di ottenere punteggi iniziali medi di poco sopra lo zero, così da poter ottenere successivamente una distinzione netta tra i server con punteggio positivo e quelli con punteggio negativo.

```
server.score = server.files / 100000 + server.users / 1000;
```

Questa scelta è stata fatta perché la maggior parte dei server che vengono aggiunti la prima volta hanno qualche decina di migliaia di utenti collegati e qualche milione di file.

## 3.1.2 Aggiornamento del punteggio

Poiché i server possono variare nel tempo (per esempio possono scomparire, possono aver problemi di linea) è stato deciso che il loro punteggio debba poter variare.
Per ogni server questo avviene ogniqualvolta ci si prova a connettere ad esso.

```
try {
      if (main) {
            currentServer = this;
      }
      this.connection = new Connection(null, this.ed2kID.ip, this.tcpPort);
      if (!this.doLogin()) { // handshake failed
            if (main) {
                  Log.printWarning("Error logging in, no LoginResponse packet
received");
                  currentServer = null;
            }
```

```
            this.connection.close(); // we're not connected at application
level, close TCP connection
            this.connection = null;
            failedLogins++;
            throw new IOException(); // bypass the rest
      }
      succeededLogins++;
      if (this.score < -50) {
            this.score = 0;
      }
      else if (this.score >= 2147483597) { //MAX_INT - 50
            this.score = 2147483647;
      }
      else if (main) {
            this.score += 50; // higher increment
      }
      else {
            this.score += 25;
      }

      // some other operations that could be done if login succeeds
      // for example ServerListRequest

      writeXML(); // update info
      return true;
}
catch (IOException e) { // already logged by new Connection()
      if (this.score >= 600) {
            this.score -= 100; // higher decrease
      }
      else if (this.score <= -2147483618) { //MIN_INT + 30
            this.score = -2147483648;
      }
      else {
            this.score -= 30; // lower decrease
      }
      writeXML(); // update info
      return false;
}
```

Codice 3.1 frammento del metodo connect(boolean main) in Server.java

Se il tentativo di connessione va a buon fine il punteggio del server interessato aumenta: se esso aveva un punteggio negativo, qualunque esso fosse passerà ad un punteggio nullo, perché con i punteggi negativi si vuole dare risalto a quei server a cui non ci si riesce a connettere. Il punteggio nullo, quindi indica soprattutto quei server che avevano punteggio negativo, ma con cui nell'ultima si è riusciti ad instaurare una connessione. Questo punteggio comporta che un nuovo fallimento riporterebbe il server a punteggi negativi a indicare la sua inaffidabilità.
Se il server aveva un punteggio positivo già precedentemente alla riuscita del tentativo di connessione, il suo punteggio aumenterà di 50 punti, il che, almeno inizialmente, consentirà a questo server di guadagnare parecchie posizione nella classifica dei server, visto che a parte poche eccezioni questi sono tutti molto vicini all'inizio. È stato inoltre instaurato un controllo per evitare l'overflow nel caso si arrivi al massimo punteggio rappresentabile con una variabile di tipo int: in quel caso il punteggio rimarrebbe fisso al valore MAX_INT, ossia 2147483647.
Inoltre è stato anche contemplato il caso in cui per cercare il file desiderato si usi la ricerca globale invece che la ricerca locale. In questo caso poiché si va a effettuare la ricerca su tutti i server conosciuti, si è deciso di aumentare comunque il punteggio per tutti i server a cui ci si riesce a connettere, ma in quantità dimezzata rispetto al caso in cui il server sia l'unico a cui ci connettiamo, quindi aumentandolo di 25 punti.
Nel caso in cui il tentativo di connessione al server fallisse si è deciso di togliere un punteggio maggiore (100 punti) ai server sopra i 600 punti, in modo tale da aumentare le possibilità di far loro perdere una posizione in classifica e al tempo stesso in modo da non penalizzarli troppo, visto che comunque i server sopra i 600 punti dovrebbero essere server a cui ci si connette spesso o con un alto numero di file e utenti.
Per tutti quelli sotto i 600 punti, invece, si è scelto di far perdere solo 30 punti, in modo tale che se il non riuscire a connettersi al server fosse solo un caso i punti persi potrebbero essere recuperati tranquillamente connettendosi la volta successiva, guadagnando anche qualche punto aggiuntivo. Per sicurezza si è voluto mettere anche il controllo sul minimo, in modo tale che se per caso qualche utente si ostinasse a cercare di connettersi a un server che non risponde (caso comunque molto poco probabile), arrivato al valore MIN_INT (-2147483648) il punteggio non scenderebbe più.

### 3.1.3 Utilizzo

Il fornire i server di un punteggio porta alcuni vantaggi; il più immediato è sicuramente quello di avere una panoramica della qualità dei server presenti nella Server List quando la si visualizza durante l'utilizzo di Mulo. Oltre a questo vantaggio sono stati pensati anche altri due metodi per rendere più comodo l'utilizzo di quest'applicazione. Il primo è l'autologin intelligente di cui si tratterà tra poco; il secondo è l'introduzione del comando "server.connect" senza parametri aggiuntivi, che è diverso del comando "server.connect id", dove id è la posizione nella Server List del server a cui ci vogliamo connetere.
La differenza è che il comando "server.connect" cerca di connettersi al server con maggior punteggio possibile e se non ci riesce a quello successivo, e così via fino a che non riesce a connettersi a uno o finiscono i server con punteggio positivo nella lista. Tutto ciò si ottiene richiamando il metodo Server.automaticLogin() che viene di solito utilizzato per l'autologin, di cui si tratterà ora.

## 3.2 Autologin intelligente

### 3.2.1 Descrizione

L'autologin intelligente è una diretta conseguenza del ranking dei server, in quanto la parola "intelligente" sta ad indicare che ci si connette al server migliore, ossia a quello con punteggio più alto.
L'autologin indica la possibilità, presente anche in eMule, di connettersi automaticamente ad un server non appena viene fatta partire l'applicazione, in questo caso Mulo. In realtà questo non porta particolari vantaggi, ma è molto comodo soprattutto se chi utilizza l'applicazione è un utente che si collega quasi

sempre agli stessi server, in quanto gli consente di risparmiare il tempo di dover aprire la lista dei server, cercare il server interessato e connettersi ad esso.

Come la richiesta delle Server List dei server a cui ci connettiamo, anche l'autologin è un'opzione che l'utente può decidere di attivare o meno, quindi anche per quest'opzione è presente un comando per attivarla o disattivarla. Questo comando è “config.autologin y/n”. Ovviamente la modifica delle configurazioni è disponibile solo dal successivo avvio di Mulo.

## 3.2.2 Implementazione

Poiché l'autologin riguarda la connessione ad un server, il metodo automaticLogin() è stato implementato nella classe Server.java.

```
/**
 * Tries to connect to some server with positive score.<br>
 * Order of attempts is unspecified.
 * @return Whether we are now connected to some server.
 */
static boolean automaticLogin() {
	// NO synchronized (list), we clone values!
	Out.printImportant("Trying to connect to a server with automatic login,
may take a few seconds, please wait...");

	LinkedList<Server> serverList =
(LinkedList<Server>)Utils.cloneList(list.values());

	Collections.sort(serverList);

	for ( int i = 0 ; i < serverList.size() ; i++ ) {
		if (i < 3 && serverList.get(i).ed2kID.isLow()) {
			if (i == 2) {
				if (serverList.get(0).score > 0 &&
serverList.get(0).connect(true)) {
					return true;
				}
```

```
                        else if (serverList.get(1).score > 0 &&
serverList.get(1).connect(true)) {
                              return true;
                        }
                        else if (serverList.get(2).score > 0 &&
serverList.get(2).connect(true)) {
                              return true;
                        }
                        else {
                              continue;
                        }
                  }
                  else {
                        continue;
                  }
            }
            else if (serverList.get(i).score > 0 &&
serverList.get(i).connect(true)) {
                  return true;
            }
      }
      return false;
}
```

Codice 3.2 metodo automaticLogin() in Server.java

Nell'implementazione del metodo per prima cosa si usa il metodo Utils.cloneList() per creare una LinkedList di Server al posto della hashTable in cui si salva la Server List. Nella LinkedList di Server, chiamata serverList, si trovano tutti i server presenti nella Server List nello stesso ordine in cui erano salvati, quindi senza seguire un ordinamento preciso. Per questo motivo subito dopo bisogna invocare il metodo della libreria standard Collections.Sort per ordinarla secondo il punteggio dei vari Server. Questo è reso possibile dalla sovrascrittura del metodo compareTo della classe Server.java.

```
/**
 * Compares two servers according to their score.
 * @param s The other server.
 * @return 1 if the other server has a higher score, -1 if it has a lower
score, 0 if they are equals
 */
public int compareTo(Server s) {
      if (this.score > s.score) {
            return -1;
      }
      else if (this.score < s.score) {
            return 1;
      }
      return 0;
}
```

Codice 3.3 metodo compareTo(Server s) in Server.java

Una volta che si ha la lista ordinata secondo il ranking dei Server dal punteggio maggiore al minore, si può iniziare ad eseguire l'algoritmo.

L'algoritmo si articola in vari casi, ma innanzitutto bisogna sottolineare che se il server a cui ci vuole provare a connettere ha punteggio negativo, l'autologin intelligente non tenta neanche di instaurare la connessione.

Inoltre un altro controllo, che viene effettuato solo per i primi tre server, è se l'id ed2k che si otterrebbe collegandosi a quel server è alto o basso.

Quindi nel caso migliore possibile ci si riesce a connettere subito al primo server con id alto. In questo caso il metodo termina restituendo true, che indica che l'autologin è riuscito.

Nel caso la connessione col primo o col secondo server avvenisse con id basso, si andrebbe a provare il successivo, nella speranza che sia con id alto. Se così fosse ci si proverebbe a connettere con quest'ultimo, altrimenti, se anche la connessione col terzo server avvenisse con id basso, significherebbe che molto probabilmente è il pc dell'utente a non consentire il possesso di un id alto, probabilmente a causa di un firewall, e quindi si proverebbe a connettersi al primo server della lista, poi al secondo e infine al terzo (ovviamente solo nel caso in cui il tentativo precedente fallisca) senza più considerare l'ed2kID.

Nel caso in cui si scorra tutta la lista fino ad arrivare ai server con punteggi negativi senza riuscire a connettersi a nessuno il metodo restituisce false e viene stampato a video il seguente avvertimento: "Could not connect to any server. Please check your connection" che suggerisce di controllare la propria connessione, in quanto non è stato possibile connettersi ad alcun server.

Quest'ultimo avvertimento non fa parte del metodo, ma viene stampato a video quando questo metodo restituisce false quando viene utilizzato nel metodo init() di Mulo, che è il metodo chiamato dal core per inizializzare Mulo, in cui vengono caricati i file XML e in cui vengono lanciati i thread di Mulo.

```
if (Config.autoLogin) {
      if (Server.automaticLogin()) {
            Server server = Server.currentServer;
            Out.printImportant("You are now connected to " + server.name + "
(" + server.getIP() + ":" + server.tcpPort + ").");
            if (Config.ed2kID.isLow()) {
                  Out.printImportant("Connected with a LowID. It's very
difficult to download something without a HighID.");
                  Out.print("Possible causes and solutions are:");
                  Out.print("- Mulo configuration is wrong =>
type \"config.tcp portNumber\" to change Mulo TCP port;");
                  Out.print("- you are behind a router => set it to forward
incoming messages on port " + Config.tcpPort + " to you;");
                  Out.print("- a firewall is blocking incoming connections =>
unlock port " + Config.tcpPort + " for Mulo;");
                  Out.print("- your public IP address ends with .0 => try
disconnect and reconnect to the Internet.");
            }
      }
      else {
            Out.printImportant("Could not connect to any server. Please check
your connection");
      }
}
```

Codice 3.4 frammento del metodo init() in Mulo.java

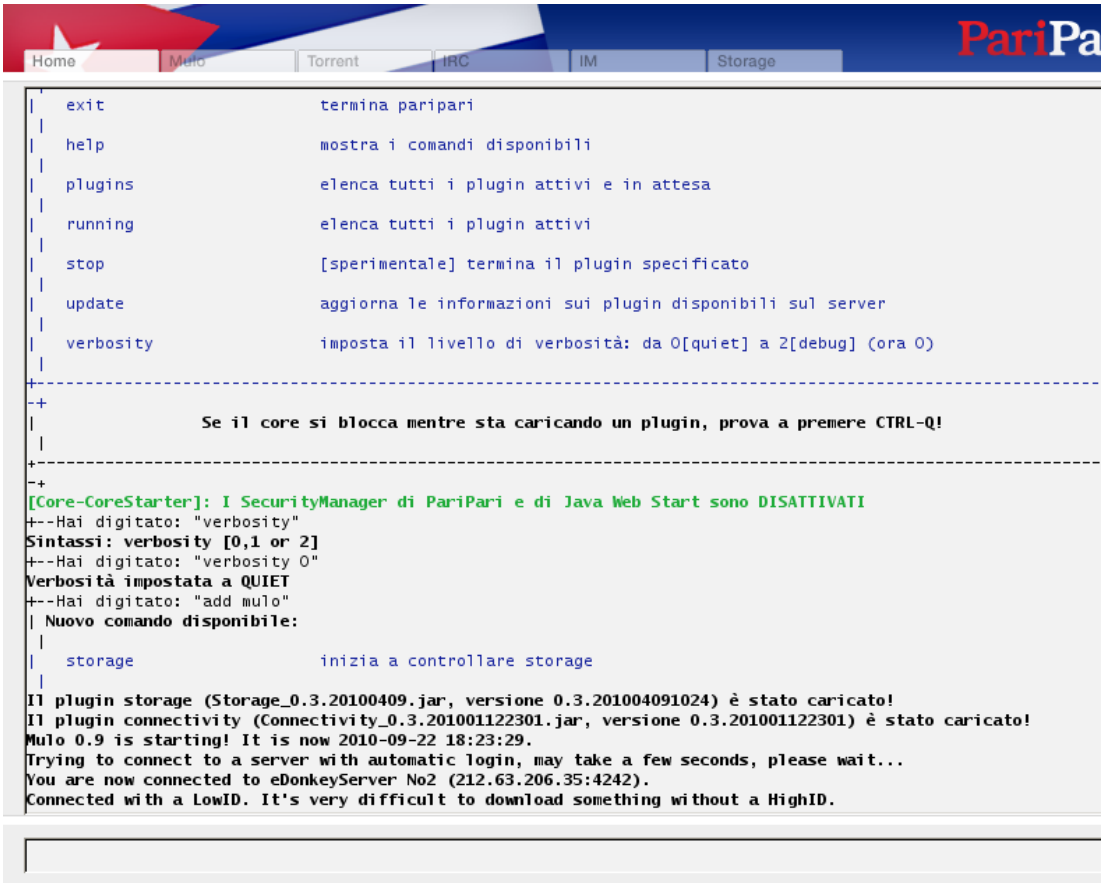


Figura 3.1 esempio di funzionamento corretto dell'autologin

# Capitolo 4

# Commenti eMule e Note Kad

In questo capitolo si tratterà della possibilità che hanno gli utenti di dare una valutazione e scrivere un commento sui file che condividono e delle modalità con cui ciò avviene: ci sono due metodi di diffusione dei commenti: tramite la rete eDonkey e tramite la rete kad.
Tutte le annotazioni degli utenti, comunque hanno una struttura ben specifica: per prima cosa compare la valutazione al file, un valore intero da 1 a 5, dove 5 è l'ottimo, obbligatorio se voglio scrivere anche un commento; il valore 0 sta ad indicare che il file che si condivide non ha un commento. Il secondo elemento che compare è il commento vero e proprio, che è facoltativo e non può superare i 128 caratteri di lunghezza. Il terzo elemento è il nome del file al quale è associato il commento, poi c'è il nickname dell'utente che ha scritto il commento e infine c'è la rete attraverso la quale è arrivato il commento: ed2k o kad. Alla rete ed2k è associato il valore 0, alla rete kad il valore 1.

## 4.1 Commenti eMule

### 4.1.1 Descrizione generale

I commenti eMule vengono diffusi attraverso la rete eDonkey. Essi vengono diffusi attraverso pacchetti specifici che viaggiano via TCP. Nel protocollo

eDonkey inizialmente non esistevano i commenti. Essi sono stati introdotti con eMule, poiché ci si accorse che la maggior parte dei file che giravano per la rete erano corrotti o fake. Infatti i pacchetti in cui viaggiano i commenti sono di tipo 0xC5, ossia del tipo dell'estensione di eMule.
I commenti eMule per un dato file non si possono cercare quando si vogliono, ma si ottengono solo dopo che si fa partire un download di un dato file e dopo che si sono contattate le fonti che hanno inserito i commenti desiderati (sempre che ce ne siano).

A differenza della richiesta delle liste dei server, in questo caso non è stato possibile catturare i pacchetti tramite Wireshark, perciò per l'implementazione in Java ci si è dovuti basare esclusivamente sul codice di eMule.

### 4.1.2 Il pacchetto

Il pacchetto che contiene i commenti è stato chiamato PacketTCPFileDescription, che viene inviato da un client a un altro client in due occasioni specifiche: nel primo caso subito dopo la richiesta del file da parte del client che scarica, c'è la risposta da parte del client che condivide seguita subito dall'invio del pacchetto PacketTCPFileDescription (solo se il file ha una valutazione diversa da 0, cioè solo se c'è un commento).
Nel secondo caso, subito dopo la richiesta di inizio upload da parte del client che scarica, c'è la risposta del client che condivide seguita subito dall'invio del pacchetto PackettCPFileDescription (sempre solo se il file ha una valutazione diversa da 0).

| Name | Size in bytes | Default Value | Comment |
|---|---|---|---|
| Protocol | 1 | 0xC5 | Protocol eMule extension |
| Size | 4 | | The size of the message in bytes not including the header and size fields |
| Type | 1 | 0x61 | The value of the FILE_DESCRIPTION opcode |
| File Rate | 1 | NA | A byte providing some kind of rating on the file |
| Comment Length | 2 | NA | The length of the comment that follows (no longer than 128) |
| Comment | Varies | NA | The comment itself |

Tabella 4.1: struttura tipica dei pacchetti PacketTCPFileDescription

Nel pacchetto basta inviare solo File Rate e Comment, non serve che vengano inviate anche le altre informazioni che fanno parte del commento, come il nome del file, il nome dell'utente e la rete utilizzata, perché le prime due si ricavano dal client da cui si scarica, mentre la terza si ricava dal tipo di pacchetto che si riceve (le note kad utilizzano pacchetti UDP e funzionano in modo completamente diverso).

Non esiste un pacchetto di risposta al pacchetto PacketTCPFileDescription per confermarne la ricezione.

### 4.1.3 La configurazione

Nel mondo ci sono molte versioni di client che implementano il protocollo eDonkey2000, basti pensare solo a tutte le diverse versioni di eMule e aMule, che, nonostante il codice venga aggiornato, devono comunque rimanere compatibili con il protocollo. Queste varie versioni, però, non è detto che abbiano tutte le stesse opzioni implementate, il che comporterebbe la ricezione di pacchetti sconosciuti che potrebbero bloccare il sistema.

Per questa ragione sono stati creati i campi MISC_OPTIONS_1 e MISC_OPTIONS_2, che vengono inviati dal client o dal server sotto forma di tag nel pacchetto Hello nel momento della connessione con un altro client.

| Numero bit | Significato |
|---|---|
| 31-29 | AICH version |
| 28 | Unicode |
| 27-24 | UDP version |
| 23-20 | Data Compression version |
| 19-16 | Source Ident |
| 15-12 | Source Exchange |
| 11-8 | Ext. Request |
| 7-4 | Comments |
| 3 | Peer Cache supported |
| 2 | No “View Shared Files” supported |
| 1 | MultiPacket |
| 0 | Preview |

Tabella 4.2: struttura tipica dei bit del tag MISC_OPTIONS_1

Come si può osservare dalla Tabella 4.2 il tag MISC_OPTS_1 serve soprattutto per specificare che tipi di versione si implementano per alcune funzioni e per indicare se altre funzionalità sono implementate o meno. Poiché in questo caso si sta trattando dei commenti il campo che verrà preso in considerazione è solo quello dei bit compresi tra il settimo e il quarto.
Inizialmente, prima di iniziare il lavoro di implementazione dei commenti, questi bit erano tutti impostati a 0, quindi era impedito l'invio e la ricezione dei commenti. Poi studiando il codice di eMule, si è notato che per permettere il loro utilizzo basta impostare a 1 il bit 4.

### 4.1.4 L'implementazione

Seguendo il codice di eMule si è osservato che il pacchetto PacketTCPFileDescription può venire inviato in due casi particolari: dopo una richiesta del file e dopo una richiesta di inizio upload da parte di un client all'altro.

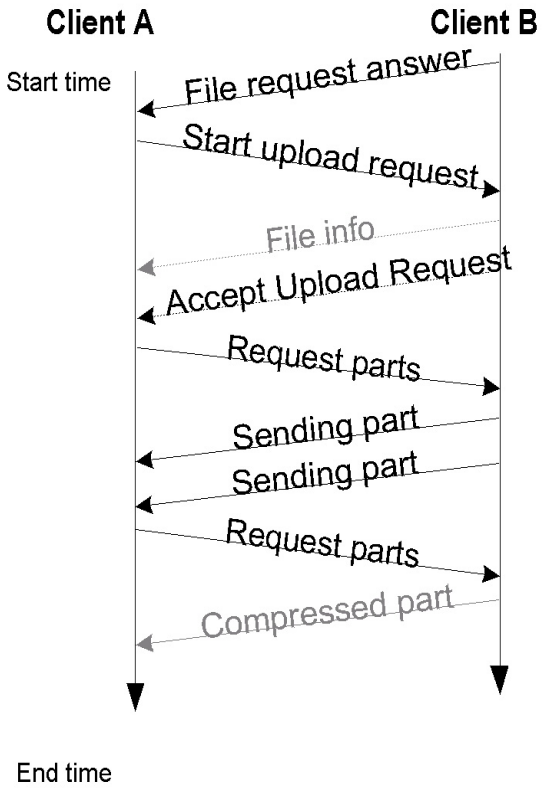


Figura 4.1 Inizio download subito dopo la richiesta di un file

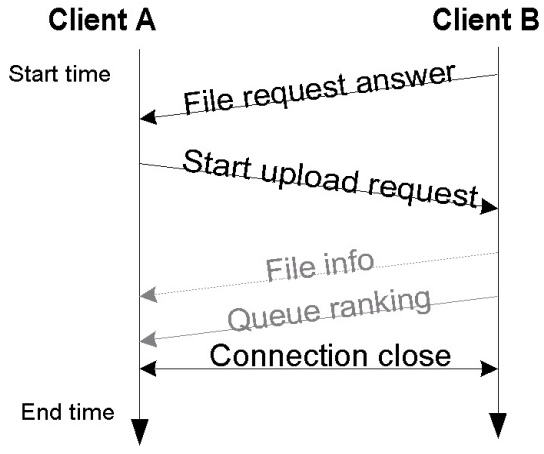


Figura 4.2 Coda d'attesa dopo la richiesta di un file

La richiesta del file è una File Request, in cui il client che deve scaricare invia al client che condivide il file cercato l'ID del file cercato e le parti che ha già scaricato.

La risposta contiene a sua volta l'ID del file cercato e il nome del file. In più questa può essere accompagnata anche da un pacchetto di File Status, che informa sullo stato delle parti del file da scaricare, per esempio specificando quali sono disponibili e quali no.

```
private boolean sendFileRequest(DownloadSession downloadSession) {
	if (!this.send(new PacketTCPFileRequest(downloadSession.download,
this.supportsExtendedRequestsVersion()))) {
		return false;
	}

	PacketTCP response = this.waitFor(PacketTCP.class);

	if (response == null) {
		return false;
	}

	else if (response instanceof PacketTCPFileNotFound) { // not a source!
		downloadSession.destroy();
		return false;
	}

	else if (response instanceof PacketTCPFileResponse) {

		downloadSession.fileName = new
String(( (PacketTCPFileResponse)response ).fileName);

		PacketTCPFileDescription comm =
this.waitFor(PacketTCPFileDescription.class);

		if (comm != null) {
			downloadSession.comment = Comment.receiveComment(comm,
downloadSession);
		}
	}
```

Codice 4.1 frammento del metodo sendFileRequest(DownloadSession) in Peer.java

```
/**
 * Creates an object Comment from a received PacketTCPFileDescription in an
open DownloadSession
 * @param comm The PacketTCPFileDescription
 * @param downloadSession The DownloadSession where we receive the comment
 * @return The new comment
 */
public static Comment receiveComment(PacketTCPFileDescription comm,
DownloadSession downloadSession) {

	return new Comment(downloadSession.source, comm.fileRate,
Comment.getCommentString(comm.comment), downloadSession.fileName,
downloadSession.source.nickname, (byte)0);

	}
```

Codice 4.2 metodo receiveComment(PacketTCPFileDescription, DownloadSession) in Comments.java

```
/** Replies to a file request. */
private boolean answerFileRequest(PacketTCPFileRequest packet) {
	Shared requestedFile = Shared.hashesList.get(packet.fileHash);

	if (requestedFile == null || requestedFile.availableParts == 0) { //
file not available
		return this.send(new PacketTCPFileNotFound(packet.fileHash,
this.supportsExtendedRequestsVersion()));
	}

	if (this.downloadSession != null &&
this.downloadSession.file.md4Hash.equals(packet.fileHash) && packet.sources >
0 && packet.partBitmap != null) {
		// he is sending us its file status of a file we are downloading,
take note!
		this.setAvailableParts(this.downloadSession, packet.partBitmap);
	}

	boolean sent = this.send(new PacketTCPFileResponse(requestedFile));
	this.send(new PacketTCPFileDescription(requestedFile.comment));
	return sent;
}
```

Codice 4.3 metodo answerFileRequest(PacketTCPFileRequest) in Peer.java

Come si può notare dal frammento di codice del metodo sendFileRequest una volta che invio la richiesta mi aspetto in risposta un pacchetto TCP e se quest'ultimo è di tipo PacketTCPFileResponse, allora mi aspetto anche che possa arrivare un pacchetto di tipo PacketTCPFileDescription contenente una valutazione e un commento del file.
Se così avviene si crea il nuovo commento per il file cercato nella downloadSession aperta con il client che ce lo ha inviato.

La richiesta di inizio upload è una Start Upload Request, in cui il client che deve scaricare invia al client che condivide il file cercato l'ID del file cercato come inizio della procedura di download.
L'arrivo della risposta (un pacchetto di tipo answerStartUploadRequest) indica che la richiesta è stata accettata e che ora si aspetta la richiesta delle parti che si cercano.

```
private boolean startUploadRequest(DownloadSession downloadSession) {
      if (downloadSession.download.status != Download.FileStatus.ACTIVE) { //
no need
            return true;
      }

      if (!this.send(new
PacketTCPUploadRequest(downloadSession.file.md4Hash)))     {
            return false;
      }

      PacketTCP response;
      while (( response = this.waitFor(PacketTCP.class) ) != null) {

            if (response instanceof PacketTCPFileDescription) {

                  PacketTCPFileDescription comm =
(PacketTCPFileDescription)response;

                  downloadSession.comment = Comment.receiveComment(comm,
downloadSession);
            }
```

```
            if (response instanceof PacketTCPQueueRanking) {
                  downloadSession.queueRanking =
( (PacketTCPQueueRanking)response ).rank;
                  return true;
            }

            else if (response instanceof PacketTCPUploadAccept) {
                  downloadSession.start(); // put it in the active sources
list and start right away :)
                  return true;
            }
```

Codice 4.4 frammento del metodo startUploadRequest(DownloadSession) in Peer.java

Come si può notare dal frammento di codice del metodo startUploadRequest se si riesce ad inviare un PacketTCPUploadRequest ci si aspetta una riposta, che può essere un PacketTCPFileDescription. In quel caso si crea il nuovo commento per il file cercato nella downloadSession aperta con il client che ce lo ha inviato.
Nella risposta alla startUploadRequest, quindi il metodo answerStartUploadRequest, il cui codice non è stato riportato in quanto la parte riguardante l'invio del commento è molto simile a quella nel metodo answerFileRequest, il pacchetto contenente il commento e la valutazione viene inviato solo nel caso di creazione di una nuova uploadSession, altrimenti si presume che sia già stato inviato e quindi non viene spedito nuovamente.

## 4.2 Note Kad

### 4.2.1 Descrizione generale

Come anticipato precedentemente in kad le valutazioni e i commenti degli utenti ai file sono gestiti in modo completamente diverso rispetto al protocollo eDonkey2000.

Innanzitutto bisogna sottolineare che kad non viaggia su TCP come eDonkey, ma su UDP.
In kad tutti i client che vogliono pubblicare una valutazione e un commento su un certo file che condividono, non devono, come nel protocollo eDonkey, aspettare che un altro client chieda loro il file per poi inviargli il commento sotto forma di PacketTCPFileDescription. In kad, invece esiste l'atto di pubblicazione vero e proprio, in cui i client immettono nella rete dei pacchetti in cui scrivono valutazione, eventuale commento, nome del file e nome dell'utente che ha inserito quel commento.
La ricerca dei commenti, infatti, non è necessario che venga effettuata solo nel momento in cui si è messo in download un file, ma si può anche effettuare su ogni singolo risultato che si ottiene tramite una ricerca, così da evitare di iniziare a scaricare file corrotti o fake.

### 4.2.2 I pacchetti

Per implementare le note in kad è stato necessario creare tre nuovi pacchetti, ovviamente tutti che viaggiano su UDP: PacketUDPKad2SearchNotesRequest, PacketUDPKadSearchNotesResponse e PacketUDPKad2PublishNotesRequest. Il pacchetto di risposta al PacketUDPKad2PublishNotesRequest è PacketUDPKad2PublishResponse, che è unico anche per la pubblicazione di fonti e parole chiave e che quindi era già stato implementato, così è bastato solo modificarne lievemente la gestione in modo che includesse anche il caso il cui si pubblichi una nota.

| Name | Size in bytes | Default value | Comment |
|---|---|---|---|
| Protocol | 1 | 0xE4 | Protocol Kad |
| Type | 1 | 0x45 | The value of the KAD2_PUBLISH_NOTES_REQUEST opcode |
| FileID | 16 | NA | Kad ID of the file |
| SourceID | 16 | NA | Kad ID of the source that owns the file |
| TagCount | 1 | NA | Number of tags that are in the packet |
| Tags | Varies | NA | List of tags containing some information about the file: it should contains FILE_NAME, FILE_SIZE, FILE_RATING and FILE_DESCRIPTION |

Tabella 4.3: struttura tipica dei pacchetti PacketUDPKad2PublishNotesRequest

Gli ID Kad sono differenti dagli ID eDonkey, anche se entrambi sono lunghi 128 bit: gli ID eDonkey sono codificati in MD4Hash, mentre gli ID kad sono Int128 che utilizzano la classe BitSet di Java.
Il tag FILE_DESCRIPTION è facoltativo da inviare.

| Name | Size in bytes | Default value | Comment |
|---|---|---|---|
| Protocol | 1 | 0xE4 | Protocol Kad |
| Type | 1 | 0x35 | The value of the KAD2_SEARCH_NOTES_REQUEST opcode |
| TargetID | 16 | NA | Kad ID of the file |
| FileSize | 8 | NA | Size of the file |

Tabella 4.4: struttura tipica dei pacchetti PacketUDPKad2SearchNotesRequest

Ovviamente il targetID si riferisce all'ID kad del file di cui stiamo cercando i commenti.

| Name | Size in bytes | Default value | Comment |
|---|---|---|---|
| Protocol | 1 | 0xE4 | Protocol Kad |
| Type | 1 | 0x3A | The value of the KAD_SEARCH_NOTES_RESPONSE opcode |
| TargetID | 16 | NA | Kad ID of the file |
| Entries | 1 | NA | Number of results found |
| SourceID | 16 | NA | Kad ID of the source that owns the file |
| TagCount | 1 | NA | Number of tags that are in the packet |
| Tags | Varies | NA | List of tags containing some information about the file: it should contains FILE_NAME, FILE_RATING and FILE_DESCRIPTION |

Tabella 4.5: struttura tipica dei pacchetti PacketUDPKadSearchNotesResponse

Il campo entries specifica il numero di risultati trovati e per ogni risultato si hanno il SourceID e i vari tag.

### 4.2.3 L'implementazione

Per implementare la pubblicazione delle note si è voluto per prima cosa creare una sottoclasse della classe KadSearch chiamata KadStoreNotesSearch, il cui costruttore imposta il tipo di KadSearch da effettuare come STORENOTES, imposta il suo target trasformando l'id di tipo MD4Hash passato come parametro in Int128 e infine salva la lista di tags che viene passata parametro.

```
class KadStoreNotesSearch extends KadSearch {

	final LinkedList<Tag> tags; //only for STORENOTES

	KadStoreNotesSearch(MD4Hash hash, LinkedList<Tag> fileTags) {
		super(KadType.STORENOTES);
		this.targetID = hash.toInt128();
		this.tags = fileTags;
	}
}
```

Codice 4.5 classe KadStoreNotesSearch

Successivamente i dati che vengono salvati vengono utilizzati nel metodo perform(), ossia nel metodo che avvia una ricerca Kad, che in questo caso specifico sarà di tipo STORENOTES. Quindi questi dati verranno utilizzati per creare e inviare un pacchetto di tipo PacketUDPKad2PublishNotesRequest che pubblica nella rete kad un commento e una valutazione di un dato file presente nella Shared List.

Il costruttore di KadStoreNotesSearch viene usato solo nella classe Shared nel metodo kadPublishFile, che viene utilizzato per pubblicare nella rete kad le parole chiave, le fonti e le note del file che sta venendo condiviso (ovviamente funziona solo se si è connessi alla rete kad).

```
if (sharedFile.comment != null) {
      //Publishes notes
      LinkedList<Tag> notesTags = new LinkedList<Tag>();
      notesTags.add(new TagString(Name.NAME, sharedFile.getName()));
      notesTags.add(new TagInt(Name.FILE_RATING, sharedFile.comment.rating));

      if (sharedFile.comment.comment != null) {
            notesTags.add(new TagString(Name.FILE_DESCRIPTION,
sharedFile.comment.comment));
      }

      notesTags.add(new TagInt(Name.FILE_SIZE, (int)sharedFile.size));

      KadSearch kSearch = new KadStoreNotesSearch(sharedFile.md4Hash,
notesTags);

      if (!kSearch.perform()) {
            Log.printError("Notes publishing on Kad of file " +
sharedFile.getName() + " failed.");
            published = false;
      }
}
```

Codice 4.6 frammento del metodo kadPublishFile(Shared) in Shared.java

Come si può notare dal frammento di codice del metodo sopra citato, se il file che si decide di condividere non ha alcun commento, allora si evita tutta la parte di

pubblicazione delle note. In caso contrario si crea una lista di tags in cui si inseriscono dei nuovi tag che indicano il nome, la valutazione, un eventuale commento e la dimensione del file condiviso. Dopodiché si crea la KadSearch di tipo STORENOTES passandogli come parametri l'hash md4 del file e la lista di tags appena creata e la si fa iniziare col metodo perform().

Anche per quanto riguarda la ricerca si è creata una nuova classe, sottoclasse di KadSearch, chiamata KadNotesSearch, che, come la classe KadStoreNotesSearch, ha un costruttore con il compito di definire il tipo di KadSearch che si effettua (in questo caso è di tipo NOTES), di definire il target e, solo per la ricerca di questo tipo, di definire la dimensione del file.

```
class KadNotesSearch extends KadSearch {

	final long fileSize; //for NOTES search

	KadNotesSearch(MD4Hash hash, long size) {
		super(KadType.NOTES);
		this.targetID = hash.toInt128();
		this.fileSize = size;
	}
}
```

Codice 4.7 classe KadNotesSearch

Quindi anche per la ricerca delle note i dati che vengono salvati grazie a questa classe vengono utilizzati nel metodo perform() di KadSearch solo nel caso, ovviamente, in cui questa sia di tipo NOTES. Quindi questi dati verranno utilizzati per creare e inviare un pacchetto di tipo PacketUDPKad2SearchNotesRequest che cerca nella rete kad tutti i nodi (ossia altri client) che hanno pubblicato un commento e una valutazione sul file con l'ID kad indicato nel pacchetto.
Questo pacchetto può essere utilizzato per cercare i commenti sia dopo l'inizio di un download del file, sia dopo la ricerca tramite parole chiave.

```
if (!Kad.isConnected()) {
      Out.printImportant("Cannot search: kad is not active.
Write \"Config.Kad y\" to activate it");
      return;
}
Out.printImportant("Kad Notes search is starting...");
KadSearch search = new KadNotesSearch(download.file.md4Hash,
download.file.size);
search.perform();
Out.printImportant("The search is finished");

if (KadSearch.comments.isEmpty()) {
      Out.printImportant("No comments for this file are available.");
}
```

Codice 4.8 frammento del metodo downloadSearchKadNotes(String[] args) in Executor.java

In questo frammento di codice del metodo downloadSearchKadNotes, metodo per la ricerca delle note kad dei file in download, si può osservare che per prima cosa si applica un controllo per controllare di essere connessi alla rete Kad, altrimenti tutto ciò che si vuole fare dopo è inutile, e solo dopo ciò si passa alla creazione di una KadSearch utilizzando il costruttore della sottoclasse KadNotesSearch, così da creare una ricerca di tipo NOTES.

Successivamente nel metodo si passa a controllare la lista comments in KadSearch, che è la lista in cui vengono salvati tutti i commenti che si ricevono in risposta alla ricerca effettuata: se è vuota significa che nessun client ha un commento sul file cercato, altrimenti nel metodo si procede con lo stampare a video i commenti esistenti.

Allo stesso modo funziona il metodo searchKadNotes, che serve per cercare le note dei file che si trovano nella lista dei risultati di una ricerca.

La lista dei commenti trovati viene quindi riempita attraverso i PacketUDPKadSearchNotesResponse, che si ottengono in risposta ai PacketUDPKad2SearchNotesRequest.

Per quanto riguarda i pacchetti PacketUDPKadSearchNotesResponse, poiché essi non fanno parte di un tipo di ricerca, ma solo di una risposta ad essa, non si è creata una sottoclasse di KadSearch come per le altre tipologie di pacchetti. Si è

deciso di creare, invece, un metodo apposito nella classe KadSearch, chiamato addNotesResult, la cui funzione è quella di analizzare il pacchetto ricevuto ricavandone così tutte le informazioni relative ai nomi del file, alle valutazioni, ai commenti e ai client che le hanno pubblicate.

```
static int addNotesResults(Int128 target, LinkedList<SearchResult>
resultsList) {
      int addedResults = 0;
      MD4Hash fileHash = target.toMD4Hash();
      for ( Download dl : DownloadManager.active ) {
            if (fileHash.equals(dl.file.md4Hash)) {
                  for ( SearchResult result : resultsList ) {

                        Tag tagName = result.getTag(Tag.Name.NAME);
                        Tag tagDescription =
result.getTag(Tag.Name.FILE_DESCRIPTION);
                        Tag tagRating = result.getTag(Tag.Name.FILE_RATING);
                        Tag tagIp = result.getTag(Tag.Name.KAD_SOURCE_IP);
                        Tag tagTcpPort =
result.getTag(Tag.Name.KAD_SOURCE_PORT);

                        if (tagTcpPort == null || tagIp == null ||
tagTcpPort.getNumber() < 0 || tagTcpPort.getNumber() > 65536) {

                              if (tagDescription != null) {
                                    Comment comm = new Comment(null,
(byte)tagRating.getNumber(), tagDescription.getString(), tagName.getString(),
null, (byte)1);
                                    comments.add(comm);
                              }
                              else {
                                    Comment comm = new Comment(null,
(byte)tagRating.getNumber(), null, tagName.getString(), null, (byte)1);
                                    comments.add(comm);
                              }
                        }
                        else {
                              Peer source;
                              Ed2kID tagId = new Ed2kID(tagIp.getNumber());
                              source = Peer.addPeer(tagId, Peer.From.KAD);
```

```
                                source.tcpPort = tagTcpPort.getNumber();
                                Comment comm = new Comment(source,
(byte)tagRating.getNumber(), tagDescription.getString(), tagName.getString(),
source.nickname, (byte)1);
                                comments.add(comm);

                                if (dl.activeSources.containsKey(source.ed2kID)
|| dl.queuingSources.containsKey(source.ed2kID)) {
                                      continue; // known source, skip
                                }

                                // New source
                                if (source.isBanned()) {
                                      continue;
                                }

                                if (source.downloadSession == null) {
                                      source.downloadSession = new
DownloadSession(source, dl);
                                      source.server = null;
                                      source.failures = 0; // start new
                                }

                                source.downloadSession.comment = comm;
                          }

                          addedResults++;
                          if (addedResults == 50) {
                                break;
                          }

                          if (dl.status != FileStatus.ACTIVE ||
dl.activeSources == null) { // download has been stopped
                                break;
                          }
                    }
              }
        }
        return addedResults;
}
```

Codice 4.9 metodo addNotesResult(Int128,LinkedList<SearchResult>) in KadSearch.java

Nel metodo qui riportato si è deciso di effettuare numerosi controlli sui casi in cui non si riceva qualche tipo di tag, come il tag relativo al commento, che è facoltativo, o quelli relativi alla fonte. In quel caso si salverà il commento senza specificarne chi è l'autore, ma solo con valutazione, eventuale commento, nome utente e rete (quest'ultima con valore 1, che significa rete kad, mentre valore 0 significava eDonkey).
Importante è sottolineare che il massimo numero di note kad che si possono avere su un file è 50, anche se comunque è veramente raro arrivare anche a 5 commenti, quindi è un limite che probabilmente non verrà mai raggiunto.
Il collegamento tra questo metodo e il pacchetto PacketUDPKadSearchNotesResponse è dato dalla resultsList di tipo SearchResult: infatti si è deciso di salvare in una lista di questo tipo tutti i risultati che si ottengono nel momento in cui si riceve un pacchetto di risposta alla ricerca di note kad, a parte l'id del file cercato che viene salvato direttamente nella classe PacketUDPKadSearchNotesResponse. Quindi ogni elemento della lista contiene, se disponibile, l'id della fonte che ha pubblicato un commento e la lista dei tag che ha pubblicato.

## 4.3 Comandi utili

Per la gestione dei commenti è necessario avere dei comandi che permettono di farlo. Per questo è stato necessario creare dei nuovi comandi da inserire nella classe Executor.
È stato creato il comando "shared.addcomment id rating comment" (abbreviato sh.ac) per aggiungere un nuovo commento al file che condividi che si trova nella posizione id nella Shared List. Se il file su cui utilizzi questa funzione ha già un commento, lo sostituisce con il nuovo. Inoltre aggiorna il file XML della Shared List inserendovi anche il nuovo commento, in modo che la volta successiva che si utilizza Mulo, il commento inserito venga caricato insieme alla Shared List.
Un altro nuovo comando che è stato creato è "shared.deletecomment

id"(abbreviato sh.dc) con cui rimuovi un commento dal file in posizione id nella Shared List.

Per visualizzare il commento di un file che condivido che si trova nella posizione id è stato modificato il comando "shared.info id"  in modo che ora mostri alla fine anche rating e commento.

Per quanto riguarda la rete eDonkey per vedere i commenti dei file in download è stato aggiunto il comando "download.comment id"(abbreviato dl.cm), dove id è la posizione del file nella Download List. Ovviamente il commento non può apparire prima di essere in coda nel o prima di iniziare a scaricare dal client che lo ha inserito.

Per quanto riguarda la rete kad se è disattivata l'autopubblicazione, ossia la pubblicazione automatica di fonti, parole chiave e note dei file che vengono condivisi non appena ci si connette alla rete kad, allora non appena si aggiunge un commento ad un file in condivisione, se lo si vuole pubblicare subito è presente il comando "shared.publish id" (abbreviato sh.p), dove id è la posizione del file nella Shared List.

Per cercare un commento nella rete kad, inoltre, sono stati introdotti due comandi: "download.kadnotes id" (abbreviato dl.kn), che cerca le note kad del file nella Download List nella posizione id e “search.kadnotes searchID resultID” (abbreviato s.kn), che cerca le note kad del file trovato nella ricerca searchID e nella posizione resultID (i due parametri sono necessari Perché si possono effettuare più ricerche e per ogni ricerca possono ovviamente esserci più risultati). La caratteristica di questi ultimi due comandi è che funzionano in background, ciò significa che si può utilizzare la console di Mulo per altre funzioni mentre si compie la ricerca delle note kad. Questo è stato possibile attraverso l'utilizzo di un thread, che esegue la ricerca utilizzando uno dei due metodi a seconda del comando digitato.

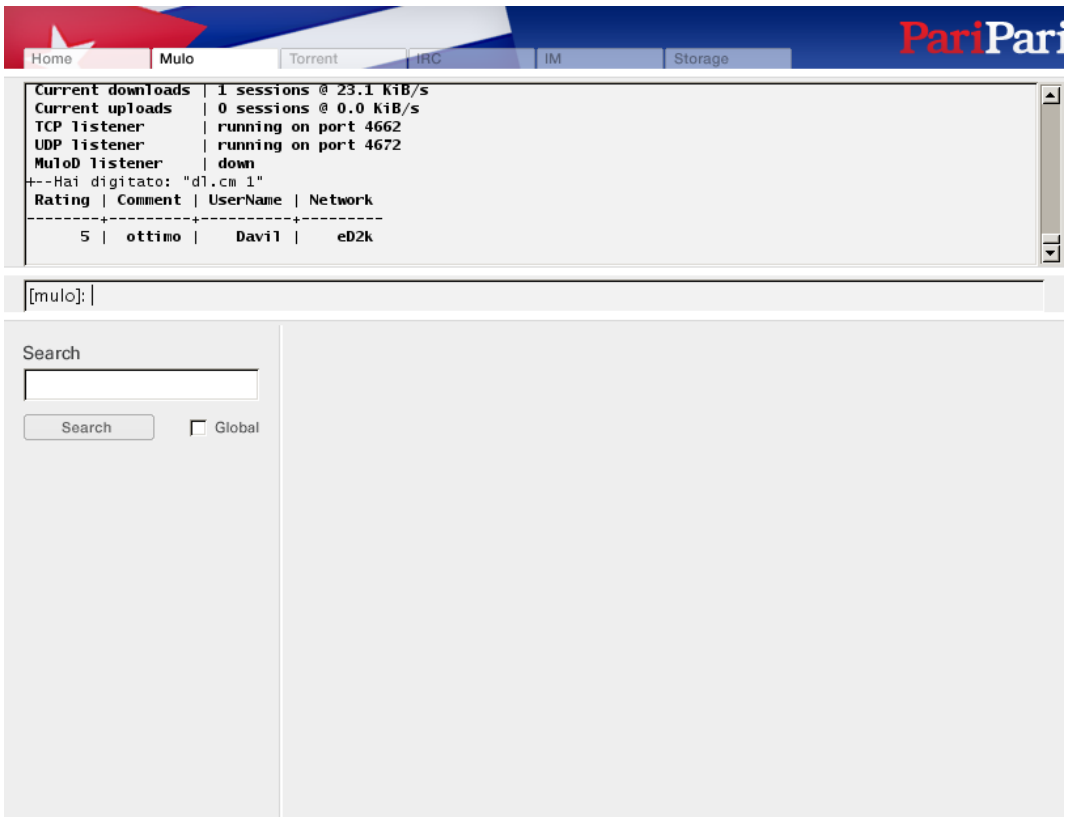


Figura 4.3 Esempio di ricezione di un commento eDonkey.

# Elenco delle figure e delle tabelle

# Elenco dei frammenti di codice riportati

# Bibliografia

[1] Yoram Kulbak e Danny Bickson, *The eMule Protocol Specification*, 2005.

[2] Oliver Heckmann e Axel Bock, *The eDonkey 2000 Protocol*, 2002.

[3] Alexey Klimkin, *Unofficial eDonkey Protocol Specification*, 2003.

[4] Roberto Ampezzan, *PARIMULO 2009*, Padova, 2009.

[5] Renè Brunner, *A performance evaluation of the Kad-protocol*, 2006.

[6] *eMule 0.49c.* http://www.eMule-project.net

[7] *Wikipedia.* http://it.wikipedia.org/wiki/Kademlia

# Ringraziamenti

Per primi devo ringraziare i miei genitori, senza cui non sarei qui oggi, per avermi sempre incoraggiato e per aver sempre creduto in me.
Ringrazio mio fratello perché so che mi vuole bene e so che è pronto ad aiutarmi se ho bisogno di lui.
Ringrazio il mio amico Stefano, “Peli”, per avermi sopportato ed aiutato con i test per il mio lavoro all'interno di PariPari e senza il quale sicuramente non sarei riuscito a finirlo in tempo.
Ringrazio tutti i componenti del plugin Mulo per avermi aiutato dandomi buoni consigli durante questi miei 10 mesi di permanenza all'interno del gruppo e in particolare la Team Leader Martina, sempre disponibile e rapida nelle risposte.
Ringrazio il professor E. Peserico e Paolo Bertasi per avermi dato la possibilità di cimentarmi in un progetto come PariPari.
Ringrazio, infine, i miei compagni di corso Alberto, Marco, Mattia, Manuel e Andrea per avermi saputo tirare su di morale e dato buoni consigli nei momenti più duri, nonché per le numerose “partitine” fatte.