INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

MLdonkey, a Multi-Network Peer-to-Peer File-Sharing Program

Fabrice Le Fessant and Simon Patarin

N° 4797

Avril 2003

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4797--FR+ENG

MLdonkey, a Multi-Network Peer-to-Peer File-Sharing Program*

Fabrice Le Fessant† and Simon Patarin‡

Thème 1 — Réseaux et systèmes
Projet REGAL

Rapport de recherche n° 4797 — Avril 2003 — 30 pages

Abstract: A lot of designers of functional languages have one dream: finding a *killer application*, outside of the world of symbolic programming (compilers, theorem provers, DSLs), that would make their language spread in the open-source community. One year ago, we tackled this problem, and decided to use Objective-Caml to program a network application in the emerging world of peer-to-peer systems. The result of our work, MLdonkey, has superseded our hopes: it is currently the most popular peer-to-peer file-sharing client on the well-known `freshmeat.net` site, with about 10,000 daily users. Moreover, MLdonkey is the only client able to connect to several peer-to-peer networks, to download and share files. It works as a daemon, running unattended on the computer, and can be controlled remotely using three different kind of interfaces. In this paper, we present the lessons we learnt from its design and implementation.

Key-words: peer-to-peer, file sharing, functional programming

* This work was partially supported by the RNTL Project Cyrano funded by the French Ministry of Research.

† fabrice@lefessant.net
‡ simon.patarin@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

MLdonkey, un programme de partage de fichiers pair-à-pair multi-réseaux

Résumé : De nombreux concepteurs de langages fonctionnels ont un rêve: trouver une application, hors du monde de la programmation symbolique (compilateurs, prouveurs de théorèmes, DSLs), qui assurerait l'adoption de leur langage dans la communauté *open-source*. Il y a un an, nous avons attaqué ce problème et décidé d'utiliser Objective-Caml pour programmer une application réseau dans le monde émergent des systèmes pair-à-pair. Le résultat de nos travaux, MLdonkey, a surpassé nos espérances: il est actuellement le client de partage de fichiers le plus populaire sur le site bien connu `freshmeat.net`, et compte près de 10000 utilisateurs quotidiennement. De plus, MLdonkey est le seul client capable de se connecter à plusieurs réseaux pair-à-pair pour télecharger et partager des fichiers. Il fonctionne comme un démon, s'exécutant en arrière-plan sur l'ordinateur et peux être controlé par le biais de différentes interfaces. Dans ce rapport, nous présentons les leçons que nous avons apprises de la conception de ce projet et de sa mise en œuvre.

Mots-clés : pair-à-pair, partage de fichiers, programmation fonctionnelle

1 Introduction

The world of peer-to-peer file-sharing applications is one of the most active domain iqn computer programming nowadays. Following the success of Napster, many new file-sharing networks have appeared in the last three years, such as Kazaa, eDonkey2000 and Overnet, Direct-Connect and Gnutella, the most well-known ones. All of them are based on the new emerging *peer-to-peer technology*: contrary to the standard client-server model, such networks are composed of similar programs, each node behaving more or less both as a client (when it downloads files from other nodes) and as a server (when it uploads files to other nodes, or provides localization services).

Unfortunately, this large number of networks, each one based on a different protocol and targeting different customers, has lead users to the difficult problem of choosing which network to access. Indeed, some of these networks are more efficient for large files — they use concurrent downloads from multiple sources, corruption detection, partial sharing, etc... — while other networks are more efficient for shorter files, or are dedicated to particular types of files. On still other networks, access to files has to be traded on chat forums.

This problem is all the more difficult as these networks are in constant evolution, making the task of understanding the differences between them, already difficult for the experienced user, even more difficult for common users. Moreover, each network requires the use of a different application to join the network: accessing several networks concurrently thus requires to run the different applications in parallel, to repeat the searches on all of them, and forces them to compete for the limited available bandwidth in an inefficient way.

1.1 MLdonkey

Confident in the power of functional languages for writing general purpose applications, we decided to tackle this problem and we implemented a multi-network file-sharing application in Objective-Caml, called MLdonkey. The main noticeable characteristics of MLdonkey are:

- It currently gives access to 6 different file-sharing networks: eDonkey2000 [2], Overnet [3], Gnutella [6] via the LimeWire sub-network,

Network	Topology	Searches	Localization	Downloads	Shar.	Corruption
eDonkey2000	Hybrid	Meta-Data	File-UID	Swarming	Part	MD4 (9 MB)
Overnet	DHT	Keywords	File-UID	Swarming	Part	MD4 (9 MB)
Direct-Connect	Hybrid	Keywords	Search	Linear	Full	
LimeWire	Auto-Hybrid	Meta-Data	File-UID	Swarming	Part	Tiger Hash
Gnutella	Diffusion	Keywords	Search	HTTP	Full	
OpenFT	Auto-Hybrid	Meta-Data	File-UID	Linear	Full	MD5 (file)
Audio-Galaxy	Centralized	Mp3s by WEB	File Prop.	Linear	Full	
SoulSeek	Centralized	Mp3s	Search	Linear	Full	
Open-Napster	Hybrid	Mp3s	Search	Linear	Full	

Figure 1: A summary of the main differences between the networks accessed by MLdonkey.

Direct-Connect [1], SoulSeek [17] and open-Napster [10]. Two other networks, OpenFT and Audio-Galaxy, were also supported in the past.

- In contrast to other file-sharing applications, it is designed to run as a daemon, i.e. unattended by the user.
- It can be controlled locally, or remotely using either a telnet client or a WEB navigator.
- It also provides a Graphical user Interface (GUI) protocol, and can be controlled locally or remotely by several different GUIs. Two such GUIs are provided in the current distribution.
- It runs on many different systems: Linux, Windows native (MinGW), FreeBSD, OpenBSD, Digital Unix, Solaris, Mac OS X.
- It is in daily usage by about 10, 000 users, and it is in the 60 most popular open-source projects on `www.freshmeat.net`.
- It has introduced some of the recent innovative techniques used in peer-to-peer file-sharing applications, such as gossip sources propagation and common chunks copying.
- It is written in about 130, 000 lines of Objective-Caml [7] code, split into about 300 different modules.

The organization of this paper is the following: section 2 briefly presents the different peer-to-peer file-sharing networks and their characteristics; then, the section 3 describes the architecture of MLdonkey, and the section 4 presents our lessons using Objective-Caml for this implementation.

2 Peer-to-Peer Networks

In this section, we present the file-sharing networks that are currently supported by MLdonkey. We give a longer description of eDonkey2000, Overnet and LimeWire/Gnutella, which are the most interesting and powerful of these networks.

2.1 eDonkey2000

The eDonkey2000 network is one of the most advanced file-sharing networks in many respects: search and localization of files are distinct operations; downloads are performed efficiently from multiple sources concurrently for the same file; data corruption is detected by hashing, and finally, files can be partially shared, i.e. before they have been completely downloaded; the protocol itself is encoded in a low-overhead binary format, on full-duplex connections mixing control and data messages. All these features are really important for sharing large files, such as video files or CD ROM images.

2.1.1 Network architecture

EDonkey2000 is a hybrid two-layer network: servers are used to initiate communications between clients. To join the network, a client connects to a server and registers the files that it is sharing (for each file, it provides both meta-data to describe the file content and a unique identifier); once registered, the client can perform either searches – queries on meta-data – or localization queries – queries for a particular file. In the first case, the query language is quite expressive, with usual boolean operators, ranges on scalar values (file size, audio parameters,...), and keyword-based queries; the server replies by providing file descriptions matching the request, as registered by the clients. In the second case, the client requests a particular file identifier, and the server replies by a list of locations (either IP addresses and ports of registered clients for that file, or local identifiers for firewalled clients).

Once a client has decided to download a file, and has received a set of locations for the file from the server, it connects to these locations, and requests information on the file: the location provides information on the file name, on the file partial hashes (used to detect corruption) and on the parts of the file currently available. The client can then decide to join the upload queue (if it is interested in some of the proposed parts), and ask for these parts (using a streaming-like protocol to request multiple ranges in advance).

2.1.2 Hashing and file corruption

To compute the unique identifier of a file, eDonkey2000 clients split the shared files in chunks of 9 MB, and compute for each chunk a MD4 digest (hash). If a file contains only one chunk, its hash is used as the file identifier. Otherwise, chunk hashes are concatenated, and a new MD4 hash is computed, which is used as the identifier of the file.

This approach has several properties: first, files are identified by their content, and not by their name, which allows the system to detect that two different peers share the same file under two different names. Second, using the chunk hashes, it is possible to check for each downloaded chunk if it has been correctly received, to detect corruption and download it again it if needed. Since chunk hashes are used to compute the file identified, it is also possible to check if the chunk hashes provided by a location are not corrupted. Nevertheless, the system can still be attacked by modifying the size of the file, which is not currently taken into account in the hash computation[1]. Finally, once a client has downloaded and verified a chunk, it can register itself as a location for the file, and share all the chunks in the file that have already been downloaded and verified.

[1]A client can, for example, register a file that is one byte shorter than the correct size propagated by other clients, so that a client deciding to download that file will never manage to download the last chunk.

2.2 Overnet

Recently, a lot of research work has been done on the use of *Distributed Hash Tables* (DHT) for localization in peer-to-peer systems [12, 18, 5, 15, 19]. Overnet is the first file-sharing network based on this technology.

Overnet is derived from the eDonkey2000 network, by removing servers and performing searches and localization using the Kademlia DHT protocol [8]: each client, each keyword and each file is associated with a 128-bit identifier; the client identifier is used to select the information which will be stored on it: for example, the files associated with a keyword are stored on the client with the closest identifier (for the XOR metric in Kademlia) to the keyword identifier. Similarly, the addresses of the clients sharing a given file are stored on the client with the closest identifier to the file identifier. Identifiers for keywords and files are computed by using the MD4 digest algorithm.

To perform a search on Overnet, a client hashes the keywords in the query, and search for the corresponding identifiers: the client discovers which other peers in its vicinity have the closest identifiers to the keywords he is searching for, these peers are then asked (using UDP) for the peers which have the closest identifiers, and so on. Finally, the clients with the closest identifiers return either the file descriptions containing the requested keyword, or the locations sharing the requested file. In contrast with eDonkey2000 searches, the results of a search only match one of the keywords of the search, so that the client itself needs to filter the returned results to completely match more complex queries.

2.3 LimeWire/Gnutella

The LimeWire network is a part of the more general Gnutella network. It differs from Gnutella by the fact that simple clients don't diffuse requests between themselves; instead, they are connected in a client-server relation to clients with a better connectivity, called *Ultra-Peers*, and only those ultra-peers behave as standard Gnutella clients. This approach is supposed to solve one of the main problems [13, 14, 16] of Gnutella, i.e. the bandwidth usage, since badly-connected clients have become a bottle-neck for efficient diffusion of requests.

On the ultra-peer, Query-Routing [11] is optimized using Bloom-Filters [4, 9]: for each keyword appearing in the name of a shared file, the client computes an index, using a simple hash function. When indexes have been computed for all the keywords, an array of bytes is sent to the ultra-peer, where each bit number corresponding to a keyword index modulo the size of the array has been set to 1. This table is then used by the ultra-peer to select, for each query, which clients are likely to share the file. However, once a result has been received for a query, the client can only download that file, using HTTP, from the client that sent the corresponding result.

The Gnutella network evolves very fast. Many interesting features, missing in the first versions, are now being introduced: in particular, most of the features available on eDonkey2000 (search by identifiers, multiple-sources downloads (*swarming*), corruption detection (THEX)) are currently available as new extensions to the protocol. The main problem is the large number of extensions introduced by the many client vendors to address these problems.

2.4 Other Networks

Other networks (Direct-Connect, Open-Napster, SoulSeek) are technologically less interesting: they are either hybrid or centralized; searches are more limited in their complexity; files have only one source which is immediately returned in the search result, and downloads are done linearly (from the first byte to the last one).

However, on these networks, forums where users can discuss about the files they are downloading have a major importance. For this reason, MLdonkey also provides chat-capabilities for these networks. These capabilities will be extended in the future to access instant-messaging networks in addition to file-sharing networks. Plugins for IRC, Yahoo, and MSN messenger are already available.

3 Architecture

The architecture of MLdonkey is illustrated on Figure 2: MLdonkey is based on a three-layer architecture: the *network layer* contains the scheduler which manages external connections and timers (for repetitive events and timeouts);

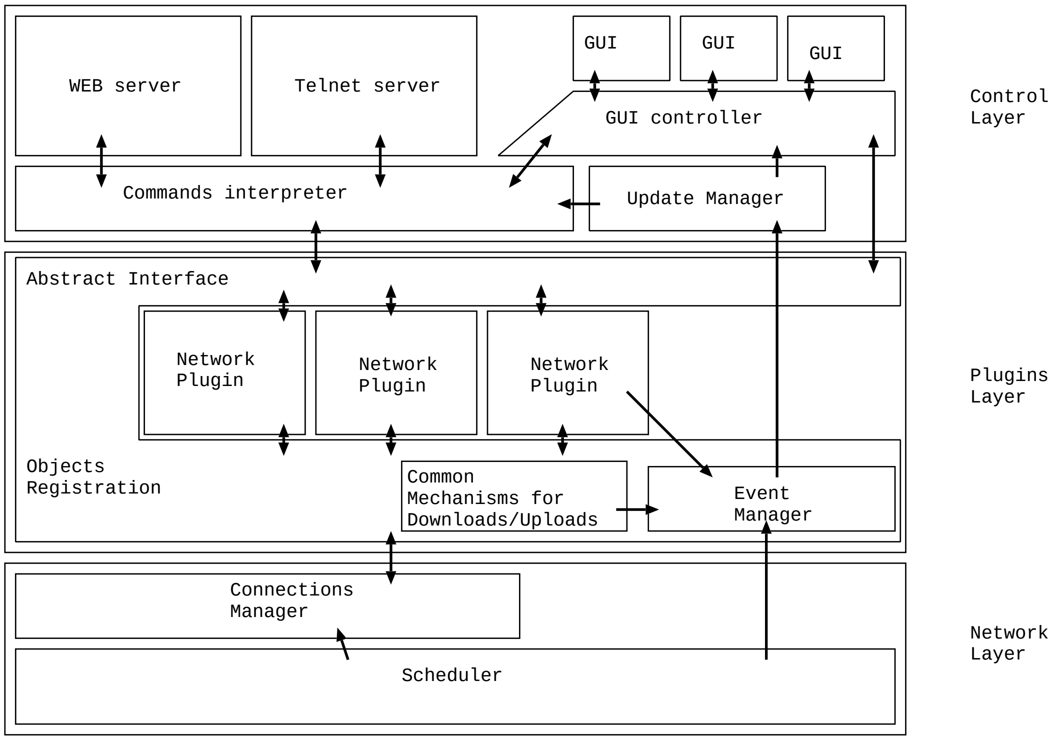

Figure 2: Global Architecture of MLdonkey

the *plugins layer* defines the abstract interface that peer-to-peer network implementations (*plugins*) should provide for the objects they create; finally, the *control layer* defines the different methods that the user can use to control MLdonkey behavior: currently, it contains a WEB server, a telnet server and a server for Graphical User Interfaces (GUI). These three layers are described in the following sections.

3.1 The Network Layer

The network layer is the heart of MLdonkey: since MLdonkey is mostly mono-threaded (see Section 4.3.3), the execution is determined in a scheduler from the state of the network connections: the scheduler is built around a `select` system call, or a `poll` when available, since the latter can handle many more sockets (more than 1024) more efficiently.

The network layer defines four classes of objects to manipulate network sockets: the `BasicSocket.t` object is the common ancestor, from which are derived `TcpBufferedSocket.t` (for TCP data connections), `UdpSocket.t` (to send and receive UDP packets) and `TcpServerSocket.t` (to accept incoming TCP connections).

Depending on the object class, the programmer can define handlers for different events:

`CAN_READ` (`BasicSocket.t`): data can be read on the socket without blocking.

`CAN_WRITE` (`BasicSocket.t`): data can be written on the socket without blocking.

`RTIMEOUT` (`BasicSocket.t`): no data could be read from the socket for a configurable delay.

`WTIMEOUT` (`BasicSocket.t`): no data could be written to the socket for a configurable delay.

`LTIMEOUT` (`BasicSocket.t`): the configurable lifetime of the socket has been reached.

`CLOSED` (`BasicSocket.t`): the socket has been closed.

`READ_DONE` (`TcpBufferedSocket.t` and `UdpSocket.t`): new data has been read from the socket and is available in a buffer.

`REFILL` (`TcpBufferedSocket.t`): the data has been written to the socket, and the buffer can be refilled.

`WRITE_DONE` (`TcpBufferedSocket.t`): data has been written to the socket, and the buffer is empty.

`CONNECTION` (`TcpServerSocket.t`): a connection has been accepted on the socket.

Since MLdonkey is used on limited-bandwidth lines (home connections most of the time), and transfers a lot of data on this link, bandwidth management is an important problem: socket objects (both UDP and TCP) can be associated with a `bandwidth manager`, that fairly schedules the possible read and write operations on buffered sockets depending on the desired bandwidth usage.

3.2 The Plugins Layer

The Plugins layer defines the interface between the *Plugins*, i.e. network implementations, and MLdonkey. This layer is divided in two parts: the *abstract interface*, that the objects created by the plugins should implement, so that they can be controlled by MLdonkey, and the *notification interface*, that is used by the plugins to notify MLdonkey of important changes that should be forwarded to the user when possible.

3.2.1 The Abstract Interface

MLdonkey manipulates generic objects that are supposed to be the common entities of peer-to-peer file-sharing networks: servers, files to download, search results, clients (sources for a download), shared files, chat rooms and users.

Since the implementation of these objects differs completely in every plugin, the Plugins Layer defines a common Abstract Interface, that the plugins have to provide for their implementation of these objects.

Moreover, each object must have a unique identifier, that can be used by the user and the GUI to reference it. For each object created in a class, a *registration* function must be called by the plugin, to create this identifier, and store the object in a weak hash table, so that the object can be found by its identifier until it is collected.

In every interface, a `info` method is defined, that is used by the different user-interfaces to get informations on the object, depending on its type. For example, for a download, the record returned by the `info` method contains the filename, the total size of the file, the current downloaded size, the complete chunks, the chunks available on the network, etc...

Now, we briefly enumerate the methods that should be provided for each class of objects.

The network object is the first object registered by every plugin. It is possible to activate/disactivate a network, perform a search on it, and share particular files.

```
class network =
 method info : unit -> network_info;

 method enable : unit;
 method disable : unit;

 method share : string -> string -> int64 -> unit;

 method begin_search : search -> unit;
 method extend_search : search -> unit;
 method end_search : search -> unit;
 method close_search : search -> unit;
end
```

The server object is used on hybrid and centralized networks. It is possible to connect/disconnect a server, and to request information on the users that are connected to it.

```
class server =
  method info :  GuiTypes.server_info;

  method connect :  unit;
  method disconnect :  unit;

  method query_users :  unit;
  method users :  user list;
  method find_user :  string -> unit;
end
```

The result object is returned by a search on the network. The only interesting operation is to download it.

```
class result =
  method info :  CommonTypes.result_info;
  method download :  unit;
end
```

The file object is used for files being downloaded. It allows to cancel, pause or resume the download.

```
class file =
  method info : GuiTypes.file_info;

  method cancel : unit;
  method pause : unit;
  method resume : unit;
end
```

A shared_file object is created for each file shared on the network, often to get statistics on its popularity.

```
class shared =
  method info : GuiTypes.shared_info;
  method unshare : unit;
end
```

A client object is a location for a file. On most networks, it is possible to browse these clients to see all the files they are sharing, and to chat with them.

```
class client =
  method info : GuiTypes.client_info;

  method send_message : string -> unit;
  method browse_files : unit;
end
```

A room object is a forum, where users of a given server can discuss, often about the files they are sharing.

```
class room =
  method info : GuiTypes.room_info;

  method join : unit;
  method leave : unit;

  method users : user list;
  method send_message : room_message -> unit;
end
```

A user object is a client only known by its name. To interact with a user, it is necessary to register it as a *friend*, to generate a client object.

```
class user =
  method info :  GuiTypes.user_info;
  method add_as_friend : client;
end
```

3.2.2 The Notification Interface

Most Graphical User Interfaces display some pseudo real-time information on the status of the current downloads: how much has been downloaded, what is the current download rate, how many sources are available, etc... Therefore, MLdonkey must provide a way for the plugins to send this information to the GUI. This is done by two components: the *Event Manager* and the *Updates Manager*.

The Event Manager The Event Manager defines the following set of events that can be raised by the plugins. They can be classified in three kinds: *update notifications*, i.e. the state of an object has changed, *link notification*, i.e. a link has been established between two objects (such an event can only appear once), and *error notifications*, for messages that should be presented to the user.

```
type event =          (* update notifications *)
| File_info_event of file
| User_info_event of user
| Client_info_event of client
| Server_info_event of server
| Room_info_event of room
| Result_info_event of result
| Shared_info_event of shared
                      (* link notification *)
| Client_new_file_event of client * result
| File_add_source_event of file * client
| File_del_source_event of file * client
| Server_new_user_event of server * user
| Search_new_result_event of search * result
| Room_add_user_event of room * user
| Room_del_user_event of room * user
| Room_message_event of room * room_message
                       (* error notification *)
| Console_message_event of string
```

The Update Manager Once the events have been stored in the Event Manager, they have to be periodically and efficiently propagated to the GUI.

Error notifications differs from other notifications as they have to be kept even if no GUI is connected, and displayed as soon as a new one connects.

Other events are processed in three steps: first, dependencies are computed, i.e. update notifications on the objects in arguments are inserted before link notifications. In the second step, all the events are copied to multiple queues, one for each connected GUI; moreover, a timestamp is used to check if the state of the object has really changed since the last time the object state was sent to the GUI. Finally, each GUI queue is cleaned, so that duplicate events are removed, i.e. events corresponding to a notification already pending for the GUI. Pending events for a GUI are translated into messages only when they can be immediatly written to the GUI socket. These three steps prevent excessive usage of memory, and allow several GUIs to be connected concurrently with heterogeneous links.

3.3 The Control Layer

MLdonkey is designed to run unattended for days, as a daemon started during the OS boot sequence. As a consequence, MLdonkey does not provide a direct interface to the user, as most other file-sharing softwares do. On the contrary, it implements a more powerful Control Layer, that allows the user to connect to the daemon (either locally or remotely) and control its behavior. Currently, the Control Layer provides three different controllers: a telnet server, a WEB server and a GUI protocol. Moreover, MLdonkey requires user authentication, both by passwords and by network addresses, and has a basic support for multi-user interaction (private searches in particular).

3.3.1 The Telnet Server

It provides the simplest way to control MLdonkey: using the simple `telnet` application, the user can send commands to MLdonkey, or ask its current status. Commands in this interface can be classified as follows:

Status commands : to display the current connections to servers, to sources, and the status of downloads for example.

Search commands : to search for files on all networks, display the results and trigger downloads.

Control commands : to pause, resume or cancel downloads, or to require new connections to servers or sources.

Configuration commands : to dynamically display and change the parameters, and save the current configuration.

Moreover, all these commands can also be called from all the other controllers (the GUI provides a `console` panel, and the commands can output HTML code on demand).

3.3.2 The WEB Server

The built-in WEB server of MLdonkey is a user-friendly interface, that can be used from any WEB navigator, and offers the same functionalities as the telnet server.

3.3.3 The GUI Server

The GUI server allows several external GUI to connect and interact concurrently with MLdonkey, using a specific binary protocol. There are several good reasons to use this protocol instead of the telnet for this purpose:

- The telnet protocol is difficult to parse and the format of the output can change very often between MLdonkey versions.
- A binary protocol is simpler to parse, and uses less bandwidth than more verbose protocols, such as XML.
- A versioning system is included in the protocol so that any MLdonkey daemon version can be controlled by any GUI version, at the cost of some features disabled. Each connection between a GUI and the daemon starts by a negotiation of which version of the protocol should be used: MLdonkey itself is able to use all the protocol versions to adapt to any older GUI.

```
let limewire_ini = create_options_file (
  Filename.concat file_basedir  "limewire.ini")

let max_ultrapeers = define_option limewire_ini ["max_ultrapeers"]
    "Maximal␣number␣of␣ultrapeers␣connected" int_option 5

let _ = option_hook max_ultrapeers
    (fun _ -> if !!max_ultrapeers > 30 then max_ultrapeers =:= 30)
```

Figure 3: How options are defined, used, and modifications verified.

```
type options_file             (* Operations on configuration files *)
val load : options_file -> unit
val save_with_help : options_file -> unit

type 'a option_class                          (* Some options classes *)
val string_option : string option_class
val int_option : int option_class
val bool_option : bool option_class
val float_option : float option_class
val path_option : string list option_class
val list_option : 'a option_class -> 'a list option_class

type 'a option_record                        (* Operations on options *)
val define_option : options_file ->
  string list ->  string -> 'a option_class -> 'a -> 'a option_record
val option_hook : 'a option_record -> (unit -> unit) -> unit
val ( !! ) : 'a option_record -> 'a
val ( =:= ) : 'a option_record -> 'a -> unit
```

Figure 4: The basic operations on options. The list argument in `define_option` allows to define namespaces in the configuration file.

4 Functional Programming

MLdonkey has been developed in the Objective-Camllanguage. In this section, we address the interesting aspects of using a functional language to develop a peer-to-peer file-sharing program.

```
module PeerOption = struct
 let value_to_peer v =  match v with
   SmallList [MD4; ip; port] (* ("A23...7F","128.93.2.100",4662) *)
 | List [MD4; ip; port]      (* ["A23...7F";"128.93.2.100";4662] *)
 | Module ["MD4", MD4; "ip", ip; "port", port]
     (* { MD4 ="A23...7F"; ip = "128.93.2.100"; port = 4662; } *)
   -> {
            peer_MD4 = MD4.value_to_hash MD4;
            peer_ip = Ip.value_to_ip ip;
            peer_port = value_to_int port;
      }
 | _ -> failwith "Unexpected␣format␣for␣peer␣option"

    let peer_to_value p =
        SmallList [
          MD4.hash_to_value p.peer_MD4;
          Ip.ip_to_value p.peer_ip;
          int_to_value p.peer_port;
        ]

    let t = define_option_class "Peer" value_to_peer peer_to_value
end
```

Figure 5: The `PeerOption` module defines a new `option_class` to save peer addresses (IP address and port).

```
   (* Set to true if you also want mldonkey to run as an eDonkey server *)
 enable_server = false

   (* list of IP addresses, allowed to remotely control MLdonkey,
      wildcard=255 ie: use 192.168.0.255 for 192.168.0.*  *)
 allowed_ips = [ "127.0.0.1";  "192.168.0.255";]
```

Figure 6: The configuration files created by the `Options` module are written in human-readable form and can be directly edited by the user.

4.1 Advantages

The use of a functional language has many advantages over a traditional language such as C. Since most of them are quite obvious, we only focus on the major ones, which greatly helped us in the development.

```
type t                               (* Basic operations on files *)
val create : string -> Unix.open_flag list -> int -> t
val getsize : string -> int64                   (* current size *)
val mtime : string -> float               (* modification time *)
val filename : t -> string
val set_filename : t -> string -> unit             (* renaming *)
val ftruncate : t -> int64 -> unit    (* setting current size *)

val max_supported_fds : int      (* filedescr cache management *)
val max_cache_size : int ref
val close_all : unit -> unit

val flush : unit -> unit         (* write buffering management *)
val flush_fd : t -> unit
val buffered_write : t -> int64 -> string -> int -> int -> unit
val write : t -> int64 -> string -> int -> int -> unit
val max_buffered : int64 ref

                            (* sparse-files operations *)
val fd_of_chunk : t -> int64 -> int64 -> (Unix.file_descr * int64)
val read : t -> int64 -> string -> int -> int -> unit
val allocate_chunk :  t -> int64 -> int64 -> unit
val copy_chunk : t -> t -> int64 -> int64 -> int64 -> unit
```

Figure 7: The Virtual File-System Interface

4.1.1 Code Consistency

MLdonkey is a rather large implementation: it contains 100 000 lines of Objective-Caml, split in 290 modules (just a bit smaller than the 140 000 lines of the complete Objective-Caml distribution).

In such a large project, each modification of the basic modules can break many parts of the system. Finding all the occurrences where the code should be modified is a painful task with common languages. Fortunately, the strong type system of Objective-Caml, and the consistency checks between the interfaces on which compiled units depend, allowed us to overcome this problem easily.

4.1.2 Stability

In contrast with most peer-to-peer file-sharing applications, MLdonkey is a daemon, without a direct user interface: it is supposed to run unattended, for really long periods, as any file-server program. Moreover, it is not a sleeping daemon, executing a task only from time to time as most Unix daemons: MLdonkey is always working, either looking for servers to connect to, for sources to download from, for clients to upload files to, or hashing files on the local disk. Finally, the lifetimes of MLdonkey objects are hard to predict, since they mostly depend on the other peers on the network.

All these features make an application very sensible to memory management problems: whereas existing memory leaks may not appear in a short-running application, they become more likely to appear in our case. Fortunately, Objective-Caml mostly solves this problem for us, by providing:

An Efficient Garbage Collector: Thanks to it, we don't have to care about objects lifetimes. Moreover, the two-generation collector performs very efficiently, both for short-life objects as for long-life objects.

Weak References: most protocols use simple identifiers to reference objects. Consequently, MLdonkey performs many look-up operations, and must maintain many different tables to associate objects with their identifiers. Thus, these tables are global roots and prevent automatic garbage-collection for these objects, leading us to manually free the objects, by tracking the tables where they are registered and removing them, which is error-prone and may cause memory leaks.

In MLdonkey, we use the *weak references* of Objective-Caml and, in particular, the weak hash tables to avoid this problem: references found in these tables are not used as garbage-collection roots. For every object, only one table keeps a strong reference to the object (most of the time, in the plugin that created the object), while all other references are weak.

Compaction: One of the major problems with distributed applications using garbage collection is the fragmentation of the heap: indeed, contrary to most applications that use small fixed size objects, distributed applications have to manage input/output buffers with various sizes, and various lifetimes.

The Objective-Caml runtime solves this problem by compacting the heap: it continuously computes an heuristics of the heap fragmentation, and automatically triggers a complete compaction when the fragmentation is above a user-customizable threshold.

Thanks to all these mechanisms, MLdonkey can run for weeks, if there were not the need to update it for the benefit of new features. Its memory footprint is still a bit large, compared to other daemon: from 9 MB to 70 MB, depending mainly on the popularity of the files being downloaded or shared.

4.1.3 Portability

A common usage when writing distributed applications using binary protocols, and low-level languages such as C, is to optimize the encoding and decoding of messages, by casting them into C structures, and then use platform-specific swapping macros (such as `ntohl`) between the network format and the local representation of valUes.

The use of a high-level language such as Objective-Caml prevents us from such optimizations: we had to implement all the functions to encode and decode simple values from message strings. Unexpectedly for a strongly network-oriented application, the lack of such optimizations has proved to have a minor impact on MLdonkey overall performances.

An interesting remark is that users have shown to accept the higher cpu usage and memory footprint of MLdonkey compared to other daemons running on the same computer, as it is often seen as a useful task, compared to other ones, since the benefits of its execution can be seen.

However, a nice advantage of this (forced) approach is that MLdonkey runs without modifications on most platforms: in particular, it handles correctly the differences in byte ordering between x86 and Power-PC platforms, where it is one of the only available peer-to-peer open-source softwares.

Thanks to the layer added by the virtual machine upon the Operating System, MLdonkey has been ported to many systems: Linux, Windows Cygwin, Windows native (MinGW), FreeBSD, OpenBSD, Mac OS X, Solaris, Compaq Unix, etc. and all the processors where Objective-Caml is available.

4.1.4 Asynchronous IO

There are mainly two approaches in distributed applications: asynchronous IO, using `select` and `poll` system calls, or multi-threaded IO. The latter approach is limited for two reasons:

1. Threads are not available on all platforms, and therefore raise a portability issues.

2. The number of threads that can run concurrently on the same computer is often limited, mainly by the load they put on the system scheduler. Unfortunately, MLdonkey uses many concurrent connections (often more than one hundred), and creating one thread per connection is not acceptable.

As a consequence, asynchronous IO had to be used: MLdonkey core engine is a scheduler, where continuations are associated with connections waiting on network events. And fortunately, functional languages are an ideal environment for continuation passing style programming.

4.2 Drawbacks

Using a functional language such as Objective-Caml has also some drawbacks. Here, we focus on the two main problems we have encountered so far.

4.2.1 Lack of Contributors

The first drawback of using a functional language is the lack of developers with good skills in these languages. This problem is particularly important for an open-source project: contrary to a commercial application, where a developer can be hired with the desired skills, or where acquiring these skills is part of the job, an open-source project can only rely on the subset of its users with these particular skills.

Unfortunately, functional languages skills are still seldom in the open-source community, where C, perl, php are mainly used, and particularly in its network-related subset, where C, C++ and sometimes Java are preferred.

This has been a real problem for the development of MLdonkey, since, 1 year after its first release, only 3 external developers have provided major contributions to the project. However, we are a bit more confident in the future, since the popularity success of MLdonkey (now included in the 60 most popular open-source projects on `www.freshmeat.net`) and the good ratings obtained by Objective-Caml, both in the past years ICFP programming contests and in some languages comparison experiments, are incitating open-source developers to learn functional languages.

4.2.2 Lack of Debugging Tools

Another important issue that appeared during the development of MLdonkey is the lack of two important debugging tools in the Objective-Caml distribution.

A native debugger Native debuggers are mainly useful in two cases: (1) when a memory protection fault occurs, and (2) to trace the execution of the program, using breakpoints.

Thanks to Objective-Caml garbage collector, (1) is very limited: until now, memory faults have only appeared in MLdonkey, due to small bugs in the few C functions used to compute digests of files (MD4 or SHA1), to hardware problems causing kernel faults, and to changes in the internal representation of registers in the trap handlers of Mac OS X, used by Objective-Caml exception handling code.

However, (2) would be much more interesting for us. In particular, the impossibility of being able to set a breakpoint in an Objective-Caml function, and of conveniently inspecting the data structures at that breakpoint, lead us to painful modify, retry, and wait for the bug sequences, and to examine log files to understand what was the state when the bug occurred.

A memory debugger An important problem with garbage-collected applications is the difficulty to understand where memory is used, and when memory can be collected. We have had, and still have bug reports complaining about a memory footprint of more than 100 megabytes, used by MLdonkey without reasons in very particular cases.

The memory debugger could have solved such problems by providing two different useful informations on the program:

- Amount of memory allocated for each particular data type.
- Amount of memory kept live by each global root.

We hope that such a debugger will be provided in the future for Objective-Caml, and we are currently implementing a module for heap inspection.

4.3 Programming Pearls

We now focus on some interesting parts of the implementation of MLdonkey in Objective-Caml.

4.3.1 Complex options

One of the interesting parts of MLdonkey is the way options are manipulated. The Figures 3 and 4 give an example of use, and an overview of the `Options` module interface.

The `Options` module allows the programmer to manipulate options just as references in Objective-Caml, in a completely type-safe way: the `=:=` operator is used to modify them, whereas the `!!` operator is used to extract their value. Options are defined using the `define_option` function, that defines the option name, its default value, a comment to help the user, its type and the configuration file in which it is saved.

The type of the option is defined by providing an `'a option_class` value in argument, that forces the option type to be inferred with the correct type. Different option classes are provided for the most common types, and a lower-level interface provides functions to define new classes. An example of an option class for the a `peer` object is given in Figure 5: using this option class, lists of peers can be easily saved in the configuration files to be used at next restart.

Moreover, the `Options` module reads and writes configuration files in human-readable format: this allows the user to read and modify these files using a simple text editor, whereas the programmer is not concerned with the representation of the options in the file, since it directly manipulates the options as easily as references.

Interestingly, it can seem useless to implement complicated algorithms in this module, as the options are often simple values (integer,strings) and are only saved at long intervals (15 minutes, or the end of the execution). However, the `Options` module has proved to be very convenient to implement persistence across executions of many other data structures, such as list of servers, list of sources for each file, etc...It has been observed that such lists could contain more than 100000 elements in normal execution.

As a consequence, it has been necessary to implement some optimizations:

- To avoid stack overflow errors, all used functions are tail-recursive (for example, the non tail-recursive `List.map` function has been completely removed).
- The basic saving mechanism first translates option values into an intermediate representation, which is then saved to a file. This mechanism can unnecessarily allocate hundreds of megabytes for the intermediate representation of very long lists. Hence, an abstract 'map' value has been introduced in the intermediate representation, that allows to delay the computation of the intermediate representation of each element of the list until it is really used, and so that it can be immediately collected.
- To decrease the size of configuration files containing long lists of files, sharing of values has been introduced: some values are only printed once in the configuration file, with an alias attached to them, that can be used to name the value multiple times in the file.

4.3.2 The Virtual File-System

The main job of MLdonkey is to write the data received from the network to files on the local disk. Data is written linearly to the file, except when several sources are available, in which case the data can be written at different positions in the file. The simplest scheme to achieve this task is:

1. Create a file of the size of the complete desired file
2. Each time some data is received,
   (a) Open the file.
   (b) Move to the position where the data should be written.
   (c) Write the data to the disk.
   (d) Close the file.

A first remark is that, when the (c) operation fails to complete, it is most of the time related to a lack of space on the disk. In such a situation, an exception is raised, and MLdonkey immediately pauses the download; the user must then free some space on the disk and resume the paused downloads.

Now, we can describe four improvements on this basic scheme.

File Descriptor Cache Since opening and closing a file are expensive system calls, MLdonkey keeps a cache of opened files, limited in size by the maximal number of file descriptors on the system, that have to be shared with network connections.

64-bit Operations the first versions of MLdonkey were able to handle files of $2^{31}-1$ bytes (the maximal positive value of the Objective-Caml `int32` type). As bigger files have appeared on some of the supported network, we were forced to use the `int64` type. Fortunately, Objective-Caml already provides most Unix system calls supporting values of this range, and only a few of them had to be added.

Read/Write Buffering MLdonkey tries to share the bandwidth available fairly between the different connections. As a consequence, most read and write operations are done on small buffers, and it induces an unnecessary heavy load and dangerous on the disk. We recently implemented buffering of IO operations to solve this problem. Moreover, buffering improves the grouping of file blocks on disk, so that files are less fragmented.

Simulating Sparse Files MLdonkey contrasts with former file-sharing programs, as it is not designed for interactive downloads. Indeed, downloads using MLdonkey can last for weeks, since every file can reach hundreds of megabytes, and sources are not always available. As a consequence, it is not seldom to have tens of concurrent downloads, and the creation of all these files would lead to tens of gigabytes, filled with zeros, reserved on the disk for incoming data. Fortunately, modern file-systems don't allocate this space, they create *sparse-files*, where all zero-filled blocks of the file point to a special block. And unfortunately, MLdonkey is also used on older file-systems (the file-system of Mac OS X and the FAT file-system of Windows 98 for example) that do not support such files, so that the Virtual File-System has to provide emulation of sparse-files. This is done by storing received data in smaller separate files, merging them when possible, and providing functions that hide the real implementation of files. This is also the reason why buffering is done in the Virtual File-System and not using simpler buffered C functions, such as `fprintf`.

4.3.3 Asynchronous Operations

As explained earlier, MLdonkey is mainly mono-threaded, for portability and for simplicity of design (no concurrent accesses to data structures). This design choice lead us to split long computations into shorter ones, that are scheduled using short timers.

However, this mechanism is not always sufficient: computing the MD4 digest of a 9 megabytes chunk on a modern computer can take between 1 second (which is acceptable), if the file is in the memory cache, and 10 seconds (which is not acceptable, since in the order of magnitude of the timeouts on connections), if the file is on the disk, and fragmented. Converting a computer name to an IP address (DNS resolution) in a synchronous call[2] can also take about 30 seconds in some cases, which is unacceptable.

Consequently, we recently decided to use POSIX threads for these two tasks. The two corresponding functions takes a continuation argument, so that they can be implemented using timers, when POSIX threads are not available. Objective-Caml provides support for POSIX threads, but its garbage-collector is not re-entrant, so that several threads cannot allocate Objective-Caml data concurrently. Fortunately, these two tasks were already completely implemented in C (digest algorithms in C are available in RFC for MD4 (1320), SHA1 (3174), ...).

For both tasks, the mechanism is the same, only one thread is used for all computations (one for hashing, reniced to a lower priority, and one for DNS resolution): operations to be computed are stored in a FIFO; every tenth of second, a timer triggers an Objective-Caml function that checks if the last job is finished (using a native C function), calls the continuation if needed, checks if a job is waiting in the FIFO, and reactivate the thread in that case (using another native C function, and a *condition* for inter-threads communication).

5 Conclusion

In this paper, we have briefly described our experience in the design and the implementation of a peer-to-peer file-sharing application, MLdonkey, in the

[2]We had decided not to depend on third-party libraries, such as the Asynchronous DNS library.

Objective-Caml functional language. First, we have presented some of the networks to which MLdonkey can connect, and try to highlight their main characteristics. We have then described the general architecture of MLdonkey, and its three layers: the network layer, the plugins layer and the control layer. Finally, we also enumerated some of the advantages and drawbacks of using a functional language in this implementation. In particular, we think there is a real need for advanced debugging tools for functional languages, for example to profile the memory usage of garbage-collected applications. There is also a a substantial effort to do in developing general-audience applications (in the instant-messaging, multimedia, and system management domains for example) to attract more open-source programmers towards functional languages.

6 Acknowledgements

We would like to thanks all the contributors to MLdonkey, and in particular our colleague Maxence Guesdon who provided the current GUI of the distribution. MLdonkey was developed within the SOR action, at INRIA Rocquencourt, as an application of the Cyrano RNTL project.

References

[1] Direct-connect. `http://www.neo-modus.com/`.

[2] edonkey2000. `http://www.edonkey2000.com/`.

[3] Overnet. `http://www.overnet.com/`.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *8th IEEE Workshop on Hot Topics in Operating Systems*, germany, 2001.

[6] Gnutella. http://www.gnutella.com.

[7] X. Leroy, D. Doligez, and J. Vouillon. The objective-caml distribution.

[8] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *1st Internaional Workshop on Peer-to-Peer Systems*, MIT, 2002.

[9] Mitzenmacher. Compressed bloom filters. In *PODC: 20th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2001.

[10] Open-Napster. http://opennap.savannah.org.

[11] Query-Routing 0.1. http://www.limewire.com/developer/query_routing/keyword%20routing.htm.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[13] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *International Conference on Peer-to-peer Computing*, 2001.

[14] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), Jan./Feb. 2002.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[16] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, 2002.

[17] SoulSeek. http://www.soulseek.org.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[19] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, 2001.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399