UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Uso de clientes híbridos Kad-BitTorrent para compartir contenidos.

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Damián Alberto Vicino

Director: Dr. Claudio Righetti

Codirector: Dra. Isabelle Chrisment

Asesor externo: Ing. Juan Pablo Timpanaro

Jurado: Ing. Alejandro Furfaro, Lic. Rodolfo Baader

Buenos Aires, 2012

USO DE CLIENTES HÍBRIDOS KAD-BITTORRENT PARA COMPARTIR CONTENIDOS.

La aplicación para compartir archivos en redes peer to peer BitTorrent tiene su foco en optimizar la propagación de contenido, esto la hace muy atractiva respecto de sus competidores, para evitar el uso de recursos centralizados al momento de transferir archivos, BitTorrent utiliza Tablas de Hash Distribuidas basadas en Kademlia que permiten encontrar fuentes del contenido sabiendo el hash de su metadata. Por otro lado, la red eMule que también provee transferencia de archivos sobre redes peer to peer, utilizando un protocolo de transferencia de archivos conocido como eDonkey 2000 (ED2K) basado en colas de prioridad, posee una solida implementación de Kademlia llamada Kad. Esta implementación provee un servicio de doble indice, con el cual no solo indexa fuentes, ademas es usada para indexar y buscar contenido en base a claves. Nuestro trabajo, estudia la posibilidad de implementar un cliente híbrido compatible con ambas redes peer to peer, el cual pueda explotar las ventajas de indexación de Kad, al mismo tiempo que la velocidad de propagación de contenido de BitTorrent. Para ello desarrollamos un prototipo, el cual soporta indexación de contenido usando Kad, transferencia usando BitTorrent y es retro-compatible con los clientes que implementan estas tecnologías actualmente. Usando este prototipo, medimos tiempos y velocidades de propagación de contenido en clusters de nodos mixtoss para concluir que el desarrollo es factible y beneficia ampliamente a los usuarios. Finalmente describimos los cambios necesarios para que el prototipo desarrollado pueda ser distribuido a usuarios finales.

Keywords: Kad, BitTorrent, Distributed Hash Table (DHT), Peer to peer (P2P) Architecture, Performance, Security.

USING KAD-BITTORRENT HYBRID CLIENTS TO SHARE CONTENTS.

BitTorrent is a fast, popular, P2P file-sharing application focused on fast propagation of content. Its trackerless approach uses a DHT based on Kademlia to search for sources when the hash of the metadata of the content to transfer is known. On the other hand, the eMule network use the old ED2K protocol for file-sharing including a system of priorized queues, but indexation is done through a solid Kademlia based DHT, named Kad. The Kad DHT stands for a search engine, wich provides an extra level to map keywords to file identifiers. We propose an hybrid approach, compatible with both P2P file-sharing networks, which has the Kad advantages on indexation and the BitTorrent throughput for transfer while maintaining backward compatibility with both of these networks. To validate our proposal we developed a prototype which supports content indexation provided by the Kad network and is able to transfer files using the BitTorrent protocol without losing retro-compatibility. Using this prototype, we measured the propagation of new content in clusters of aMule clients, BitTorrent clients, hybrid clients, and a mix of them. Comparing the propagation velocity of content propagation in each scenario, we conclude the development of a full implementation would significantly benefit the users. Finally we describe the evolution the prototype needs before being distributed as production quality software.

Keywords: Kad, BitTorrent, DHT, P2P Architecture, Performance, Security.

AGRADECIMIENTOS

Al Dr. Claudio Righetti, por motivarme a investigar, a iniciar una carrera científica y apoyarme a lo largo de este proceso.

A la Dra. Isabelle Chrisment, quien fue mi directora y guiá en la pasantía que dio origen a este trabajo, y luego siguió de cerca mis avances con todos los problemas que la distancia acarrea.

Al Ing. Juan Pablo Timpanaro, que tuve el agrado de conocer en mi estadia en Francia y que desde entonces se convirtio en un gran amigo y apoyo a la distancia.

A los miembros del jurado, Ing. Alejandro Furfaro y Lic. Rodolfo Baader que revisaron detalladamente mi trabajo, aportando comentarios y sugerencias para lograr el resultado actual.

A mi esposa Danila, mi hermana Nidia, y mi gran amigo Matias Celani por acompañarme en todos mis proyectos, empujar para que las cosas salgan y ayudarme siempre con cualquier cosa que necesite.

A mi familia, que estuvo conmigo siempre.

A mis compañeros de estudio, en especial a Luis Agustín Nieto, Matias Gonzalez, Wenceslao Zuloaga, Cynthia Bonomi, Gabriel Barbuto, sin los cuales no hubiera llegado nunca hasta esta instancia.

A los docentes del DC, que me transmitieron sus conocimientos y experiencias a lo largo de todos estos años.

Al BSD User Group del DC y la comunidad BSD Argentina por las charlas y eventos compartidos.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Since the beginning of the Internet one of its most popular uses has been file-sharing between users.

In early days, a Client/Server (C/S) model was adequate to handle file-sharing among users (i.e. a FTP server could contain all the files and users could retrieve the files as needed). After the big worldwide expansion of the Internet the resources required to provide central FTP servers became too expensive and a new set of technologies for sharing files needed to be developed. An alternative approach was found in P2P models. Within these models, users transfer files between themselves, avoiding a central server and therefore scaling more quickly and efficiently.

1.1 Content-sharing distribution models

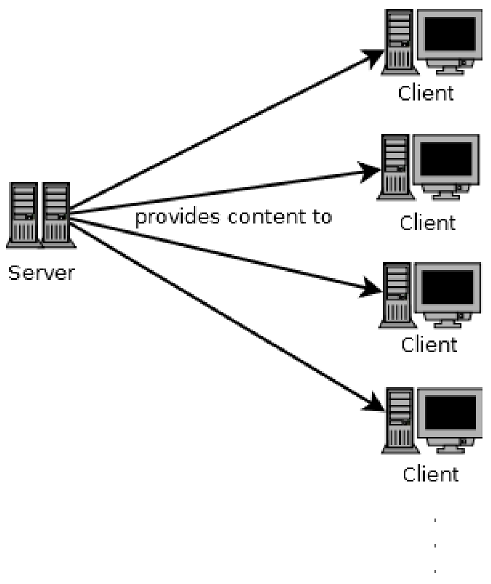


Fig. 1.1: Client-server content distribution

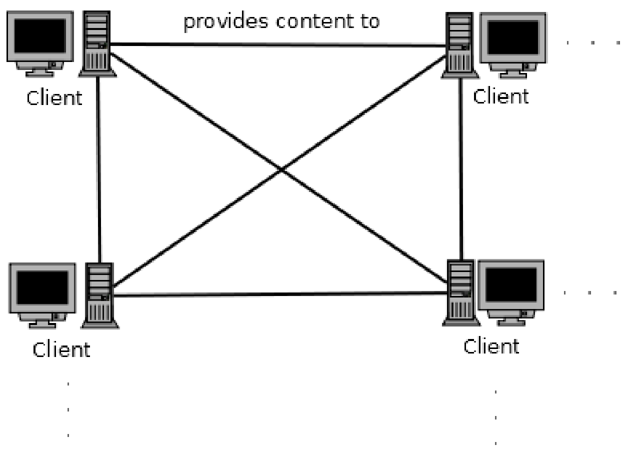


Fig. 1.2: Peer-to-peer content distribution

The two most used models for content-distribution using Internet are the C/S and the P2P models. The C/S model has scalability issues because it centralized resources nature. On the other hand P2P has more difficulties for retrieve content and sources given its decentralized implementation.

Figure 1.1 illustrates the C/S model. The producer uploads the content to a central server and links it from a website providing this content. The consumer searches this content in the website and downloads it directly from the server.

In the P2P model, as depicted by figure 1.2, transfers are done between users without central server, even, in some cases for indexation. Let's consider the following scenario: a user shares a tutorial BitTorrent video using the eMule P2P network; the user selects content to be publish in the network (for which he/she will be the only available source). Then the eMule client creates a set of keywords (*BitTorrent* and *Tutorial* for our example), which along with the content itself are indexed in a distributed hash table named Kad in the following manner: Keywords –>Content –>Source. A second user can search in the Kad distributed hash table for the content through keywords that will provide him a content ID. Once the client obtains the content ID, it searches for sources providing the content and starts the transfer from the found sources.

1.2 P2P Networks

We adopt the definitions of P2P and Pure-P2P from [11]:

> *A distributed network architecture may be called a Peer-to-Peer (P-to-P, P2P,...) network if the participants share a part of their own hardware resources (processing power, storage capacity, network link capacity, printers,...). These shared resources are necessary to provide the service and content offered by the network (e.g. file sharing or shared workspaces for collaboration).*
>
> *The participants of such a network are thus resource (service and content) providers as well as resource (service and content) requesters (Servent-concept).*
>
> *A distributed network architecture has to be classified as a Pure-Peer-to-Peer network, if it is firstly a Peer-to-Peer network (...) and secondly if any single, arbitrary chosen Terminal Entity can be removed from the network without having the network suffering any loss of network service.*

P2P turned popular when traffic bottlenecks generated by using a central point for data transfer (the server) became notoriously expensive, specially for free of charge content distribution where the provider should pay large bandwidth charges.

Three different approaches were used in P2P network designs: pure P2P[1] where users never need to interact with a centralized resource (i.e. gnutella, freenet), statically centralized hybrid P2P where users transfer the files between each other but coordination between peers is decided by a central known resource (i.e. eDonkey2000) and dynamically centralized P2P where a small subset of selected peers work as coordinators, usually called super-nodes (i.e. Kazaa).

We are particularly interested in the pure P2P design which doesn't need anything but users joining the network to be in service.

1.3 Content Indexation in P2P

Due to the distributed nature of P2P systems, there is challenge on implementing a distributed indexation mechanism so all participants could search for content in the network. One effective solution is to use a distributed data structure, such as a DHT[16]. DHT provides the same abstract interface than normal hash tables, allowing the association between keys and values, which can be used to provide a search-by-key service. Also, the DHT implementations should cover the following properties: decentralization, fault tolerance and be high scalability.

Within a pure P2P approach, we need to consider two associations: between keywords and content, and between content and sources. A user searches for a given content through keywords, and once the content is found, the user will search for sources providing that content so as to download it. A two-level indexation DHT is adequate to fulfill this double indexation.

Kademlia is one of the most popular DHT implementations for P2P content-sharing. Both, eMule and BitTorrent, follow the Kademlia specification for their DHT implementations.

1.4 eMule vs BitTorrent

The most popular content-sharing technologies at the moment are the eMule and BitTorrent networks.

[1] Also known as decentralized file sharing network.

On the one hand, eMule uses the ed2K protocol for file transfers and a Kademlia-based implementation named Kad for content indexing. On the other hand, the BitTorrent protocol applies a tit-for-tat based algorithm for file transfers and 2 Kademlia-based DHTs for its distributed trackers, along with a central tracker option.

In recent work, Timpanaro et al.[13] compares the file transfer protocols of eMule and BitTorrent evaluating strengths and weaknesses of them. They show that BitTorrent outperforms the ED2K protocol for file transfers, while the indexations mechanism of eMule has several benefits over those used in the BitTorrent network in terms of security.

1.5 Objectives

Our main objective is to evaluate our proposal of an hybrid client which merges together the benefits of Kad for indexation mechanism along with BitTorrent transfer protocol to speed up the transfers.

To validate our approach and obtain the non trivial to see requirements, we developed a functional prototype. This prototype (named hMule) has all functionalities needed to join Kad network, transfer content using BitTorrent, detect when to call each protocol and fairly uses both networks to avoid been banned in any of the them.

1.6 Document Structure

The report is organized as follows. Chapter 2 presents how file-sharing service is provided using current implementations of eMule and BitTorrent compatible applications. Chapter 3 discuss hMule development. We start with architecture decisions, after that, we describe the changes and extensions to the protocols and logging introduced. Chapter 4 shows the experimentation design, the results obtained and its analysis. Chapter 5 exposes the conclusions. Chapter 6 provides some ideas for future work in the hMule project. Finally, the appendices include an installation guide, a user guide, a troubleshooting for known issues, documentation of the most relevant new classes implemented and some scripts developed in the experimentation phase.

2. RELATED WORK

2.1 Introduction

A file-sharing service is composed of 3 steps: (1) a user searches for files that suits his/her needs using keywords; (2) he/she selects which files from the search results he/she would like to obtain and asks for the list of sources; (3) at last he/she asks to the sources for the content of the files.

In the client/server approach, the first step is accomplished using a site like `www.yahoo.com`, `www.google.com`, `www.download.com`, etc. or links provided by email, IRC or another channel. The second step is provided by the server which has the content itself or a secondary server providing a portal with the index of the content and resources location. Finally, on the third step, the transfer is granted by ftp, http or a similar protocol.

There are different approaches to P2P with slight variants. We will focus on two of the available P2P technologies, eMule and BitTorrent.

For the first step, the eMule Kad network has a first level index that maps keywords to file identifiers while BitTorrent uses web sites that publish info-files containing the metadata needed for the second step (pretty much like the C/S approach). The second step for eMule is handled by a second level index in Kad while BitTorrent uses an implementation of DHT that maps file metadata to sources. Both networks can also use centralized coordination servers[1] in some cases. This behavior is implemented for backward compatibility reasons, it allows interaction with clients which were developed before the introduction of DHTs. In the last step, each client uses it own protocol for transfer.

2.2 Distributed Hash Table (DHT)

The DHT[16][5] is a distributed data structure which virtually acts as a normal hash table, but the implementation is distributed along several nodes. To avoid centralization of resources, each key-value pairs is stored among a set of the participants of the network in a way so all peers handle similar load of data and transfer.

A DHT needs to have the following properties:

- Decentralization: Every peer has the same duties
- Fault tolerance: Connection and disconnection of peers do not affect service uptime
- Scalability: The system supports a large number of peers

Several architectures were proposed, such as Chord[12], Pastry[10], Tapestry[17], Kademlia[7]. They assign a unique identifier to each key and each value, and a function to evaluate the distance between two identifiers. When searching for a specific key, the closest peers (the so called *replica set*) to a key are queried. The same approach is used when storing a new value: it is stored in the closest peers, so as to assure fault tolerance against peers that go offline.

Kademlia is one of the most popular DHT implementations for P2P content-sharing. The identifiers are randomly generated and a XOR metric evaluates the distance between identifiers. It is used in the distributed BitTorrent trackers and the indexation scheme of the eMule/Kad network.

[1] Know in BitTorrent as trackers and in eMule as eDonkey servers

2.2.1 Kademlia system for indexation

Kademlia is a general purpose P2P DHT based on XOR metrics originally proposed by Maymounkov et al.[7]. In Kademlia every node has a 160-bit identifier randomly generated before joining the network. The distance between two nodes is defined as the XOR operation between each of the node's IDs. This distance, when the operation in a fully populated tree of 160 bits that is equivalent to the height of the smallest tree that contains both nodes.

For each i between 0 to 160, every <ip, port, id> triplets of nodes that are in distance 2^i and 2^{i+1} from itself are saved and called K-buckets. When a new node joins the network, it looks for itself and starts taking note of the closest nodes. It keeps record of the k closest known nodes(usually $k = 20$). These known nodes are used to route query messages to the other nodes.

To store information in Kademlia, a 160 bits key is created and the nodes which IDs are the closest to the key value will store data for that key. To locate the stored data, the closest nodes to the key identifying the data should be found. To locate a node by its identifier you should ask to the closest nodes you know, they will reply with the closest nodes they know and you iterate with the new closest nodes to the target until you reach the target node.

2.2.2 The eMule implementation: Kad

The eMule network uses a variation of Kademlia called Kad where the size of the keys is 128 bits[9] (since eMule uses MD4 hashes as identifiers for files to keep compatibility with ED2K networks). Kad has two index levels to provide the fully double indexation mechanism. The first level maps MD4 hashes of keywords to file identifiers (MD4 of the file contents) and the second level maps those file identifiers to the sources that share the file.

Previous work[13][14] has shown that Kad has a more reliable Kademlia implementation in terms of security than BitTorrent DHTs.

2.2.3 The BitTorrent implementations - Mainline and Vuze

There are two Kademlia implementations for BitTorrent's DHT: Mainline or Vuze, both incompatible between themselves. All content in BitTorrent has a metadata file associated. This file, known as info-file, contains all the necessary information to fulfill the transfer.

Both implementations of DHT use 160 bits keys for indexing as proposed in the first paper describing Kademlia[7] since BitTorrent uses as key the `sha1` hash of the info-file also known as the info-hash. The BitTorrent DHT implementations only have one level of indexation that maps content identifiers to peers, which is the reason why they need to access the web or other channels to find the file identifiers in order to fulfill the first step in a file sharing service.

The links in the web proving the info-hash to start the info-file transfer are called magnet-links and follow an standard defined by the magnet-uri project[8].

2.3 Ed2k

The ED2K protocol was developed originally for the eDonkey2000 network. Each file is divided into 9.28 MB parts, additionally those pieces are split in blocks of 180 KB each. Each peer has two queues for transfers: one to handle requests for transfer from consumers (upload queue) and another to track requests send to others for parts needed (download queue). All requests are part-wise, but recovery from interrupted transfers uses block information.

When a peer asks for a part to another peer, the peer providing the file replies announcing how many users asked beforehand and waits until finishing with those previous requests to start the transfer of the requested piece of data. Improvements related to peer exchange and credit systems were implemented in some alternative clients (i.e. eMule[6]).

The original eDonkey2000 client was shut down in 2006 because of legal issues [1] [2]. Since then, eMule took the central place in the scene when talking about standardization and protocol changes and most the clients claiming to be compatible with eDonkey2000 are actually compatible with eMule nowadays.

As consequence of the legal issues, all the original indexing servers where shutdown when eDonkey2000 was, nevertheless there are some free implementations around that allow final users to run their own server.

2.3.1 eMule

The eMule client is an eDonkey2000 open source alternative for windows started in 2002. Since then, eMule has been expanding the original ED2K protocol with new non-official functionalities like peer exchange, credit systems and protocol obfuscation[6].

The eDonkey2000 network used P2P just for transfer and a centralized index services based on a set of servers distributed around the world. Around 2003 some countries started auditing and regulating Internet content transfers. Starting in 2004 P2P file sharing communities were prosecuted in some countries because users indexed copyrighted material for distribution. In some cases, the local governments took action to censor or block the central index (i.e. Kazaa [1] [2]). The servers who had content indexed that was considered illegal by the country where the server was running were shutdown) having a worldwide impact on the users (those transferring legally or not). Since then, the P2P content-sharing networks had to evolve into pure P2P to avoid being compromised by conflicts or blackout of a few nodes.

The eMule project developed the Kad DHT based in Kademlia with two index levels, one for mapping file identifiers to peers, and one to map keywords to file identifiers; so search for content can be done inside the network[9]. The Kad guidelines for implementation of clients making use of the Kad network are very strict, it is not allowed to index anything but files shared using ED2K protocol. The source code is available under GPLv2+ in VC++ and can be obtained from `http://www.emule-project.com`.

2.3.2 aMule

The aMule client is a fork of xMule, a multiplatform open source alternative implementation compatible with eMule. It is the second most used client with active development nowadays.

The original fully monolithic architecture has been replaced with a modular one using ECP[2] to separate the core functionality from other components that provide different kinds of user interfaces. Nevertheless, there is still the option to get a monolithic build from the same source code.

The aMule components are:

- aMule Daemon: Handles all the file-sharing work and can be connected to any user interface provided by aMule from a local or remote origin.
- aMule Remote GUI: Similar to eMule GUI and implemented using wxWidgets library.
- aMule Web: A service that provides user interface using HTTP protocol, it's based on PHP code so it can be easily extended by webdevelopers.
- aMule Cmd: A simple user interface for command line.

[2] External Connection Protocol

- aMule: The monolithic version that resembles a Daemon + Remote Gui squashed together in a single binary.

The source code is available under GPLv2+ in C++ with wxWidgets library and can be obtained from `http://www.amule.org`.

2.4 BitTorrent

Pareto efficiency is a concept from economics. In a Pareto efficient economic allocation, no one can be made better off without making at least one individual worse off. Given an initial allocation of goods among a set of individuals, a change to a different allocation that makes at least one individual better off without making any other individual worse off is called a Pareto improvement. An allocation is defined as Pareto efficient when no further Pareto improvements can be made.

To achieve Pareto efficiency, BitTorrent protocol uses tit-for-tat strategy. An agent using this strategy will first cooperate, then subsequently replicate an opponent's previous action. If the opponent previously was cooperative, the agent is cooperative. If not, the agent is not.

The first implementation of BitTorrent had centralized resources called trackers that were responsible of providing the means to set together peers interested in an specific content, creating what we call *swarms*. The participants of a swarm follow a set of defined rules to maximize availability of content and redistribute the cost of upload and download between them [4].

After some years, a trackerless implementation was provided using Kademlia implementations, being Vuze and Mainline the most popular ones. The trackerless implementation of BitTorrent has only the ability to find sources for files when the info-hash is known. Nowadays, most clients, when processing an info-file which includes a tracker, after request a list of nodes to it check in any DHT to find more peers. A secondary network for keyword search is needed and the web is the most used for this mean.

BitTorrent transfer strategies are divided in 2 groups, those to select which piece to download, and those to chose which peer to choke/unchoke. Choking is a temporary refusal of transfer to another peer, although transfer from that peer are not blocked.

- Piece selection
  - Strict priority: Once a subpiece of a piece has been obtained, transferring the remaining subpiece of that piece has priority over anything else.
  - Rarest first: When starting the transfer of any piece, but the first downloaded, the less available piece is requested.
  - Random first piece: The first piece of the transfer is selected randomly.
  - End game mode: Once all sub-pieces which a peer doesn't have are actively being requested it sends requests for all sub-pieces to all peers.
- Choking algorithms used
  - Pareto efficiency: BitTorrent tries to maximize the reciprocate of upload connections.
  - BitTorrent choking algorithm: BitTorrent unchokes a fixed amount of peers to try to saturate upload capacity, decision is based in download rate.
  - Optimistic unchoking: Every 3 cycles, a single peer is unchocked to check if there is some better option than keep going with those peers already unchoked.
  - Anti-snubbing: If not getting data from a peer for a minute, that peer is chocked.
  - Upload only: Once the download is complete and there is no download rates in which to base decisions, the upload rate is used to maximize availability.

Changes to the BitTorrent protocol are proposed and decided by the community in `www.bittorrent.org`.

It is important to notice from previous description of Kad and ED2K network that keys in Kad come from MD4 hashes while in the other hand BitTorrent tracking sites index content structured in directories and files as a whole using as key the SHA1 hash of the info-file that describes the files and their relative location. This has an impact in the way we index BitTorrent info-hashes since they have 160bits and Kad only has 128bits reserved for keys.

2.4.1 The BitTorrent client

BitTorrent is the official client originally developed by Bram Cohen and the protocol specification is publicly available in the BitTorrent site (`http://www.bittorrent.org`).

BitTorrent uses Mainline DHT to map info-hashes to peers providing its info-file. The search for the file by keywords is handled outside of BitTorrent using the Web or other channels.

In previous work BitTorrent transfer speed was compared with eMule in several scenarios showing that the download strategies used by BitTorrent perform better than those used by eMule[14].

BitTorrent was originally developed in Python in 2001 and licensed under MIT. Since version 4 the code was closed by BitTorrent Inc.. However, several open-source alternative clients were developed before the code was closed as μTorrent, azureus and rTorrent.

2.4.2 Vuze (formerly azureus)

Vuze is one of the most deployed alternatives to BitTorrent clients. Their developers implemented a lot of extra features that are not in BitTorrent standard, also they provide an easy interface to develop third party plug-ins.

The Kademlia based DHT used by Vuze is not Mainline, neither it is compatible with it.

The source code is developed in Java under an hybrid license having some components under proprietary license and some under GPLv3. The open source pieces of code can be downloaded from `http://dev.vuze.com/`.

2.4.3 rTorrent

The rTorrent is a command line BitTorrent client whose core is also released as libtorrent (rakshasa) and used to implement other clients.

The source code is available under GPLv2+ in C++ and can be obtained from `http://libtorrent.rakshasa.no`.

2.4.4 libtorrent-rasterbar

LibTorrent-rasterbar is a library implementing the BitTorrent protocol and a set of related tools needed to build a complete client in a few lines of code, including session management, plugins management, DHT access, metadata exchange and persistence of the content. The build from source code offers the option to generate Python and Ruby bindings.

LibTorrent-rasterbar allows to externally define extensions to the protocol through its API.

The source code is available under BSD license in C++ and can be obtained from `http://www.rasterbar.com/products/libtorrent/`.

2.5 Summary

We introduced the concept of DHT and described a few implementations of it based in Kademlia specifications. Most remarkable difference between the DHT implementations is that Kad has two levels of indexation while others only have one. Having only one level of indexation is not enough to provide pure P2P content-sharing.

We introduced the ideas behind ED2K and BitTorrent transfer protocols. Furthermore, we survey a set of implementation of those protocols, some as clients and others as libraries.

In following chapter, this concepts and survey will be used to support the architecture decisions taken in prototype design and implementation.

3. HMULE: A KAD-BITTORRENT HYBRID CLIENT

3.1 Introduction

We introduce our global vision of how today *Mule and BitTorrent clients interact with other components and between themselves as well as our expectations towards an hybrid approach with hMule [15]. After we reviewed the global interaction of the pieces, we present a set of use cases and a list of compatibility requirements from it.

The use cases are bound to specific clients (*Mule, hMule, BitTorrent) so we don't oversight the issues that can rise in other clients as results of our changes.

After the requirements are completely described, we explain why we chose to extend the aMule client and the architectural changes proposed to it in order to fulfill these requirements.

Finally, we introduce the protocol changes and analyze how they are compatible with all the scenarios we required.

We keep the internals of experimentation development for next chapter, after discussing our experimentation plan.

3.2 Overview

In table 3.1 we see how different clients interact when seeking content and sources, for the subsequent file-transfer.

We can observe that BitTorrent clients need to search by keyword in the Web or in another external resource to retrieve the info-file or the info-hash. Those peers who obtain the info-file can join a swarm using a tracker and avoid being connected to any global DHT sharing only with those connected to the same tracker. Sometimes, users are required to login to join a (private) tracker.

The peers having only the info-hash (using magnet links most of the time) have to join a DHT (Vuze or Mainline) to look for peers sharing the wanted content or ask to a known peer if it has the info-file (using peers known from other swarm can work). Vuze client can join both DHT's so they have access to more peers to share with than those peers who only connect to mainline DHT. It's important to notice that any swarm can be joined by any client using BitTorrent protocol, the only resource that is not shared between

Client	Lookup by keyword	Search sources	Transfer protocol
aMule eMule	Kad ed2k-server third party (magnet from Web, irc, mail...)	Kad ed2k-server	ED2K
BitTorent μTorrent rTorrent	third party (magnet from Web, irc, mail...)	Mainline tracker swarm	BitTorrent
Vuze	Vuze search engine third party (magnet from Web, IRC, mail...)	Vuze DHT Mainline tracker swarm	BitTorrent

Tab. 3.1: Interaction of clients without hMule

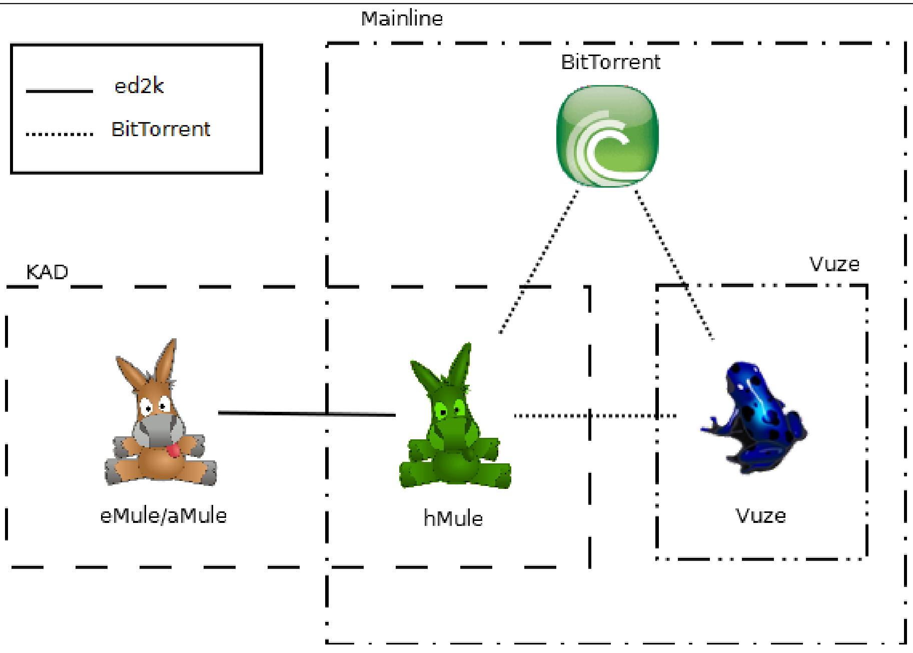


Fig. 3.1: All clients interaction and the DHT they collaborate with.

all of them is the DHT.

Most *Mule clients search by keywords in Kad, some users don't use this feature and prefer to use similar indexing servers that those that were used by eDonkey2000. After a file has been chosen from the list of results provided by Kad when looking up by keywords, another query is sent to Kad looking for sources. When a set of sources is obtained, the consumer asks to be subscribed into the upload queue of each source for the desired file. *Mule clients reply this request by providing the metadata of the file, which contains information about the content such as format, name, size, and several others. Then they queue the request for transfer in its upload queue.

The difference between BitTorrent and *Mule about the order of the steps taken to obtain metadata is important, because it changes assumptions in the protocols, i.e. The BitTorrent info-file is a unique metadata for a content, while the metadata for *Mule transfers can have as many as detected sources providing it.

In figure 3.1 we can see how we expect our developed client to interact with all those networks and get the best out of each. Also we plan to work as a kind of bridge of content between networks, providing to both networks what we obtained from any of them.

3.3 Requirements analysis

3.3.1 Use cases

We categorize our cases by participants involved in the interaction

- Request started by an hMule client using Kad
  - Start a query from Kad with a keyword to get possible files.
  - When Kad replies to the query, prompt the user to select the file to download.
  - Start a query in Kad with the selected file to get sources.
  - For each source
    - query source for metadata about the file.

        * when the client replying us is an hMule client, metadata should include enough information to identify the file as content in the BitTorrent network The download will be added to BitTorrent's transfer queues, and a swarm will be joined using the peer providing the metadata as entry point.
        * otherwise metadata should be added to the ED2K queue.
    - If BitTorrent protocol was not used, when transfer is complete, generate and store the info-file and info-hash to provide it in future transfers.

- Request by hMule to BitTorrent network having info-file or info-hash
    - User introduces the info-file or info-hash.
    - Start download using BitTorrent protocol, trackers and Mainline DHT if needed.
    - When transfer is complete, compute Kad metadata and keywords.
    - Publish in Kad.
- Publish original new content from hMule
    - Compute BitTorrent info-file of the new content to be able to reply to BitTorrent requests for info-file from peers that only have the info-hash and register the Mainline DHT.
    - Compute Kad metadata and keywords.
    - Publish in Kad.

3.3.2 List of compatibility requirements

- hMule should be able to join Kad to search and publish content.
- hMule should be able to obtain content from info-files, info-hashes, ed2k-links, and make it available for all other *Mule clients by Kad.
- hMule should be able to transfer content to other clients using BitTorrent transfer protocol whenever possible (hMule or BitTorrent peers).
- hMule should be compatible with previous *Mule and BitTorrent clients:
    - It should be agnostic about transfer protocol when publishing content in Kad, meaning that a content should not have to be searched with different criteria depending on the transfer protocol desired to be used and Kad should return a valid result set for all kinds of user.
    - It should share the published files to old *Mule clients using the ED2K transfer protocol.
    - It should be able to join swarms with other BitTorrent clients using trackers or Mainline DHT.

3.4 Possible approaches and evaluation

To fulfill the requirements of the project several approaches were studied to combine them into a solution.

3.4.1 An hybrid client using libraries for Kad, BitTorrent and ed2k

Obtaining the Kad source code from eMule to use it as library is possible since other clients using Kad already had the need to do that, but extracting the ED2K transfer protocol and session components is unpractical, since eMule is tightly coupled between these functionalities and its main application loop. Also we couldn't find any open source libraries providing ED2K protocol, so it should be stripped from some *Mule.

Given that libraries providing Kad and ED2K functionalities were not available for production or needed exhaustive work to be extracted from original implementation, we can't build our own prototype based in just libraries. We suggest then, to start form a BitTorrent client and build over it the needed functionalities to interact with other eMule-compatible clients, or to start from a eMule-compatible client and build over it the BitTorrent transfer mechanisms.

3.4.2 Extending a BitTorrent client to use Kad-DHT

We can use Vuze and implement a plugin to access Kad from it. Kad file identifiers are smaller (128 bits) than the files used to communicate metadata between BitTorrent peers (known as info-files) which has no fixed length and can take several bytes. The difference of length in the metadata needed to start a download brings up the necessity to define a new identifier for those contents provided using BitTorrent or a change to the Kad structure. The Kad structure of a DHT can't be changed by one single client. The problem about variable lenght of identification of content in BitTorrent was also noticed by the BitTorrent DHTs developers, so they indexed their DHT implementations using info-hashes of 160 bits (info-hash). BitTorrent already provides a protocol to exchange info-files between peers. To use the info-file exchange you need to know the info-hash and at least one source providing the content. To take advantage of the info-file exchange protocol we need the info-hash, which doesn't fit into the 128 bits of Kad keys. There is no direct translation to compute an info-hash from the Kad identifier without losing backward compatibility with Kad or BitTorrent.

To overcome these problems we thought to include the info-hash as a keyword when publishing the file, but some clients drop keywords or override them when other publication for the same file arrives (probably from an old client that doesn't provide this information). As we (at least until our client becomes popular enough) are a small minority in the network, there is a high probability that our info-hash gets eclipsed by other clients providing the same content.

The solution we came up for the issue matching Kad file identifiers to info-files was to implement a new protocol message that has a 128 bits identifier. The peers that share the file ask for the info-file to clients identified as sources by Kad and if a client replies with the info-file, content transfer can be started using BitTorrent protocol.

This approach only provides a shared index between networks that got misleading info for both kind of users, those using BitTorrent can find transfers that can only be acquired using ED2K protocol and those using ED2K can find content that can only be acquired using BitTorrent protocol for transfers. Those clients following strictly the eMule guidelines of use for Kad will ban us from the network, they are the most deployed in the network right now. So, we should need an implementation of ED2K protocol to get and provide files when there is no BitTorrent sources to avoid the exclusion from Kad.

We could try to filter results in our client so only those available for BitTorrent transfer appear, but filtering is complex because it can not be done without asking each source of each result what application version got installed. Asking to all sources for all results is overwhelming, and registering if the file is provided or not by an hybrid as a keyword in Kad is not possible for the same reasons that info-hashes can't.

3.4.3 Extending an eMule client to support BitTorrent as transfer protocol

Libraries providing BitTorrent were evaluated and both of them provide a good start for a BitTorrent client. The Kad protocol is working in any eMule client, but we still need a way to map the 128 bits identifiers to the 160 bits info-hashes to use the info-file exchange extension in BitTorrent libraries. This can be introduced as a libtorrent extension or hooked to some eMule operation when bootstrapping downloads.

We also know that eMule clients asks every new source added for metadata. We can take advantage of this by extending the eMule protocol. So when our client replies, it also comes up with the info-hash we require.

3.4.4 Chosen approach

We decided to expand an eMule-compatible client to support BitTorrent as an alternative transfer protocol.

- We use aMule as an eMule-compatible client since our experimentation infrastructure is based on Linux and also because the aMule project has a more modular architecture than eMule.
- We use libtorrent-rasterbar to provide the BitTorrent protocol. It has a complete high level interface to handle sessions, individual torrents and peers, and also it has access to low level interfaces to do adjustments when needed. The implementation is based on the widely known Boost libraries that makes it easily portable to other operating systems it provides interfaces to develop extensions. Extentions for Mainline DHT, metadata exchange between peers, μpnp and IPFilters are already offered by the library.

3.5 Changes proposed to the aMule client

We got to retrieve the BitTorrent metadata from peers. When a file is added to the download queue in an aMule, it asks every source for the metadata. There is no need to implement a message to explicitly request the info-hash since OP_REQUIRE_FILENAME is sent to all known sources and expect several asynchronous replies with different kinds of metadata such as user comments, media tags, etc. We should be able to reply the request for metadata with one more asynchronous message including info-hash and peer's port used for BitTorrent protocol. If a peer replies with the info-hash, that is enough to say it provides BitTorrent services and an info-file exchange request should be sent to this peer using the already implemented methods in the LibTorrent-rasterbar.

3.5.1 Protocol changes

- The new message providing the info-hash and the openned port number to listen for BitTorrent protocol messages will be implemented as part of eMule extended protocol and the message will be named OP_BTIH.
- Some extra messages should be defined in the EC protocol to check Torrent specific stats, add new downloads starting from info-files, info-hash or magnet links, disable or enable Mainline DHT support.

In figure 3.2 we can see a simple flow of messages intervening in the download of a file in the modified client hMule when the file is obtained using the BitTorrent protocol for transfer.

The changes in messages are numbered:

- The new implemented message, providing the info-hash and which port is open for listening to BitTorrent protocol, is sent from sources having hMule
- The info-file is requested to the peer who provided the info-hash to the announced port using BitTorrent metadata-extension.
- The info-file is obtained using BitTorrent metadata exchange extension
- The swarm is joined and the content to start the transfer.

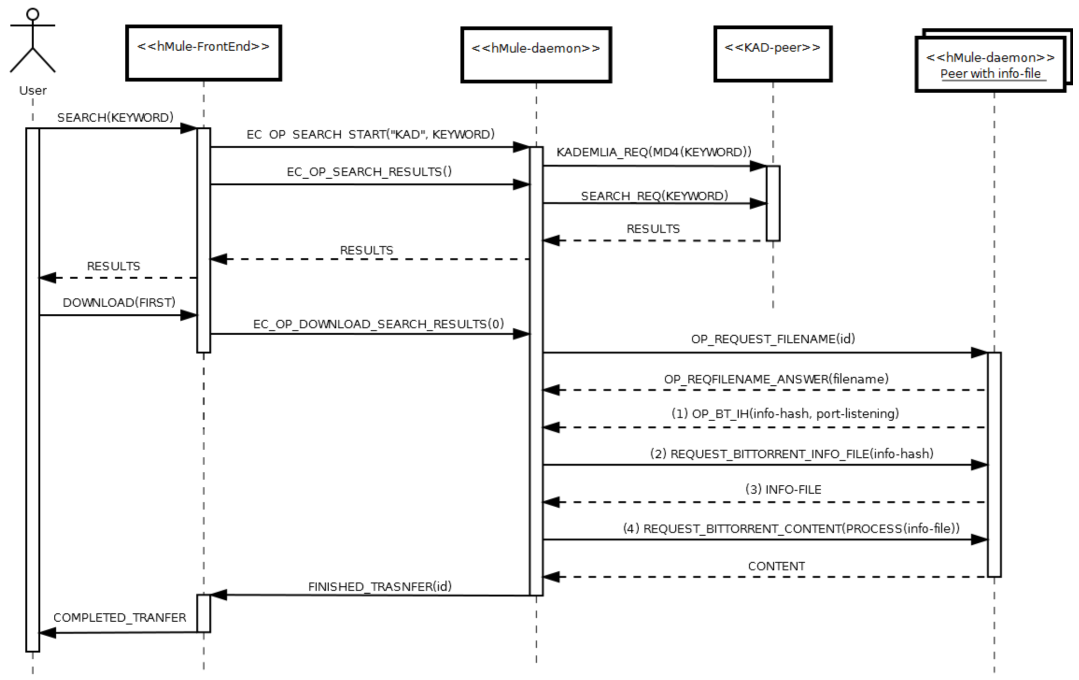


Fig. 3.2: Sequence diagram of an hMule search and transfer using BitTorrent protocol

3.5.2 Application changes

- To reply with OP_BITH, an internal dictionary that maps eMule file IDs to BitTorrent metadata hashes should be maintained internally. This dictionary should persist between sessions since torrent metadata coming from different sources has different metadata hash for the same content and we don't want to lose the original swarms.
- To provide backward compatibility, when a file was fully transferred using ED2K protocol, the info-file is generated and it is added to the dictionary. So users with hMule client can retrieve info-file and info-hash to use the BitTorrent protocol. An external torrent file should be able to be added to the session. After being fully transferred using BitTorrent protocol, the file should be published in Kad and the dictionary updated.
- A class should be defined to wrap the LibTorrent-rasterbar and a coordinator between transfer protocols should be defined to avoid overlap of efforts and keep on deciding which protocol suits better at every time. This coordinator should provide a simple way to change strategies for experimental uses.
- A translation between wxWidgets structures and Boost ones should be provided in some cases.
- Building scripts should be modified to provide a non-hybrid built and to detect the required libraries.

3.6 Current implementation details

The interface between aMule and LibTorrent-rasterbar was implemented defining a namespace torrent that contains a class *Torrent* to handle the wrapping joint with a class *TorrentMuleMapping* that keeps track of the metadata relations.

The *TorrentMuleMapping* class is internally represented as 3 unordered maps to index the relations (used when searching by key) and a *vector* (used to iterate those relations). The relations have 4 fields, a *CMD4hash*[1], a *SHA1hash*[2], a *boost::filesystem::path* that should point the info-file and a status of download

[1] Type of identifier used for files in aMule
[2] Type of identifier used for the BitTorrent content

enum. Several getters and update methods are defined to access the container, including constant iteration.

The *Torrent* class is implemented as a singleton and contains internally the libtorrent session, an instance of *TorrentMuleMapping* and methods to wrap the libtorrent behavior as *createMetadataForFile* (which generates the torrent metadata for a known file). Torrent class also initializes and calls the *TorrentStrategy* used to select protocols for transfer, which is selected in the app's configuration file.

3.6.1 Initializing and shutting down the application

The main application class, called *amuleApp*, has an *OnInit* method and an *OnExit* method that are called at start and end of the application run. There are also specific methods for subapplication customization of those methods.

When an application is launched and *OnInit* is called, aMule reads all the preferences of the user from the configuration file, initializes the *DownloadQueue* and *UploadQueue*, reads a set of metadata files called PartMet files to register the current metadata in the download queue. There are some extra activities handled in the *OnInit* method based in configuration, i.e. join eDonkey2000 servers and start running Kad. They try to join the chosen networks and, as soon as they connect, they start looking for sources for all the queued downloads.

Once all ports for aMule connections are open, the *Torrent* class instance is created setting the ports for BitTorrent connections (by default port 7000, but tries others if that one is taken). The *Torrent* class uses two configuration options to select where to persist the info-files and where to persist the dictionary between sessions. The strategy chosen is read from the configuration file only once, so if you want to change it, session should be restarted.

After the download queue is completly loaded, the method *refreshMetadata* in the Torrent class is called. This method iterates the *TorrentMuleMap* to load all the metadata known from previous sessions, then checks the torrent directory defined in the configuration for info-files that are not mapped yet and loads them. This works as a dropping directory for those not using the amulecmd UI.

Then aMule calls *reloadFiles* method in *SharedFilesList* class to update the metadata of the shared files, this method was modified so it calls *addFilesFromDirectory* method of the *Torrent* class to update the info-files and the information in the *TorrentMuleMap*.

When the application shutdowns, it calls *OnExit* method which handles the persistence between sessions and shutdown of services. We modified the *OnExit* method to include info-files and *MuleToTorrentMap* in the set of objects to persist and call the shutdown methods for the BitTorrent session.

3.6.2 Starting a download from Kad

In previous section, we showed the flow of messages intervening in a search for a file in Kad and its transfer between hMule clients using BitTorrent protocol (fig. 3.2).

In figure 3.3 we show how different objects collaborate to start a transfer with an Id obtained from Kad but using BitTorrent as transfer protocol.

Firstly, user provides the Id of the file to be downloaded to hMuleApp (the class running the main loop in the application), hMuleApp calls to the ED2K DownloadQueue singleton object to add the Id. The DownloadQueue calls to the Kad object to find sources. For each source that Kad finds, it calls the DownloadQueue to inform of it. When DowloadQueue receives a new source, it tries to contact it sending an OP_REQUIRE_FILENAME message which is used to request for the metadata of the file. If the source uses the extended eMule protocol, it will reply with several message with different information as idv3, comments, filename and others. If the source can transfer the file using BitTorrent protocol, it also sends an OP_BT_IH message which carries the info-hash of the file we want to transfer and the listening port for BitTorrent interactions. When a OP_BT_IH message is received by hMule, the info-hash of the file

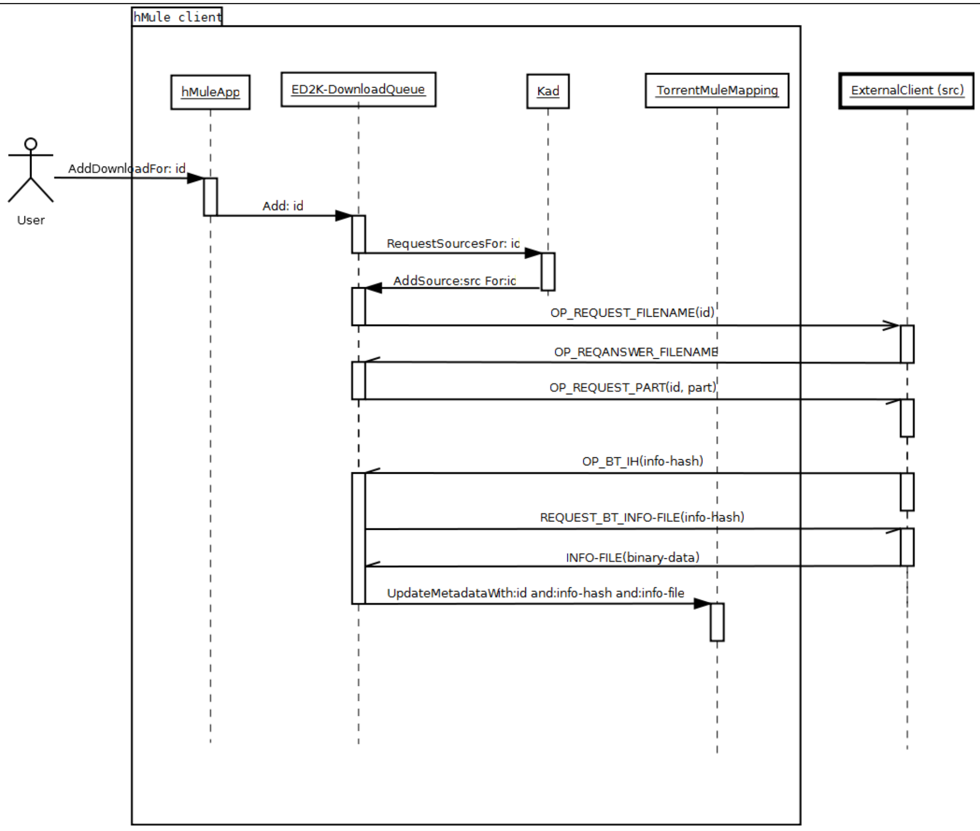


Fig. 3.3: Sequence diagram for how objects interact to start a transfer in hMule when a peer providing BitTorrent transfers is available

is known, so a request for info-file is sent to that peer. After info-file is obtained, the Id, info-hash and info-file are introduced into the TorrentMuleMapping object so the strategy starts deciding what to do about protocols.

While hMule doesn't receive an OP_BT_IH message it just transfers using the ED2K protocol.

Hooks:

- When our client receives metadata from sources, it processes it using the *processFileInfo* method in *DownloadClient* class. This one was modified to check for OP_BT_IH messages. If any OP_BT_IH messsage were received, the information contained is passed to the *Torrent* class using the *addTorrentUsingSHA1AndPeer* method. This method adds the ip and port of the peer that provided the information and the info-hash of the file to the BitTorrent session and asks to the known peer for the info-file exchange.

- The main loop of the application has a second thread to process asynchronous tasks. Every time a loop is completed the process method of download and upload queues is called. A call to the process method in Torrent class was added to the loop so asynchronous alerts from LibTorrent-rasterbar can be attended and *TorrentStrategy* can switch protocols as needed.

- After a file is fully downloaded using BitTorrent protocol, a FINISH_DOWNLOAD alert is received by the *TorrentStrategy* object. If the MD4 hash of the content matches the identifier of the file, it is flagged as completely downloaded and moved to the incoming directory, otherwise the file pieces are checked using AICH and those pieces that failed the verification process are downloaded again.

Notice that the identifier we publish in Kad for a content doesn't change according to the protocols available to transfer it. The unique identifier allows us to switch between transfer protocols at any time during the transfer without the need for another search. Also it should allow us to implement the use of both networks simultaneously to download a single content.

3.6.3 Strategies for referee

The referee described in previous section is an abstract class which should be subclassed. Two basic strategies were implemented.

The first one saves metadata given by others, provides the OP_BT_IH replies to other peers requesting it and transfers content to others but doesn't use BitTorrent protocol to download content. This strategy was named *NoTorrentStrategy* and can be selected in the configuration file by setting **strategy=0**.

The second strategy handles the full transfer using BitTorrent protocol when the info-file is obtained from another peer. This strategy was named *AlwaysFallToTorrentStrategy* and can be selected in the configuration file by setting **strategy=1**.

3.6.4 Starting a download from info-file

When an info-file is provided from the web or another source, it needs to be added to the LibTorrent-rasterbar session to be downloaded. After finished downloading the content, it is published in Kad.

The file path can be provided in the `add` command of amulecmd or dropping the file in the torrent directory configured. If the use of mainline DHT is enabled, `add` command also allows the use of magnet links.

Mainline DHT can be enabled using `connect mainline` command in aMulecmd.

3.6.5 Sharing a new file

When a new file is added in the sharing directory and `reload` button or `reload` command is executed, the SharedFilesList's reload method is called. We extended the code so a similar operation is called in Torrent class to create the necesary info-files.

3.6.6 Additional changes

Autoconf scripts have been modified to check for the new libraries needed in the software. An **–enable-torrent** option has been added to the *./configure* script so users can choose to build the original aMule client or our modified version.

3.7 Summary

To provide the prototype we started choosing the scenarios on which we expect our client to work. Distilled from those scenarios, we provided a list of use cases and actors with the compatibility requirements needed to interact against them.

When our survey of requirements was completed, we evaluated how we could combine the different libraries and clients described in previous chapter to cover these use cases. Firstly, we decided to start our prototype forking from aMule code base. Secondly, we choosen to use LibTorrent-rasterbar for the BitTorrent protocol and mechanisms. Finally, for the glue between networks, we extended the eMule protocol with a new message which notifies the info-hash associated to a file when metadata is required between peers.

After explain how we planned to develop our prototype, we describe how we put everything together. We show the way hMule clients collaborate to produce a transfer between them using the right protocols. Some important changes are hooked to initialization and shutting down events, where application had to handle new connections and sessions. Two strategy classes, which decide when to use each transfer protocol and how they collaborate to each other, are provided. Also, we supply an abstract strategy class, which may be easily extended for research and development of new collaborative schemes.

More details about the final implementation can be found in the appendices where a user guide, an installation guide and some fragments of code for most relevant classes are shown.

4. EXPERIMENTATION

4.1 Introduction

In this chapter we explain how we conducted a set of experiments for our hybrid client using hMule.

The chapter is structured as follows. Firstly, we explain our experimentation goals. Secondly, we introduce PlanetLab as our experimentation workbench and the limitations we found when using it. Thirdly, we present the procedure used to run each experiment. Then, we describe the variables we are measured and the final setup of the experimentation workbench. Finally, we show the obtained results and analyze them.

4.2 Experiment goals

We conducted a series of experiments aiming to compare hMule's performance against the normal aMule client, in which we expect hMule to outperform aMule.

Our experiments have several goals:

- **Validate implementation functionalities.** Is our new client compatible with other clients using Kad? Is it compatible with other clients using BitTorrent protocol? We deployed some hMule peers, looked for content to obtain and then share it using the different alternatives the client provides: Kad, ED2K link, info-file with tracker, info-hash with mainline. After checking compatibility with previous clients, we checked whether swarms were maintained when hMule provides content downloaded from BitTorrent networks and files get published in Kad. Finally, we tested that files obtained from ED2K networks get transferred using BitTorrent protocol by any other hMule clients.
- **Compare content propagation speed.**
  - **Full-content propagation of newly published content.** How much time does it take for every single peer to complete the download? We aim to measure how fast a new content gets propagated within a set of peers starting from a single source.
  - **Content propagation when adding a new peer to a previously propagated content.** In the previous item, we described the situation of a content that was just published in the network and several peers asking for it. In that case we were interested in providing complete transfer to as many users as possible as soon as the content gets published, because most of the contents are requested by the majority of the users in the first few hours or minutes since its publishing, and only one source can provide it at that time. This creates a bottleneck for the distribution. This issue usually disappears after a few days or hours of the original publication time and we get into a different scenario where there is more sources than consumers. For the second scenario we did some measures when we added only one new node asking for a content already distributed in a set of nodes.
- **Compare sustainability of transfer rates.** We expect the nodes to maximize the accumulated transferred data between them as long as possible. This measure was used to try to understand how smartly the clients are obtaining resources and distributing the transfer load. Accumulated upload rate was also measured, but given that we are transferring data between nodes in a cluster, the output and input will always be equal (only a small packet lost difference). We use the upload measure only for experiment results validation.

  Also, we measured the progress in every peer and accumulated it. We used this measure to check how many useful data was propagated at a given time.

- **Evaluation of wasted payload transfer.** We measured how long (and how much) was payload transfer before the protocol was switched. We expect that this measure will motivate the implementation of new strategies in the future.

4.3 PlanetLab

PlanetLab[3] is a global research network that supports the development of new network services. PlanetLab currently consists of 1137 nodes at 544 sites providing Linux nodes where researchers can deploy their experiments for distributed storage, network mapping, peer-to-peer systems, distributed hash tables, etcetera.

For final users planet lab looks pretty much like a set of virtual machines connected to the Internet using a Fedora Linux distribution. Each node comes with a clean installation of just an operating system in a 8GB hard disk and the access to a root user account.

We were able to get 64 nodes from PlanetLab to use in our work. Some of those nodes usually have frequent outages and after they come back they usually come back as a clean installation. After a few weeks of dropping unstable nodes from our list we got 40 nodes with both clients installed and running in a stable environment.

When we started taking measures we noticed that PlanetLab bandwidth was too high for a standard residential user and it was even a problem for measuring the progress of the transfers, having in some nodes download and upload rates over 10Mbps.

After we did this experience, we tried to throttle down the bandwidth using the well known Linux Advanced Routing & Traffic Control = LARTC (LARTC)[1] traffic shaping tool. LARTC is a set of networking tools that provides a series of services, including traffic shaping and network behavior simulation to provide very realistic experimentation scenarios.

We found out that PlanetLab's virtualization does not currently support running *LARTC* commands because PlanetLab nodes run a single shared kernel that all slices use, so it restricts which syscalls you can use, even when you have a root account. In consequence the shaping using LARTC was not possible.

Another tool we tried was Trickle[2]. The advantage of Trickle over LARTC is that it is able to shape in user-space without the need of any special privilege. Anyway, Trickle requires a specific library linking scheme to work, and we found out that aMule was not completely compliant with it.

After we failed to shape the traffic bandwidth outside the application in several ways, we decided to limit it internally in the code. This approach is more error-prune and hard to check in terms of the trustiness of the data. We introduced a log on which every 15 seconds, the total download rate, total upload rate, payload download rate and payload upload rate were registered. The main purpose of this log was to validate that our client never uses more bandwidth than what it should. We noticed in some nodes that peaks over the allowed limit may occasionally appear, but never more than a 20% of the allowed bandwidth and never more than 1 or 2 per experiment. We considered this error in the measures affordable considering the platform limitations and given that we did several runs of each experiment.

4.4 Experiment design

We deployed aMule and hMule clients in each node provided by PlanetLab after filtering those ones with frequent outages. All client nodes in this experiment are running the hMule client using Strategy 1 (AlwaysFallToTorrent), and a normal aMule client as it was distributed at the time we branched the development source code base.

[1] `http://www.lartc.org/`
[2] `http://monkey.org/~marius/pages/?page=trickle`

All deployed nodes were using the standard configuration that came with their distribution package. We limited the bandwidth used for each node to a standard asymmetric kind of connection generally available in ISPs for home users (cablemodem 1.5Mbps/180Kbps).

We identified one node as *the lead* in each run of the experiment, this particular node was the one publishing the content to be transferred at the start of each experiment run. On the lead node we uploaded one file of the experiment files set, which includes files of 1KB, 1MB, 30MB and 300MB. This sizes are representative of common transferred files in an aMule network, other common sizes are 650M, 4.7G, 9G and 30G which we skipped because of space limitations in PlanetLab nodes, anyway we believe they should perform even better using BitTorrent protocol than smaller ones.

A single run of the experiment consists of a series of coordinated steps:

- We start one client in each node.
- We upload one of the experiment files to the lead node.
- The lead node publishes the content.
- As soon the content is published, all nodes search for the content every 2 minutes until finding a match for it and starting the downloading process for the file.
- Every 5 seconds the status of the transfer with several other variables is logged in each node (including upload bandwidth, download bandwidth, progress of the transfer, downloaded size, parts, known sources, known complete sources and others), also every 15 seconds a higher grained status is recorded with accumulative data separated by protocol (including TCP, Kad, peer-exchange, packets lost, etc).
- After a defined time (where at least 80% of the nodes completed the transfer) we shutdown all nodes.
- We download all nodes logs to a centralized place and merge them into a single experiment log for later analysis.
- All data recorded in internal structures is flushed so next time the client runs, it starts brand new.
- Configuration files are put in place for the next run.
- And we start again.

4.5 Experiment development

4.5.1 hMule changes to retrieve internal information

A few changes had to be introduced in the hMule client to extract experimental data. We modified the strategy AlwaysFallToTorrent to log the values of several session variables in the libtorrent session every 5 seconds in the Process method.

The logged variables were:

- Upload rate.
- Download rate.
- Upload payload rate (the upload rate considering only useful content transferred with no protocol overhead).
- Download payload rate (the download rate considering only useful content transferred with no protocol overhead).

- Progress.
- Known peers in swarm.
- Unchocked peers.
- Download total transferred bytes.
- Upload total transferred bytes.

Also all the debugging messages previously defined in the client for strategies using BitTorrent protocol were enabled at the time of the experiment.

This included a set of interesting alerts logged when:

- Metadata is asked to another peer.
- Metadata is obtained at first time and when it is provided to another peer.
- The switching protocol is called.
- A transfer is finished using BitTorrent protocol and the client needs to enter maintenance activities to republish in aMule network as fully downloaded.

4.6 Setup of the nodes

Since most of the nodes in PlanetLab have exactly the same operating system (Fedora Linux 8) and libraries deployed, we created a similar machine in our lab and we built everything locally. Once it worked as expected, we rsynced all binaries and libraries to every node. After deploying all the applications, we developed a set of bash scripts that reproduced the whole process described in the experiment design.

We scheduled several runs of each of the experiments using crontab daemon. The nodes clock difference was measured for reference since crontab was used to launch the experiments. The highest difference we found between the clocks was 2 seconds. Considering all obtained results are in the scale of hours or at least tens of minutes, the clock differences were treated as not significant enough.

In some runs of experiments we noticed that some nodes freeze and don't start the downloads because they don't find any sources, neither uploads to any other node using aMule. Those nodes which repeatedly had this behavior were removed from the experimentation valid nodes lists. We suspect there was an issue in those nodes opening the necessary TCP/UDP ports.

The aMule configuration was reset after each run to avoid favoritism from the credit system after different runs of the experiment.

4.7 Results

4.7.1 Implemented functionalities validation

We have deployed mixed setups with different ratios of aMule/hMule clients. After we tested that all our nodes can share the content between them, we did a second experiment where popular content was searched in the Kad network and downloaded by all our peers.

First we ran searches in Kad sequentially, one node at the time, waiting several minutes after each search to start the following one. When the last one started to download it was verified that it was using only BitTorrent resources.

Next, we repeated the test but starting downloads from all the nodes simultaneously. We observed, in this scenario, that the swarm wasn't joined by all possible peers since they join when a peer asks for metadata to others. When download is started, none of them can provide an info-file. Anyway, when at least one source finished the download, if a new peer asks for the same content afterwards it will gather all those peers together in a swarm.

Two more tests were run where peers running hMule successfully downloaded content using magnet links or info-files from the web. To keep new downloaders joining the peers in the original swarm, the original info-file should be provided since the same file can have more than one info-file associated. So, after downloading a content from BitTorrent networks, it was published in Kad. Then, a mixed set of peers searched for it in Kad and downloaded it using any of the 2 protocols. Those peers using BitTorrent transfer protocol joined the original swarm with peers using any BitTorrent client. This has validated that our client publishes in Kad the original info-file obtained from the BitTorrent networks when using them.

Tests also showed that downloads stale if the whole swarm goes offline when a file started download using BitTorrent protocol and AlwaysFallToTorrentStrategy. Anyway, after a new peer connects (or some of the previous reconnect) the transfer is resumed using the new peers detected in the aMule network.

Given the nature of PlanetLab limitation about using limit for bandwidth we could not measure how the protocols compete with each other for resources when working in an hybrid situation. We suggest that when using BitTorrent protocol, to maximize the odds of getting the files fast, a low limit should be applied over ED2K connections. This way, you can keep maintaining your position in the queue and use the most bandwidth in the protocol, which provides a faster download. In our implementation the internal limit bandwidth could be set in each protocol independently. Anyway, limiting both of them with specific values while externally not limited at all, will perform the same as having 2 different clients running.

4.7.2 Content propagation speed

4.7.2.1 Full content propagation of newly published content.

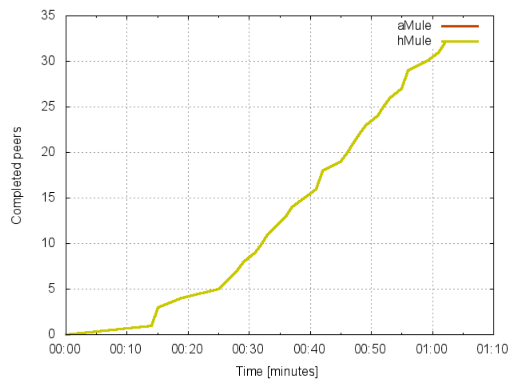


Fig. 4.1: Propagation of 1KB files in a 32 nodes set

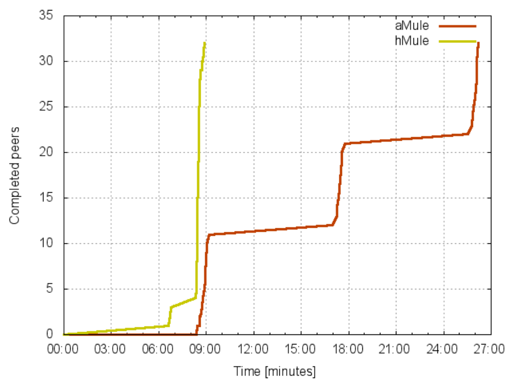


Fig. 4.2: Propagation of 1MB files in a 32 nodes set

Files with just 1 KB of content always transferred faster than the metadata exchange for protocol switch in hMule. That means that the transfer in hMule had a small overhead (a few bytes) over the aMule client because of the unused require for metadata. Figure 4.1 shows how a 1K file gets propagated to 32 nodes for both clients as reference.

Figure 4.2 presents a slight advantage for the first 10 nodes in which the transfer of a 1MB file is completed when using hMule rather than aMule. Anyway when the quantity of nodes requesting the file is more than 10, the completion takes one third of the time taken when using aMule. The plot includes the time taken

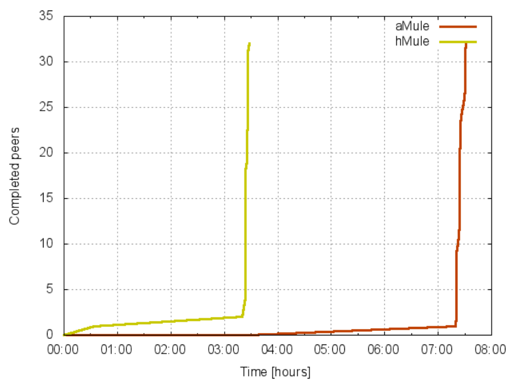


Fig. 4.3: Propagation of 30MB files in a 32 nodes set

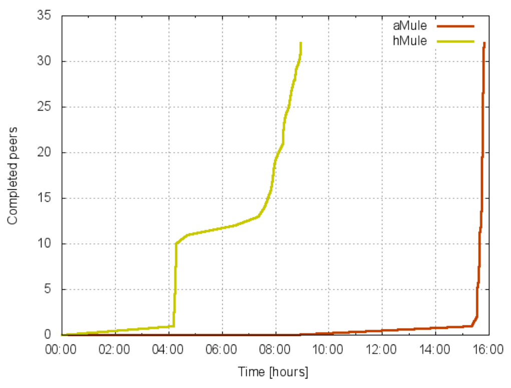


Fig. 4.4: Propagation of 300MB files in a 32 nodes set

by peers to recognize each other as hMule compatible and the time used to exchange metadata to start the transfer, this takes up to 2 minutes.

The transfer using aMule shows a step-like pattern. We think this is because the first few nodes which asked for the file are getting equal quantity of transfer bandwidth from the original peer. This transfer is not fragmented. The file has a single piece of 1MB, so there is no place for optimization of distribution in aMule. After a while, each transfer completes for each peer that was at the top of the queue, giving us the first step in the chart.

After the first set of peers completed the download, it takes some time for others to notice them as new providers. So, the second set of peers in the queue to be processed doesn't use them. That is why we see that every step in the stair got pretty much the same quantity of nodes. This suggests that the quantity of completed nodes doesn't increase the availability of content right away. Given enough time, the peers who completed the download will be seen as new sources for the other peers. In the meantime a lot of usable transfer capability is wasted.

When using BitTorrent protocol, sometimes we had to wait for a few seconds or minutes before the hMule clients recognize each other as such and transfer the info-file between them before starting the transfer. Content of 1MB is small enough to avoid being break down into pieces by BitTorrent, same as in aMule. Anyway, BitTorrent's strategy worked a lot better, it was able to propagate to 32 nodes in one third of the time taken by aMule.

In figure 4.3 we used files with 30MB of content for 32 peers. In this case the file in aMule is split in 4 pieces giving the queue more freedom to behave and try to optimize the propagation. However, all runs took between 7 and 8 hours to propagate to 32 peers while it took 3 to 4 hours in hMule for the same amount of nodes when using BitTorrent protocol for transfer.

In this case both protocols have issues with the last piece. They couldn't increment its availability enough. Since almost all nodes have bottleneck to get the last piece, they all end almost at the same time. It is important to notice that the slope of BitTorrent protocol is higher than the one of ED2K protocol, probably because the pieces size is smaller.

When using bigger files in figure 4.4 there was no win for any client running aMule and all clients completed the download between 15 and 16 hours. In the other hand, using BitTorrent protocol, a small set of nodes is prioritized to generate seeds (3 in this case) and then, those nodes help to speed up all others. Those using hMule could complete the transfer in less than half of the time taken by those using aMule.

Last plot shows that, when the file is bigger, BitTorrent strategies to provide availability are more effective to avoid bottleneck in the last piece, while ED2K has no improve at all.

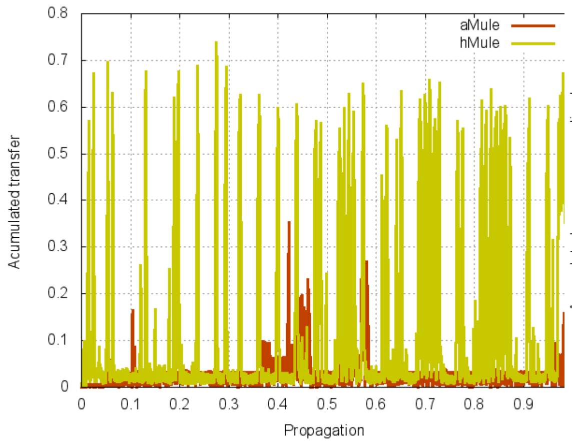
Client	1KB	1MB	30M	300M
aMule	12s	44s	17m	38m
hMule	12s	44s	12m	28m

Tab. 4.1: Content propagation of a new node in a seed-like environment.

4.7.2.2 Content propagation when adding a new peer to a previously propagated content.

In table 4.1 we can see there is no improvement in the use of BitTorrent protocol when transferring small files. However, the BitTorrent protocol keeps the throughput saturated for more time and there is an important improvement when transferring larger files.

4.7.3 Sustainability of transfer rates



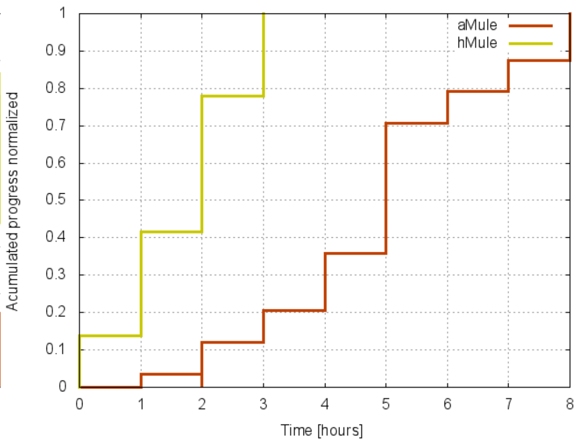


Fig. 4.5: Accumulated download ratio normalized for 30MB files

Fig. 4.6: Accumulated download progress normalized for 30MB files

Figure 4.5 compares the accumulated transfer rate between both clients for a 30MB file distribution. The normalization for transfer rate is based in the max upload bandwidth, so value of 1 would be all nodes uploading at full upload capacities.

Figure 4.6 compares the payload distributed at a given time in each client for a 30MB file distribution. hMule outperforms aMule in the whole distribution by using BitTorrent protocol for transfers.

Measures of progress and transfer rates were taken every 15 seconds in every node. The stepped curve in progress accumulation is due to the progress increasing only when a full part or pieces is completed and the almost simultaneous finish of them by several clients at the same time.

For this distribution, we can see how BitTorrent is accumulating more bandwidth allocation than aMule most of the time, sometimes getting up to 75% of available bandwidth. In the other hand, aMule use is more predictable and stable in the use of the resources.

In figures 4.7 and 4.8 we compare the accumulated transfer rate between both and the payload distributed at a given time in each client, but this time, for 300MB file distributions.

When running for a larger file, we see the amount of transferred payload is equal for both clients several times until around 40% of the payload distributed. After 40% of transfer is distributed, hMule rate raises considerably as a consequence of the rarest first strategy, at that time all pieces can be consumed by any

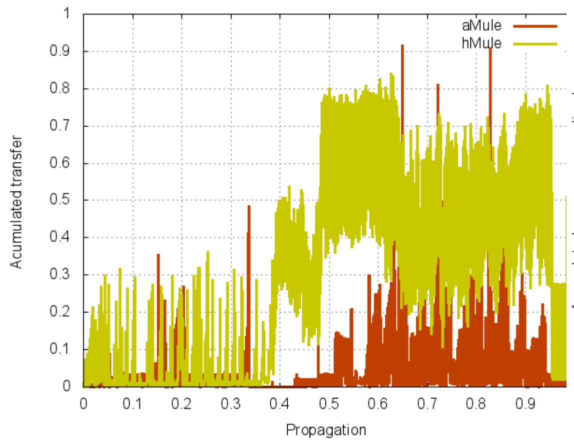


Fig. 4.7: Accumulated download ratio normalized for 300MB files

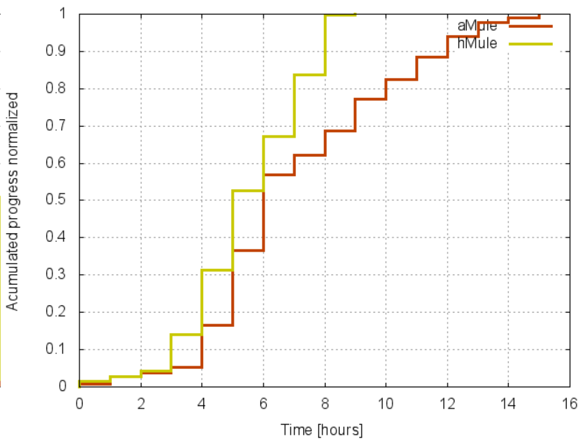


Fig. 4.8: Accumulated download progress normalized for 300MB files

other peer needing it and none is waiting for specific pieces to get available. In the other hand, aMule's queue system doesn't increase the availability of pieces needed by others, so those pieces are not populated.

This creates a competition between aMule clients for the pieces needed, while there is a lot of pieces already fully propagated. So, bandwidth use in aMule still increases after some time, but it is only based on quantity of nodes with parts required after they obtain them.

4.7.4 Evaluation of wasted payload transfer before switching protocols

4.7.4.1 Info-file exchange is provided

In *Mule networks there is a set of hashes, called AICH, to early detect when a file is not going to match the expected content. A similar set of hashes is produced in BitTorrent when using merkle.

When hMule uses AlwaysFallToTorrentStrategy, it starts downloading using ED2K until it recognizes at least one peer using hMule providing the content and an info-file is obtained. After hMule gets the info-file, it drops the downloaded data using ED2K and starts using BitTorrent protocol for transfer. We wanted to evaluate, if given this scenario, it is useful to try to recover any data already downloaded by ED2K protocol using *merkle* for torrent validation of the recovered data. The transfer of the merkle torrent metadata and the effort of hashing for validation should be justified by the quantity of data downloaded.

As we said before, 1KB files don't switch protocol so they are not considered in this analysis. When transferring 1MB files, up to 3 nodes don't switch protocol, 50% to 70% of them switched before 20KBs were transfered and, in some runs, all other nodes transfered something between 20KBs and 110KBs before switching. Since the algorithm we use to switch protocols doesn't take in account the size of the file at all, there was no different results for transfers of 30MBs or 300MBs, we got some outliners but never over 200KBs. These results for transfers that are already published in both protocols at the start of the transfer suggest the reuse of ED2K downloaded data isn't worth the effort. Anyway, in a different strategy it should be considered the use of AICH and merkle should avoid a lot of unnecessary retransfer of data, if both protocols are allowed to switch between themselves back and forth given a metric on the availability of sources in each network.

4.7.4.2 Info-file exchange is not provided

When using large files, over 100MBs, we noticed that the time taken for the lead node to generate the info-file and its associated info-hash at the beginning of transfer takes several seconds. When launching all nodes simultaneously the client publishes in Kad as soon as possible even before the info-hash completes its creation. This behavior made possible for some nodes to ask for the file before it was available for transfer using BitTorrent since we included the info-hash as extension to the negotiation phase when registering in ED2K queue.

The easiest path to take is just to delay Kad publishing until the info-file is fully processed. However, there is a special scenario where this behavior is recognized by other peers as unfair use of the network resources and we can get banned from Kad. All nodes requesting for a file should be listed as nodes providing the file, then if you start downloading a file using hMule from an aMule source you should be published as a source of the file, but until at least one node using hMule gets a complete download of the file it can't even start to process the info-file.

Therefore, we evaluated three approaches to avoid this issue in the case we should keep publishing in Kad as soon as possible, but allowing other peers to start using BitTorrent protocol as soon as possible too.

Firstly, we thought to add a TAG telling the file will be available for transfer using BitTorrent protocol in the Kad publishing, but we dropped this approach after validating that most client implementations don't republish any TAG they don't recognize as a defense mechanism, so while using this approach we lost the special TAGs in several experiment runs.

Secondly, we thought about re-asking all nodes after a few minutes based on the reported size of the file, given that we get the file size in the ED2K metadata at the time we join the queue. Asking several times the same node for metadata when being already registered in the queue can be interpreted as a denial of service attack by some implementations of the original client and we want to avoid being banned from the network at all costs.

Thirdly, the option we finally suggest is to create a new service in the client where a node can register as waiting for the info-file and when the metadata is fully processed it notifies the change to every node waiting. This request is called only on those nodes identified as hMule clients, this service can be hooked to the actual request for metada service.

The implementation for the changes proposed is not immediate at all. For our research we just adjusted our experiment to avoid the problem. We knew in our experiment which nodes were supposed to get the info-file. If they didn't get a reply with it when asking from at least 1 node, we forced a *relook* into Kad and *reask* for metadata of those nodes. We also had to check this behavior didn't blacklist our own nodes from each other which never happened because nodes never asked for metadata more than once.

4.8 Experimentation conclusions

The experimentation workbench that we used was not flexible enough to handle traffic shaping properly. Anyway, we could conduct several experiments to validate the functionalities implemented really work.

We had a real small set of peers to experiment which showed improvements in all the scenarios for hMule. Reproduction of same set of experiments in a massive quantity of nodes may not be possible until the client gets massively adopted, but simulation or emulation tactics may help to get performance estimations in those scenarios.

The introduction of recovery mechanisms based in merkle or AICH may not be useful for small files. In the other hand, in the case of large files which had transferred several parts using ED2K should be implemented in next version to avoid retransfer of considerable payload.

5. CONCLUSIONS

Based on the results of the experimentation, we can conclude our developed client is stable and can be used for the evaluation of strategies. The client works fine not only with several others hMule clients, but also when others peers involved in the transfer were aMule clients showing retro-compatibility.

Also, the results obtained in the experimentation show that hMule outperforms aMule when distributing large files, even using a simple strategy as AlwaysFallToTorrent. When using small files the difference was not so significant.

PlanetLab limitations prevented us to measure how the competition for bandwidth when both transfer protocols are active can impact in global performance. We think this is an important measure that needs to be taken using nodes with common users specifications and the ability to apply quality of service on it.

Given that we only had access to a few tens of nodes while real contents distributed using P2P have as many as tens of thousands of nodes, we suggest to use the taken measures as reference for the implementation of a simulator to determine how implementation could impact Kad if massively adopted.

Analysis of wasted transfer payload because protocol switching suggests that using merkle and AICH is not necessary for small files (less than 1MB of content when using a CableModem connection). This is faster to keep downloading in ED2K protocol if close to finish or just redownload the whole file using BitTorrent if download rate is too low. In the other hand, when the wasted transfer payload is large because there was no hMule sources providing info-file of the content earlier in the transfer, the merkle validation of the content should be implemented to avoid unnecessary retransfers.

Finally, to produce an implementation ready for final users, we suggest two strategies: First one should introduce the ability to switch back to ED2K protocol for transfer when not getting enough sources or download rate from BitTorrent protocol, the threshold of sources or download rate may be set by user configuration or an algorithm can be developed. Second one should use both transfer protocols together, initially using BitTorrent protocol, but when there is still unused bandwidth it should try to connect to standard eMule-compatible clients using ED2K. Implementation of the second strategy needs a lot of coordination between both protocols to avoid overlap efforts and minimize the waste introduced by the differences in alignment of pieces and parts.

6. FUTURE WORK

6.1 Simulation of massive adoption

For evaluation of use in higher swarms a DEvS model should be written and validated. Once the model is validated, we should be able to evaluate the advantages of adoption at great scale of the client.

`http://thepiratebay.se/` is known as the most popular torrent provider. We were monitoring the PirateBay using a script that checked PirateBay website every hour to see which was the most popular torrent and how many seeds it had, we found out that the most popular torrents in pirate bay use to have around 30000 seeds.

Since we don't have the resources to get a cluster of 30000+ nodes around the world to deploy hMule to get real world results, we looked up for some alternatives. An interesting alternative is the use of simulation that interacts with real clients for model, and tools for emulation of nodes, which can be provided using CD++.

6.2 Hybrid download using sparse files

One of the main problems to switch protocol in a download is to keep the downloaded data, aMule and LibTorrent-rasterbar can use "sparse file" for saving. When using "sparse file", before starting the download the client reserves the whole space of the content and starts writing everything in the position it will have when the download completes. Using the "sparse file" is reasonable to switch protocols without problems, it will only require a full recheck of the pieces hash at the start to know what the other protocol has already downloaded.

There are a few issues involving that. Both libraries use different ways to open the file so they should be unified or the file should be split in 2 or more pieces so it can be opened separately, the point of split should be wisely selected. The second issue will be to avoid overlap of work, this includes the creation of a map of pieces overlapping, since pieces in torrent and ED2K have different sizes and will probably not even share common divisors.

6.3 Multi-file multi-protocol content support and collections

Torrent metadata of a content can include several files and even directories, the most popular content downloaded using BitTorrent clients got more than one file. On the other hand eMule provides a kind of file name .emulecollection containing a list of MD4 hashes of a set of files. After an .emulecollection file is downloaded the client will read the file and add every listed item to the download queue. It is difficult to match the 2 approaches (specially because the eMule approach doesn't have concept of directory structure).

The proposed approach for this issue will be to read the shared files directory. If a directory is there, it will be shared in torrent as a directory structure in a single torrent file and in ED2K as a eMule-collection. If the file is downloaded from BitTorrent, an eMule-collection file will be defined where the full directory structure is downloaded and if downloaded from an eMule-collection a directory will be created with all the files to be shared in BitTorrent.

6.4 Handle the fake SHA1 detection error

Since there is no direct conversion between the ED2K identifier and the info-hash of a file we trust the user to provide the translation. If a user fakes the info-hash, we will download content that doesn't match the original indexed content, but still matches the info-hash. If after we finish a download we fail to validate the content's MD4 hash using the original Kad identifier, we throw an error. In a future version this error should be handled to retry download and blacklist the info-hash or the peer who provided it to avoid repeating the mistake.

6.5 Fake SHA1 earlier detection

We should be able to find some pieces validated using merkle that when put together have the full content of an ED2K part. If the ED2K part can not be validated using AICH it means we are downloading content that matches the provided info-hash but doesn't match the ED2K identifier and we conclude that the provided info-hash is fake and the content download should be restarted using a new info-file.

6.6 Torrent Fast Resume support

LibTorrent-rasterbar provides some tools to skip rehashing and management overheads that should be used.

6.7 Thread safeness

Before creating more complex strategies the operations that would be used for them should be checked for need of locks, specially iteration operations, because most of the code only locks at the start and finish of the method, and that is not enough in some cases.

APPENDIX

1. Installation guide

The procedure to install hMule is pretty straight foward in any Linux or BSD system[1].

Step by step install for Ubuntu 11.04 systems:

- Using synaptics, aptitude or apt-get install the following packages.
  - libgtk2.0-dev
  - libssl-dev
  - autoconf
  - libtool
  - libcrypto++-dev
  - autopoint
  - bison
- Download wxWidgets 2.9.2 from `http://www.wxwidgets.org` and unpack it.
- Open a terminal, step into wxWidget's code directory and run the following commands.

```
./configure --enable-debug --enable-unicode \
        --disable-shared --prefix=/usr
make
make install
```

- Download Boost 1.47+ from `http://www.boost.org` and unpack it.
- Open a terminal, step into Boost's code directory and run the following commands.

```
./bootstrap.sh --with-libraries=system,filesystem --prefix=/usr
./b2
./b2 install
```

- Download libtorrent-rasterbar 0.16 from `http://www.rasterbar.com/products/libtorrent/` and unpack it.
- Open a terminal, step into libtorrent-raterbar's code directory and run the following commands, if you are trying to find a bug there are options in configure that provide logging of every operation done by the library that can be really useful.

```
./configure --enable-debug --prefix=/usr
make
make install
```

- Download hMule and unpack it.
- Open a terminal, step into hMule's code directory and run the following commands.

```
./autogen.sh
./configure --enable-torrent
make
make install
```

- There is interesting options that could be added to the configure command in hmule.

[1] The software is suppose to build an run in windows also, but was not tested by us in those platforms

  - –enable-amule-daemon builds amuled, a deamon to run aMule that can be controlled remotely with amulecmd or amule-gui (best option to use in PlanetLab).
  - –enable-amulecmd builds amulecmd, a command line interface to control an amuled.
  - –enable-amule-gui builds amule-gui, a gtk GUI for amuled.
  - –disable-monolithic prevents the building of the full monolithic client (useful to add when using any of the previous options).

2. User guide

This guide doesn't intend to fully cover hMule usage, but to introduce the user to the new functionallities provided by this implementation, this guide can be complemented by the original user guide of aMule provided in `http://wiki.amule.org`.

2.1. New configuration parameters

In aMule configuration file, usually in ∼/.aMule/amule.conf, we defined a new category called Torrent which has 2 configuration options.

- TorrentDir that defines a directory where the info-files will be saved.
- Strategy that defines the strategy used to decide which protocol to use. So far we got NoTorrentStrategy (using Strategy=0) or AlwaysFallToTorrent (using Strategy=1).

2.2. Connection to Mainline DHT

To connect to or disconnect from Mainline DHT two new commands are provided in aMuleCmd.

- connect mainline
- disconnect mainline

2.3. New files

All files being downloaded using BitTorrent protocol will be stored in Temp directory with the name provided by the info-file until fully transfered. When the transfer completes, the file will be moved to Incoming directory. All metadata obtained or generated for BitTorrent transfers will be saved into Torrent directory.

Two new files are recorded in the configuration directory, MTBT.dat contains the MuleToTorrentMap persistance in a plain text format and lt-state.dat contains the state of the session dump benconded as described in libtorrent-rasterbar's documentation [2].

3. Known Issues

- In some systems, localization fails and crashes when openning amuled. To disable localization, export LC_ALL=C before running amuled and you can skip the error at the price of using non localized version.

[2] `http://www.rasterbar.com/products/libtorrent/manual.html#load-state-save-state`

- Sometimes, when trying to find updates an http error appears and crashes the app. Starting the app again should be enough to overcome the problem. The update checking tries once every few days when starting the app, so nothing breaks if the app is not stopped.

- When shutting down the application sometimes, the app tries to remove some objects more than once creating a core dump, so it doesn't affect the next run.

More known bugs can be found in the `http://bugs.amule.org/`, but only these show up in the experimentation we have done.

4. Most relevant new classes documentation

4.1. torrent::CTorrent Class Reference

`#include <Torrent.h>`

Public Member Functions

- void StartTorrentSession ()
- void EndTorrentSession ()
- const int GetPort () const
- void RefreshMetadata ()
- void CreateMetadataForFile (const CMD4Hash fileID, const CPath &filename, const CPath &storeDir)
- void CreateMetadataForFile (const CMD4Hash fileId, boost::filesystem::path filename, boost::filesystem::path storeDir)
- bool HasBTMetadata (const CMD4Hash fileId) const
- std::string GetBTIHAsString (const CMD4Hash fileId)
- void AddDownloadUsingSHA1AndPeer (const CMD4Hash fileId, std::string SHA1Hash, std::string peerIP, int port)
- bool AddDownloadUsingTorrentFile (boost::filesystem::path file)
- bool AddDownloadFromMagnet (std::string magnet)
- void Process ()
- uint64 GetCompletedSize (CMD4Hash fileId)
- void StartMainline ()
- void StopMainline ()
- bool IsMainlineConnected ()
- void LoadUnregisteredTorrents ()
- boost::filesystem::path SaveTorrent (libtorrent::create_torrent &t, boost::filesystem::path &filename)
- void GiveUp (CMD4Hash)
- virtual ∼CTorrent ()

Static Public Member Functions

- static CTorrent & GetInstance ()

  constant identifying the SwitchToTheMostUsablePeersStrategy.

Static Public Attributes

- static const int **NO_BT** = 0
- static const int ALLWAYS_FALL_TO_BT = 1

  constant indetifying the NoBtStrategy.

- static const int SWITCH_TO_THE_MOST_USABLE_PEERS = 2

  constant identifying the AlwaysFallToBTStrategy.

4.1.1. Detailed Description

Wrap class for torrent functionalities.

It is implemented as a singleton, to get the instance use: Torrent::getInstance() method.

4.1.2. Constructor & Destructor Documentation

4.1.2..1 virtual torrent::CTorrent::∼CTorrent ( ) `[virtual]`

Destructor

4.1.3. Member Function Documentation

4.1.3..1 bool torrent::CTorrent::AddDownloadFromMagnet (std::string *magnet*)

Add a download to Torrent queue having a torrent magnet link.

Parameters

magnet	A magnet link for torrent.

Returns

true if the file was registered successfully in tmm.

4.1.3..2 void torrent::CTorrent::AddDownloadUsingSHA1AndPeer (const CMD4Hash *fileId,* std::string *SHA1Hash,* std::string *peerIP,* int *port*)

Add a download to Torrent queue knowing SHA1 and a peer.

If the file is not in the queue, it is added and the peer is associated hoping to get a full metadata exchange from it, else the peer is added to the set of known peers for that file.

Parameters

fileId	An MD4 identifier for a file known in aMule.
SHA1Hash	The SHA1 provided by Kad that identifies the file's torrent metadata.

4.1.3..3 bool torrent::CTorrent::AddDownloadUsingTorrentFile (boost::filesystem::path *file*)

Add a download to Torrent queue having a .torrent file.

The file is copied into the torrent files directory and an unregistered torrents call is made.

Parameters

file	The complete filename for a info-data torrent file.

Returns

true if the file was registered successfully in tmm.

4.1.3..4 void torrent::CTorrent::CreateMetadataForFile (const CMD4Hash *fileID,* const CPath & *filename,* const CPath & *storeDir*)

Creates torrent Metadata for a file.

Parameters

fileId	The MD4 aMule identifier of the file.
filename	The name of the file.
storeDir	Path where the file is stored

4.1.3..5 void torrent::CTorrent::CreateMetadataForFile (const CMD4Hash *fileId,* boost::filesystem::path *filename,* boost::filesystem::path *storeDir*)

Creates torrent Metadata for a file.

Parameters

fileId	The MD4 aMule identifier of the file.
filename	The name of the file.
storeDir	Path where the file is stored

4.1.3..6 void torrent::CTorrent::EndTorrentSession ( )

Closes all torrent conections and destroy session object.

4.1.3..7 std::string torrent::CTorrent::GetBTIHAsString (const CMD4Hash *fileId*)

Obtain the SHA1 content identifier for torrent knowing its muleId.

Parameters

fileId	An MD4 identifier for a file known in aMule.

Returns

SHA1 torrent content identifier.

4.1.3..8 uint64 torrent::CTorrent::GetCompletedSize (CMD4Hash *fileId*)

Given a file get an estimation of how much of it was downloaded.

Parameters

fileId	The MD4 hash identification of a file in the aMule Download Queue.

4.1.3..9 static CTorrent& torrent::CTorrent::GetInstance ( ) `[static]`

constant identifying the SwitchToTheMostUsablePeersStrategy.

Get the instance.

Returns

the single CTorrent instance.

4.1.3..10 const int torrent::CTorrent::GetPort ( ) const

Port for the torrent incomming connections.

This method is used to bootstrap Torrent traffic with known Kad peers.

4.1.3..11 void torrent::CTorrent::GiveUp (CMD4Hash)

If downloading in bt this file, just give up.

Parameters

fileId	A MD4 aMule identifier of a file.

4.1.3..12 bool torrent::CTorrent::HasBTMetadata (const CMD4Hash *fileId*) const

Checks if BT Metadata is known for a file identified with a MD4 aMule identifier.

Parameters

fileId	A MD4 aMule identifier of a file.

4.1.3..13 bool torrent::CTorrent::IsMainlineConnected ( )

Check if Mainline DHT service is runnning.

Returns

true if the Mainline DHt is running.

4.1.3..14 void torrent::CTorrent::LoadUnregisteredTorrents ( )

Checks the ThePrefs::TorrentDir for info-files that are known yet.

This method iterates the dir, and queue all unknown torrents into actual session for download. All Torrents in the Torrent metadata directory are loaded, it works as a Drop box for Torrents.

4.1.3..15 void torrent::CTorrent::Process ( )

Process decides the way content should be downloaded.

This method is suppossed to be called in interval ticks as same as CDownloadQueue::Process It reads Torrent and aMule queues and decides what to start, pause, stop in each of them based in the selected TorrentStrategy.

4.1.3..16 void torrent::CTorrent::RefreshMetadata ( )

Iterates the TorrentMuleMap to load/save metadata.

Loads all the torrents info-files that are known but not loaded in session yet and saves all the torrent info-files of those that were received but not persisted yet.

4.1.3..17 boost::filesystem::path torrent::CTorrent::SaveTorrent (libtorrent::create_torrent & *t*, boost::filesystem::path & *filename*)

Saves torrent metadata into file.

Parameters

t	create_torrent instance of the file to be persisted.
filename	The filename of the content.

Returns

filename of the saved info-file torrent metadata.

4.1.3..18 void torrent::CTorrent::StartMainline ( )

Starts Mainline DHT service.

4.1.3..19 void torrent::CTorrent::StartTorrentSession ( )

Starts the torrent session.

Torrent sessions are a container for all the shared and downloading torrents and handle all the connections some ports are opened when starting the session and if prefered Mainline DHT is joined too.

4.1.3..20 void torrent::CTorrent::StopMainline ( )

Stops Mainline DHT service.

The documentation for this class was generated from the following file:

- Torrent.h

4.2. torrent::MD4ToHash Struct Reference

```
#include <TorrentMuleMapping.h>
```

Public Member Functions

- std::size_t **operator()** (CMD4Hash const &v) const

4.2.1. Detailed Description

Hash function for MuleIdToMetadataRelation indexing

The documentation for this struct was generated from the following file:

- TorrentMuleMapping.h

4.3. torrent::SHA1ToHash Struct Reference

```
#include <TorrentMuleMapping.h>
```

Public Member Functions

- std::size_t **operator()** (libtorrent::sha1_hash const &v) const

4.3.1. Detailed Description

Hash function for BTIdToMetadataRelation indexing

The documentation for this struct was generated from the following file:

- TorrentMuleMapping.h

4.4. torrent::filenameToHash Struct Reference

```
#include <TorrentMuleMapping.h>
```

Public Member Functions

- std::size_t **operator()** (boost::filesystem::path const &v) const

4.4.1. Detailed Description

Hash function for InfoFileToMetadataRelation indexing

The documentation for this struct was generated from the following file:

- TorrentMuleMapping.h

4.5. torrent::CTorrentMuleMapping Class Reference

```
#include <TorrentMuleMapping.h>
```

Public Types

- typedef std::vector< MetadataRelation ∗ >::const_iterator const_iterator

Public Member Functions

- MetadataRelation $*$ UpdateMetadata (CMD4Hash muleId, libtorrent::sha1_hash torrentId, boost::filesystem::path torrentFile)
- MetadataRelation $*$ UpdateMetadata (CMD4Hash muleId, libtorrent::sha1_hash torrentId)
- void Erase (CMD4Hash muleId)
- MetadataRelation $*$ UpdateMetadata (libtorrent::sha1_hash torrentId, boost::filesystem::path torrent-File)
- void SetDownloading (CMD4Hash muleId)
- void SetDownloading (libtorrent::sha1_hash torrentId)
- void SetSharing (CMD4Hash muleId)
- void SetSharing (libtorrent::sha1_hash torrentId)
- void SetRemoved (CMD4Hash muleId)
- void SetRemoved (libtorrent::sha1_hash torrentId)
- bool HasTorrentPath (CMD4Hash muleId) const
- bool HasTorrentPath (libtorrent::sha1_hash torrentId) const
- bool HasTorrentPath (boost::filesystem::path torrentFile) const
- bool HasBTIH (CMD4Hash muleId) const
- bool HasBTIH (libtorrent::sha1_hash torrentId) const
- bool HasBTIH (boost::filesystem::path torrentFile) const
- bool HasMuleIH (CMD4Hash muleId) const
- bool HasMuleIH (libtorrent::sha1_hash torrentId) const
- bool HasMuleIH (boost::filesystem::path torrentFile) const
- const boost::filesystem::path & GetTorrentPath (CMD4Hash muleId) const
- const boost::filesystem::path & GetTorrentPath (libtorrent::sha1_hash torrentId) const
- const libtorrent::sha1_hash & GetBTIH (CMD4Hash muleId) const
- const libtorrent::sha1_hash & GetBTIH (boost::filesystem::path torrentFile) const
- const CMD4Hash & GetMuleIH (libtorrent::sha1_hash torrentId) const
- const CMD4Hash & GetMuleIH (boost::filesystem::path torrentFile) const
- bool IsDownloading (CMD4Hash muleId)
- bool IsDownloading (libtorrent::sha1_hash torrentId)
- bool IsDownloading (boost::filesystem::path &)
- bool IsSharing (CMD4Hash muleId)
- bool IsSharing (libtorrent::sha1_hash torrentId)
- bool IsSharing (boost::filesystem::path &)
- bool WasRemoved (boost::filesystem::path &)
- const_iterator begin () const
- const_iterator end () const
- void Load (boost::filesystem::path filename)
- void Save (boost::filesystem::path filename)
- virtual ∼CTorrentMuleMapping ()

4.5.1. Detailed Description

CTorrentMuleMapping is a container for the MetadataRelations.

4.5.2. Member Typedef Documentation

4.5.2..1 typedef std::vector<MetadataRelation∗>::const_iterator torrent::CTorrentMuleMapping::const_iterator

Iterator type, only const iteration is allowed.

4.5.3. Constructor & Destructor Documentation

4.5.3..1 virtual torrent::CTorrentMuleMapping::~CTorrentMuleMapping ( ) `[virtual]`

Destructor

4.5.4. Member Function Documentation

4.5.4..1 const_iterator torrent::CTorrentMuleMapping::begin ( ) const

Standard iterator begin

4.5.4..2 const_iterator torrent::CTorrentMuleMapping::end ( ) const

Standard iterator end

4.5.4..3 void torrent::CTorrentMuleMapping::Erase (CMD4Hash *muleId*)

Erases a MetadataRelation.

Parameters

muleId	A MD4 identifier of an aMule known file.

4.5.4..4 const libtorrent::sha1_hash& torrent::CTorrentMuleMapping::GetBTIH (CMD4Hash *muleId*) const

Obtain the SHA1 content identifier for torrent knowing its muleId.

Warning

This method assumes you asking for a valid tuple, check before use.

See also

HasBTIH

Parameters

muleId	An MD4 identifier for a file known in aMule.

Returns

SHA1 torrent content identifier.

4.5.4..5 const libtorrent::sha1_hash& torrent::CTorrentMuleMapping::GetBTIH (boost::filesystem::path *torrentFile*) const

Obtain the SHA1 content identifier for torrent knowing the filename of its info-file.

Warning

This method assumes you asking for a valid tuple, check before use.

See also

HasBTIH

Parameters

torrentFile	The name of a info-file containing torrent metadata.

Returns

SHA1 torrent content identifier.

4.5.4..6 const CMD4Hash& torrent::CTorrentMuleMapping::GetMuleIH (libtorrent::sha1_hash *torrentId*) const

Obtain the MD4 aMule file identifier for torrent knowing the torrent SHA1 content identifier.

Warning

This method assumes you asking for a valid tuple, check before use.

See also

HasMuleIH

Parameters

torrentId	An SHA1 identifier for a content known in Torrent.

Returns

MD4 identifier for a file known in aMule.

4.5.4..7 const CMD4Hash& torrent::CTorrentMuleMapping::GetMuleIH (boost::filesystem::path *torrentFile*) const

Obtain the MD4 aMule file identifier for torrent knowing the filename of its torrent info-file.

Warning

This method assumes you asking for a valid tuple, check before use.

See also

HasMuleIH

Parameters

torrentFile	The name of a info-file containing torrent metadata.

Returns

MD4 identifier for a file known in aMule.

4.5.4..8 const boost::filesystem::path& torrent::CTorrentMuleMapping::GetTorrentPath (CMD4Hash *muleId*) const

Obtain the filename of a torrent info-file knowing its muleId.

Warning

This method assumes you asking for a valid tuple, check before use.

See also

HasTorrentPath

Parameters

muleId	An MD4 identifier for a file known in aMule.

Returns

filename of the torrent info-file.

4.5.4..9 const boost::filesystem::path& torrent::CTorrentMuleMapping::GetTorrentPath (libtorrent::sha1_hash *torrentId*) const

Obtain the filename of a torrent info-file knowing its torrentId.

Warning

This method assumes you asking for a valid tuple, check before use.

See also

HasTorrentPath

Parameters

torrentId	An SHA1 identifier for a content known in Torrent.

Returns

filename of the torrent info-file.

4.5.4..10 bool torrent::CTorrentMuleMapping::HasBTIH (boost::filesystem::path *torrentFile*) const

Check if the info-file has some related torrent SHA1 identifier.

Parameters

torrentFile	A filename of where the info-file metadata is saved.

Returns

true if the info-file has some related torrent SHA1 identifier.

4.5.4..11 bool torrent::CTorrentMuleMapping::HasBTIH (CMD4Hash *muleId*) const

Check if the muleId has some related torrent SHA1 identifier.

Parameters

muleId	A MD4 identifier for a file known in aMule.

Returns

true if the muleId has some related torrent SHA1 identifier.

4.5.4..12 bool torrent::CTorrentMuleMapping::HasBTIH (libtorrent::sha1_hash *torrentId*) const

Check if the BT info-hash is declared

Parameters

torrentId	A SHA1 identifier for a content known in Torrent.

Returns

true if the BT info-hash is declared.

4.5.4..13 bool torrent::CTorrentMuleMapping::HasMuleIH (CMD4Hash *muleId*) const

Check if the muleId is declared.

Parameters

muleId	A MD4 identifier for a file known in aMule.

Returns

true if the muleId is declared.

4.5.4..14 bool torrent::CTorrentMuleMapping::HasMuleIH (libtorrent::sha1_hash *torrentId*) const

Check if the info-file has some related MD4 aMule identifier.

Parameters

torrentId	A SHA1 identifier for a content known in Torrent.

Returns

true if the info-file has some related MD4 aMule identifier.

4.5.4..15 bool torrent::CTorrentMuleMapping::HasMuleIH (boost::filesystem::path *torrentFile*) const

Check if the info-file has some related MD4 aMule identifier.

Parameters

torrentFile	A filename of where the info-file metadata is saved.

Returns

true if the info-file has some related MD4 aMule identifier.

4.5.4..16 bool torrent::CTorrentMuleMapping::HasTorrentPath (CMD4Hash *muleId*) const

Check if the muleId has some related torrent metadata info-file.

Parameters

muleId	A MD4 identifier for a file known in aMule.

Returns

true if the muleId has some related torrent metadata info-file.

4.5.4..17 bool torrent::CTorrentMuleMapping::HasTorrentPath (boost::filesystem::path *torrentFile*) const

Check if the torrentFile is declared

Parameters

torrentFile	A filename of where the info-file metadata is saved.

Returns

true if the torrentFile is declared.

4.5.4..18 bool torrent::CTorrentMuleMapping::HasTorrentPath (libtorrent::sha1_hash *torrentId*) const

Check if the torrentId has some related torrent metadata info-file.

Parameters

torrentId	A SHA1 identifier for a content known in Torrent.

Returns

true if the torrentId has some related torrent metadata info-file.

4.5.4..19 bool torrent::CTorrentMuleMapping::IsDownloading (CMD4Hash *muleId*)

Check if a file is in Downloading state knowing its MD4 aMule identifier.

Parameters

muleId	An MD4 identifier for a file known in aMule.

Returns

true if the file is in state Downloading.

4.5.4..20 bool torrent::CTorrentMuleMapping::IsDownloading (boost::filesystem::path &)

Check if a content is in Downloading state knowing its torrent filename.

Parameters

torrentFile	The name of a info-file containing torrent metadata.

Returns

true if the file is in state Downloading.

4.5.4..21 bool torrent::CTorrentMuleMapping::IsDownloading (libtorrent::sha1_hash *torrentId*)

Check if a content is in Downloading state knowing its SHA1 torrent identifier.

Parameters

torrentId	An SHA1 identifier for a content known in Torrent.

Returns

true if the file is in state Downloading.

4.5.4..22 bool torrent::CTorrentMuleMapping::IsSharing (CMD4Hash *muleId*)

Check if a file is in Sharing state knowing its MD4 aMule identifier.

Parameters

muleId	An MD4 identifier for a file known in aMule.

Returns

true if the file is in state Sharing.

4.5.4..23 bool torrent::CTorrentMuleMapping::IsSharing (libtorrent::sha1_hash *torrentId*)

Check if a content is in Sharing state knowing its SHA1 torrent identifier.

Parameters

torrentId	An SHA1 identifier for a content known in Torrent.

Returns

true if the file is in state Sharing.

4.5.4..24 bool torrent::CTorrentMuleMapping::IsSharing (boost::filesystem::path &)

Check if a content is in Sharing state knowing its torrent filename.

Parameters

torrentFile	The name of a info-file containing torrent metadata.

Returns

true if the file is in state Sharing.

4.5.4..25 void torrent::CTorrentMuleMapping::Load (boost::filesystem::path *filename*)

It loads a persisted instance of the class from filename

4.5.4..26 void torrent::CTorrentMuleMapping::Save (boost::filesystem::path *filename*)

It persists actual instance into filename.

4.5.4..27 void torrent::CTorrentMuleMapping::SetDownloading (CMD4Hash *muleId*)

Set state of a relation as Downloading.

Parameters

muleId	A MD4 identifier for a file known in aMule.

4.5.4..28 void torrent::CTorrentMuleMapping::SetDownloading (libtorrent::sha1_hash *torrentId*)

Set state of a relation as Downloading.

Parameters

torrentId	A SHA1 identifier for a content known in Torrent.

4.5.4..29 void torrent::CTorrentMuleMapping::SetRemoved (CMD4Hash *muleId*)

Set state of a relation as Removed.

Parameters

muleId	A MD4 identifier for a file known in aMule.

4.5.4..30 void torrent::CTorrentMuleMapping::SetRemoved (libtorrent::sha1_hash *torrentId*)

Set state of a relation as Removed.

Parameters

torrentId	A SHA1 identifier for a content known in Torrent.

4.5.4..31 void torrent::CTorrentMuleMapping::SetSharing (CMD4Hash *muleId*)

Set state of a relation as Sharing.

Parameters

muleId	A MD4 identifier for a file known in aMule.

4.5.4..32 void torrent::CTorrentMuleMapping::SetSharing (libtorrent::sha1_hash *torrentId*)

Set state of a relation as Sharing.

Parameters

torrentId	A SHA1 identifier for a content known in Torrent.

4.5.4..33 MetadataRelation∗ torrent::CTorrentMuleMapping::UpdateMetadata (CMD4Hash *muleId,* libtorrent::sha1_hash *torrentId,* boost::filesystem::path *torrentFile*)

Updates or creates a MetadataRelation.

It looks for MetadataRelations partially containing the data provided and replaces the null data so the Metadata Relation increases knowledge, it can't remove info or modify it, only increment it. Usually this Update is used when a file starts sharing or when a download started in Kad and some peer provided the info-file metadata for Torrent transfer. (both cases content metadata is completely known).

Parameters

muleId	A MD4 identifier of an aMule known file.
torrentId	A SHA1 identifier of a torrent known content.
torrentFile	A filename of where the info-file metadata is saved to load it again in next session.

Returns

Pointer to the created or updated Metadata Relation.

4.5.4..34 MetadataRelation∗ torrent::CTorrentMuleMapping::UpdateMetadata (CMD4Hash *muleId,* libtorrent::sha1_hash *torrentId*)

Creates a MetadataRelation.

It creates a MetadataRelation with the aMule Identifier and the BT info-hash, but keeps the torrentFile as NULL. Usually this Update is used when a download started in Kad and some peer provided BTIH, but the info-file metadata for Torrent transfer was not acquired yet.

Parameters

muleId	A MD4 identifier of an aMule known file.
torrentId	A SHA1 identifier of a torrent known content.

Returns

Pointer to the created or updated Metadata Relation.

4.5.4..35 MetadataRelation∗ torrent::CTorrentMuleMapping::UpdateMetadata (libtorrent::sha1_hash *torrentId,* boost::filesystem::path *torrentFile*)

Creates a MetadataRelation.

It creates a MetadataRelation with the BT info-hash and the filename of the info-file containing torrent metadata and keeps the torrentFile as NULL. Usually this Update is used when a download started in from mainline or .torrent file and the MD4 identifier for aMule is not known yet.

Parameters

torrentId	A SHA1 identifier of a torrent known content.
torrentFile	A filename of where the info-file metadata is saved to load it again in next session.

Returns

Pointer to the created or updated Metadata Relation.

4.5.4..36 bool torrent::CTorrentMuleMapping::WasRemoved (boost::filesystem::path &)

Check if a content Was Removed knowing its torrent filename.

Parameters

torrentFile	The name of a info-file containing torrent metadata.

Returns

true if the file is in state removed.

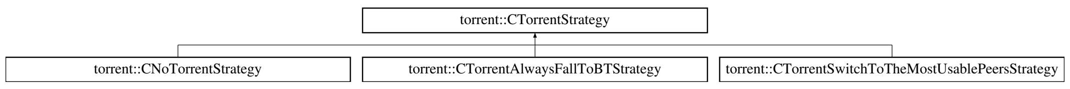
The documentation for this class was generated from the following file:

- TorrentMuleMapping.h

4.6. torrent::CTorrentStrategy Class Reference

```
#include <TorrentStrategy.h>
```

Inheritance diagram for torrent::CTorrentStrategy:


torrent::CTorrentStrategy
torrent::CNoTorrentStrategy
torrent::CTorrentAlwaysFallToBTStrategy
torrent::CTorrentSwitchToTheMostUsablePeersStrategy

Public Member Functions

- CTorrentStrategy (libtorrent::session ∗torrentSession, CTorrentMuleMapping ∗mapping)
- virtual void Process ()=0
- uint64 GetCompletedSize (CMD4Hash fileId)
- virtual void GiveUp (CMD4Hash fileId)
- virtual ∼CTorrentStrategy ()

Protected Member Functions

- void ProcessAlert (std::auto_ptr< libtorrent::alert >)
- virtual void OnReceivedMetadata (libtorrent::metadata_received_alert ∗)
- virtual void OnTorrentResumed (libtorrent::torrent_resumed_alert ∗)

- virtual void OnFinishedDownload (libtorrent::torrent_finished_alert ∗)
- void ValidateDownloads ()
- CTorrentStrategy ()

  This Vector keeps record of those hashes awaiting validation and how many tries for validation were done.

Protected Attributes

- uint32 **m_lastTimeProcessWasRun**
- uint32 m_lastTimeAlertsWereProcessed

  last time the Process method was called for this strategy.

- uint32 m_lastTimeValidationQueueProcessed

  last time the Alerts for torrent asynchronous task were processed.

- libtorrent::session ∗ m_ts

  last time the Validation Queue was processed.

- CTorrentMuleMapping ∗ m_tmm

  pointer to the active torrent session.

- std::vector< std::pair< CMD4Hash, int > > m_validationQueue

  pointer to the Metadata Relation for torrent and amule.

4.6.1. Detailed Description

Abtract class for Transfer protocol selection strategies.

Different strategies can be created to handle the way it is decided to choose transfer protocol and when to switch to the other one.

4.6.2. Constructor & Destructor Documentation

4.6.2..1 torrent::CTorrentStrategy::CTorrentStrategy (libtorrent::session ∗ *torrentSession,* CTorrentMuleMapping ∗ *mapping*)

Constructor.

Parameters

torrentSession	A pointer to the running torrent session.
mapping	A pointer to the MetadataRelations between Torrent and aMule.

4.6.2..2 virtual torrent::CTorrentStrategy::∼CTorrentStrategy ( ) `[virtual]`

Destructor

4.6.2..3 torrent::CTorrentStrategy::CTorrentStrategy ( ) `[protected]`

This Vector keeps record of those hashes awaiting validation and how many tries for validation were done. Prevent default constructor

4.6.3. Member Function Documentation

4.6.3..1 uint64 torrent::CTorrentStrategy::GetCompletedSize (CMD4Hash *fileId*)

Given a file get an estimation of how much of it was downloaded.

Parameters

fileId	The MD4 hash identification of a file in the aMule Download Queue.

4.6.3..2 virtual void torrent::CTorrentStrategy::GiveUp (CMD4Hash *fileId*) `[virtual]`

Removes any internal representation of torrents that are no longer needed.

Parameters

fileId	The MD4 hash identification of a file in the aMule Download Queue.

Reimplemented in torrent::CTorrentSwitchToTheMostUsablePeersStrategy.

4.6.3..3 virtual void torrent::CTorrentStrategy::OnFinishedDownload (libtorrent::torrent_finished_alert ∗) `[protected, virtual]`

Process the received finished download alert

If not overrided by inheritance, it will just do nothing.

4.6.3..4 virtual void torrent::CTorrentStrategy::OnReceivedMetadata (libtorrent::metadata_received_alert ∗) `[protected, virtual]`

Process the received metadata alert

If not overrided by inheritance, it will save the received metadata in a torrent file and exit.

4.6.3..5 virtual void torrent::CTorrentStrategy::OnTorrentResumed (libtorrent::torrent_resumed_alert ∗) `[protected, virtual]`

Process the received torrent resume alert

If not overrided by inheritance, it will just do nothing.

4.6.3..6 virtual void torrent::CTorrentStrategy::Process ( ) `[pure virtual]`

Process decides the way content should be downloaded.

Abstract method that should read Torrent and aMule queues and decides what to start, pause, stop in each of them, also decides when alerts from asynchronous operations in torrent should be handled and how should they be handled.

Implemented in torrent::CTorrentAlwaysFallToBTStrategy, torrent::CNoTorrentStrategy, and torrent::CTorrentSwitchToTheMostUsablePeersStrategy.

4.6.3..7 void torrent::CTorrentStrategy::ProcessAlert (std::auto_ptr< libtorrent::alert >) `[protected]`

Processes the alerts coming from asynchronous torrent tasks.

4.6.3..8 void torrent::CTorrentStrategy::ValidateDownloads ( ) `[protected]`

Check the torrents that were finished if any of them failed the MD4 check.

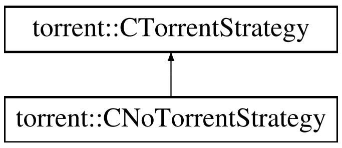
The documentation for this class was generated from the following file:

- TorrentStrategy.h

4.7. torrent::CNoTorrentStrategy Class Reference

`#include <TorrentStrategy.h>`

Inheritance diagram for torrent::CNoTorrentStrategy:



Public Member Functions

- CNoTorrentStrategy (libtorrent::session ∗torrentSession, CTorrentMuleMapping ∗mapping)
- void Process ()
- virtual ∼CNoTorrentStrategy ()

4.7.1. Detailed Description

A strategy that doesn't use Torrent Protocol for transfers, just noop and back.

4.7.2. Constructor & Destructor Documentation

4.7.2..1 torrent::CNoTorrentStrategy::CNoTorrentStrategy (libtorrent::session ∗ *torrentSession*, CTorrentMuleMapping ∗ *mapping*)

Constructor Overrides CTorrentStrategy so nothing is done.

Parameters

torrentSession	A pointer to the running torrent session.
mapping	A pointer to the MetadataRelations between Torrent and aMule.

4.7.2..2 virtual torrent::CNoTorrentStrategy::∼CNoTorrentStrategy ( ) `[virtual]`

Destructor

4.7.3. Member Function Documentation

4.7.3..1 void torrent::CNoTorrentStrategy::Process ( ) `[virtual]`

Does nothing.

See also

CTorrentStrategy::Process and CTorrentStrategy::ProcessAlerts

Implements torrent::CTorrentStrategy.

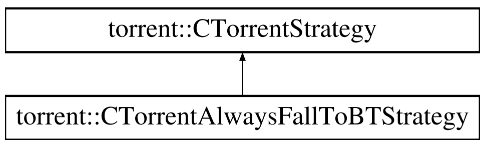
The documentation for this class was generated from the following file:

- TorrentStrategy.h

4.8. torrent::CTorrentAlwaysFallToBTStrategy Class Reference

`#include <TorrentStrategy.h>`

Inheritance diagram for torrent::CTorrentAlwaysFallToBTStrategy:


torrent::CTorrentStrategy
torrent::CTorrentAlwaysFallToBTStrategy

Public Member Functions

- CTorrentAlwaysFallToBTStrategy (libtorrent::session ∗torrentSession, CTorrentMuleMapping ∗mapping)
- void Process ()
- virtual ∼CTorrentAlwaysFallToBTStrategy ()

4.8.1. Detailed Description

A strategy that Falls to Torrent transfer protocol as soon as BTIH is known

This strategy checks if any peer provided a Info Hash for Torrent transfer and as soon as it is transfered pauses the download in the aMule download queue and fully transfer the content using BitTorrent transfer protocol. It is not a very smart strategy but is great for debugging. Check CTorrentStrategy comments.

4.8.2. Constructor & Destructor Documentation

4.8.2..1 torrent::CTorrentAlwaysFallToBTStrategy::CTorrentAlwaysFallToBTStrategy (libtorrent::session ∗ *torrentSession*, CTorrentMuleMapping ∗ *mapping*)

Constructor: Same as CTorrentStrategy constructor so far

Parameters

torrentSession	A pointer to the running torrent session.
mapping	A pointer to the MetadataRelations between Torrent and aMule.

4.8.2..2 virtual torrent::CTorrentAlwaysFallToBTStrategy::∼CTorrentAlwaysFallToBTStrategy ( ) `[virtual]`

Destructor

4.8.3. Member Function Documentation

4.8.3..1 void torrent::CTorrentAlwaysFallToBTStrategy::Process ( ) `[virtual]`

When new BT info has was received moves the download to Torrent Protocol and start ProcessAlerts

See also

CTorrentStrategy::Process and CTorrentStrategy::ProcessAlerts

Implements torrent::CTorrentStrategy.

The documentation for this class was generated from the following file:

- TorrentStrategy.h

BIBLIOGRAPHY

[1] Metro-goldwyn-mayer v. grokster ltd., 2004.

[2] Metro-goldwyn-mayer studios inc. v. grokster, ltd., 2005.

[3] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[4] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72. Citeseer, 2003.

[5] A. Ghodsi. *Distributed k-ary system: Algorithms for distributed hash tables*. PhD thesis, KTH-Royal Institute of Technology, 2006.

[6] Y. Kulbak and D. Bickson. The emule protocol specification. *eMule project, http://sourceforge. net*, 2009.

[7] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems*, pages 53–65, 2002.

[8] Gordon Mohr. Magnet v0.1, June 2002.

[9] D. Mysicka and R. Wattenhofer. Reverse engineering of emule-an analysis of the implementation of kademlia in emule, 2006.

[10] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[11] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101–102. IEEE, 2001.

[12] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[13] J.P. Timpanaro, T. Cholez, I. Chrisment, and O. Festor. Bittorrent's mainline dht security assessment. In *New Technologies, Mobility and Security (NTMS), 2011 4th IFIP International Conference on*, pages 1–5. IEEE.

[14] Juan Pablo Timpanaro, Thibault Cholez, Isabelle Chrisment, and Olivier Festor. When kad meets bittorrent - building a stronger p2p network. In *IPDPS Workshops*, pages 1635–1642. IEEE, 2011.

[15] D. Vicino, J. Pablo Timpanaro, I. Chrisment, O. Festor, et al. hmule: an unified kad-bittorrent file-sharing application. Technical report, INRIA.

[16] K. Wehrle, S. Götz, and S. Rieche. 7. distributed hash tables. *Peer-to-Peer systems and applications*, pages 79–93, 2005.

[17] B.Y. Zhao, J. Kubiatowicz, A.D. Joseph, et al. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. 2001.

LIST OF TERMS

aMule A modular opensource multiplatform alternative to eMule. iii, 7, 8, 11, 15–17, 19, 21–29, 31, 33, 38

BitTorrent A fast protocol for P2P content distribution. iii, vii, viii, xi, 1–3, 5, 6, 8–19, 21, 23–29, 31, 33, 38

eMule Nowadays, the most popullar P2P client implementing ED2K protocol and Kad. i, iii, viii, 1–3, 5–7, 9, 11, 13–17, 19, 31, 33

hMule Our prototype for hybrid client study. viii, xi, xiii, 3, 11–13, 15–19, 21–29, 31, 33, 37, 38

info-file A file containing all the metadata needed to transfer contents using BitTorrent protocol. viii, 5, 6, 8, 9, 11–19, 21, 25, 26, 28, 29, 31, 34, 38

info-hash A SHA1 hash of the info-file commonly used as DHTs key. 6, 8, 9, 11, 13–19, 21, 29, 34

Kad An indexing engine based in Kademlia specifications. i, iii, vii, viii, 1, 3, 5–7, 9–14, 16–19, 21, 23–25, 29, 31, 34

Kademlia A DHT based in XOR metrics. i, iii, 2, 3, 5–10

Mainline An implementation of Kademlia specifications for use in BitTorrent . ix, 6, 8, 9, 11, 13, 15, 19, 38

PlanetLab A global research network that supports R+D. 21–25, 31, 38

Vuze A very popular implementation of BitTorrent client which implements its own DHT for search sources. 6, 8, 9, 11, 14

LIST OF ACRONYMS

C/S Client/Server. 1, 5

DHT Distributed Hash Table. i, iii, vii, ix, xi, 2, 3, 5–15, 19, 38

ED2K eDonkey 2000. i, iii, 3, 6, 7, 9–11, 13, 14, 16–18, 21, 25, 26, 28, 29, 31, 33, 34

LARTC Linux Advanced Routing & Traffic Control = LARTC. 22

P2P Peer to peer. i, iii, vii, 1, 2, 5–7, 10, 31