Java Bittorrent API

By Baptiste Dubuis

Supervisor: Dr. Martin Rajman
Assistant: David Portabella Clotet

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, February 19, 2007

Table of contents

1 Introduction

1.1 Peer-To-Peer networks

Peer-to-peer networks (referred as P2P in the next chapters) allow clients and providers to share files in a decentralized environment. There are several reasons for P2P networks to exist. First one may be to ensure high file availability. In the classic client/server approach, a file is hosted by a server. But if the server goes down (crash, maintenance…), there is no possibility for a client to get the file. With P2P, the file is made available by every client that shares it and there should not be any centralized server that proposes the file. A client who wants to download a file will contact other peers sharing the same file and download directly from them, which makes the files highly available, especially very popular ones. If a peer stops serving the files, there should always be other peers that still have them and share them.

A second reason is for sure the bandwidth cost. If a single server was proposing very popular services and files to download, it will have tons of clients connecting to it every second and so it will require a huge amount of bandwidth to be able to serve them at the same time. It is in this case that P2P networks appear to be the ultimate solution. Indeed every people downloading the file will share it at the same time, downloading not only (and maybe not at all) from the main server but from other peers that also share the files. Then the initial host of the files will only have to share it for a few times and without huge bandwidth requirement.

We can see easily that P2P networks provide some advantages over classical client/server approach for large files transfers. Among peer-to-peer networks the search of the most efficient algorithms to assure good throughput and availability as given birth to many different protocols. Napster, Kazaa, eMule (eDonkey2000 network) and now Bittorrent are the most famous of them.

1.2 Bittorrent

Bittorrent is one of the rising P2P transfer protocols on the internet. Many users have already adopted it and major companies in the field of entertainment (music, movies…) are investigating to use this protocol for the distribution of their products due to its popularity and the good transfer rates it provides. It uses metadata files (called torrents) to provide necessary information to its clients.

1.3 Goal of the project

The first goal of this project was to get in touch with this relatively new protocol. We wanted to know how it works and what its main characteristics are.

Further we wanted to study if this protocol could be useful for everyday's internet use, like for example web browsing and if so, what could be the best way of using it. The main purpose of this project is to see if it could be possible to modify a web server and a web browser so that it can use the Bittorrent protocol in an efficient manner, providing both bandwidth economy and increased reliability for the server by delegating the task to its clients for example.

1.4 Tools and libraries

As one will see, there is not that much implementation of the Bittorrent protocol in JAVA. It was therefore very difficult to reuse code to simplify the work. However some preexisting classes and libraries have been used throughout this project to ease the development of the software and also because of their efficiency and user-friendliness.

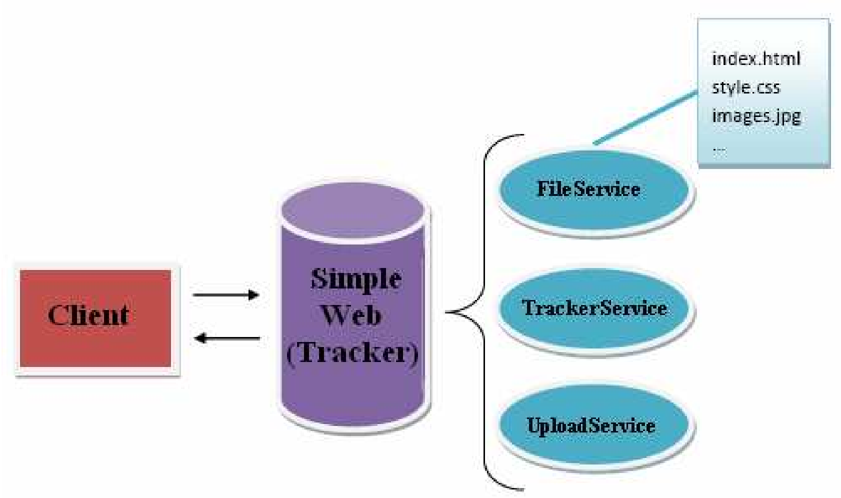

Figure 1 - Simple Web Architecture for Tracker implementation

1.4.1 Simple Web[1]

If a user wants to put a tracker online, he will need to setup a web server to host the files and respond to client request. For this reason I was proposed to use the Simple Web library. It provides the possibility to execute JAVA code (services) in order to respond to client requests which is exactly what I needed to implement the tracker proposed within the

[1] http://www.simpleframework.org

project software. In the same time, it is quite light compared to other solution like Tomcat while being complete enough to ensure a good performance.

It is very easy to create a server using Simple web library and the architecture is the following: a main class accept incoming connection on a given port and redirect them to the corresponding service according to the URL. The service is a JAVA class that is instantiated to process the request in any way and create the corresponding response. The tracker architecture is represented in Figure 1 and the services will be explained below.

1.4.2 JDOM[2]

The tracker created for this project does not use a database. Instead of that, it relies on XML files to store and retrieve torrents and peers information. In order to do that, I choose the JDOM library which provides a complete, Java-based solution for accessing, manipulating, and outputting XML data from Java code.

1.4.3 ClientHTTPRequest[3]

When communicating with trackers for uploading files, I have been confronted to a special kind of HTTP request: multipart/form-data as defined in RFC2388[4]. These are HTTP request encoded in a special manner to permit transmission on files along with text and other parameters.
The ClientHTTPRequest class implements this protocol and utility methods that permit in a simple manner to add content to the request and send it to a web server. Therefore, this class has been chosen for speculative future enhancement of the publication part of the API, which might need to communicate in a non-trivial way with trackers or web servers. In this case, the ClientHTTPRequest class should be very helpful.

1.4.4 BEncoder.java and BDecoder.java

As we will see in chapter 3.2, Bittorrent protocol uses a special encoding for the torrent files and the tracker responses called BEncoding. There are many JAVA classes that implement this encoding type. I just found the ones that are part of the Azureus[5] distribution very practical and general so that they can be used with many types of data.

[2] http://www.jdom.org
[3] See article http://www.devx.com/Java/Article/17679/0/page/3
[4] RFC2388: Returning Values from Forms: multipart/form-data. Available at http://www.ietf.org/rfc/rfc2388.txt
[5] Azureus is one of the most popular Bittorrent client and almost the only one to be written in JAVA. See http://azureus.sourceforge.net

2 Illustrative example

In order to illustrate the use of Bittorrent protocol and of our API along the next sections of this report, let's consider this simple example:

John went on holidays and he has his digital camera with him. He took about a hundred of pictures (1MB each) and made also two movies (50MB each). When he came back home, he decided to share his holiday stuff with his friends, which are located in many different countries. So he would like to propose to his many friends and also to other people the possibility to download his files.

The problem is that the total size of data is huge, 200Mo, and so sharing for example to ten people would require a transfer of 2000Mo of data (plus the overhead of the protocol used). Also he does not own a very fast Internet connection and still would like to be able to use it, even if people try to download from his computer. So he cannot simply put these files on a local website. Hopefully he recently heard about peer-to-peer networks and knows he can provide these files to other people by not using only his bandwidth, but also the one of other people downloading the files. For this, he decided to use a popular protocol and the one he found to be very used actually is the Bittorrent Protocol. He read some documentation and then follows the different steps that are required to publish his files.

2.1 Step 1: Create a torrent file

The torrent file contains essential information about the files John wants to publish and about the location he will publish it. To create such a file, John needs special software that takes as input his holidays pictures and movies and some other parameters.

Many programs permit to do it quite simply. Among them, let's quote MakeTorrent[6] or the well known Azureus, the complete JAVA Bittorrent program that also permit to create a torrent file. Taking the example of MakeTorrent, John will have to download the program on the official website. He will install it running the .exe files downloaded and launch the application. Then he will browse the directory tree to find the files he wants to publish. Once he has selected them, he can enter the tracker announce URL or choose it in the provided list. Let's suppose he wants to publish it on smartorrent tracker, which announce URL is ***http://www.smartorrent.com:2710/announce***. Finally he can enter some comment about his holiday's files and choose the length of the

[6] See http://krypt.dyndns.org:81/torrent/maketorrent/

pieces the files will be split into to be distributed. Then, he'll push the "Create .torrent now!" button and indicate the name and location this file can be saved into, for example "c:\torrents\holidaysPicture.torrent".

2.2 Step 2: Publish torrent file

Once the torrent file is created, John will have to publish it on the tracker he specified earlier in the torrent file ("announce URL" parameter). To do that, he will need to create register an account on smartorrent.com. To complete this step, he will have to:

1. Go on http://www.smartorrent.com
2. Click on the "s'inscrire" tab
3. Provide the information
    a. Nom d'utilisateur(username): johnDoe
    b. Email: john@epfl.ch
    c. Confirm email: john@epfl.ch
    d. Mot de passe (password): bittorrentAPI
    e. Confirmer mot de passé: bittorrentAPI
4. Press "Envoyer" button, this will show the login page
5. Enter his created username and password to login and press login
6. Choose the uploader tab
7. Provide the requested information about the torrent he has created
    a. Nom (name): "holidays.torrent"
    b. Torrent: "c:\torrents\holidaysPicture.torrent"
    c. Categorie: "autre"
    d. Lien de description externe: none
    e. Description: "My holidays pictures and movies"
8. Press upload, a page will announce "Votre torrent a bien été uploadé", meaning your torrent has been published correctly

2.3 Step 3: Distribute torrent

Now the torrent is published, John can advertise his friends that they can download the files from the tracker. To do that, they can simply go to http://www.smartorrent.com, register an account as John did, login and search for the torrent name John provided. This will display the list of torrents corresponding to this name. There can be more than one result, in such case they will just have to check the uploader is John (i.e. that the username is johnDoe). Then they just click on the torrent which will display a page with torrent details. From there, they just press the download link and get the torrent.

A faster way for John to provide the torrent file to his friends is simply to send them the (small) torrent file by email, so they don't have to register on smartorrent and directly get the file necessary to begin download.

2.4 Step 4: Share the pictures and movies

John now has to provide his files for download. In order to do that he has to open the torrent file he just created using a Bittorrent client. There are many of them available[7]. Since we talk about it for a moment, let's see how to use Azureus. Users can get the latest version of the program at

http://sourceforge.net/project/showfiles.php?group_id=84122

and choosing the correct distribution. Once downloaded, John will just have to run Azureus on his machine. Once it is launched, John will just have to open the torrent he wants to share and there he is: the client is now sharing the files with other peers that can now start downloading pieces of John's beautiful pictures and movies.

2.5 Step 5: Download the pictures and movies

For John's friends to be able to download the files, they have to follow the same steps as John when he had to share his files, i.e. run a Bittorrent client on their machine and open the torrent file provided by John. The client will be able to contact other peers sharing John's files and download pieces from them. Finally they will all get John's holiday's stuff on their computer.

2.6 Conclusion on the example

We have seen here that John and his friends must do quite a lot of things before everybody's got John's files. Download multiple programs, visit several web sites and so on. Although all the preceding steps could have been done using the latest version of Azureus, it seems that it would have been too difficult and furthermore too heavy to use Azureus for the main goal of the project, which is using Bittorrent in web browsing by adapting a web browser and/or a web server. For this reason, Java Bittorrent API project should be the solution for John and his friend since it provides a simple API and furthermore simple example program that permit to execute the different step presented above in a quite easy way. Moreover, the API behind these examples is complete and performing enough to use it for the next part of the project.

[7] Visit http://en.wikipedia.org/wiki/Comparison_of_BitTorrent_software for more information about Bittorrent Clients

3 Presentation of the protocols

Before using the programs he just downloaded John, who is curious by nature, decided to check what that Bittorrent protocol is. He learned lots of things…

The Bittorrent protocol needs several steps to complete any download.

First John, the owner of the original files, will have to create a relatively small metadata file which will contain all information that any other user will need to retrieve John's pictures and movies. This description file is called the torrent file, or simply the **torrent.**

Once this file has been created, John will have to **publish** it on a **tracker**, which should coordinate the file transfers. Then, John will have to run a Bittorrent client, which will be responsible to communicate with other **peers** and share with them John's holiday files, which will be called hereafter the **target files**.

When the previous steps are completed, John is able to share his files with people. In order to do that, he will have to provide people the description file. Although it should already be available on the tracker, John can upload this small file to as many **web servers** he wants, so that it is available for many people. And for his close friends and relative he can even send them the torrent by email, so that they don't have to bother searching the file on the Web.

At the time they get this metadata file, they can run a Bittorrent client as John do and therefore start downloading. At the beginning, they will find the files part on John's computer but as the time goes by they will also download parts from other people, making John's connection bandwidth less used…

Peers, tracker, web server… These terms are not that clear at the time. So let's see what they are all about.

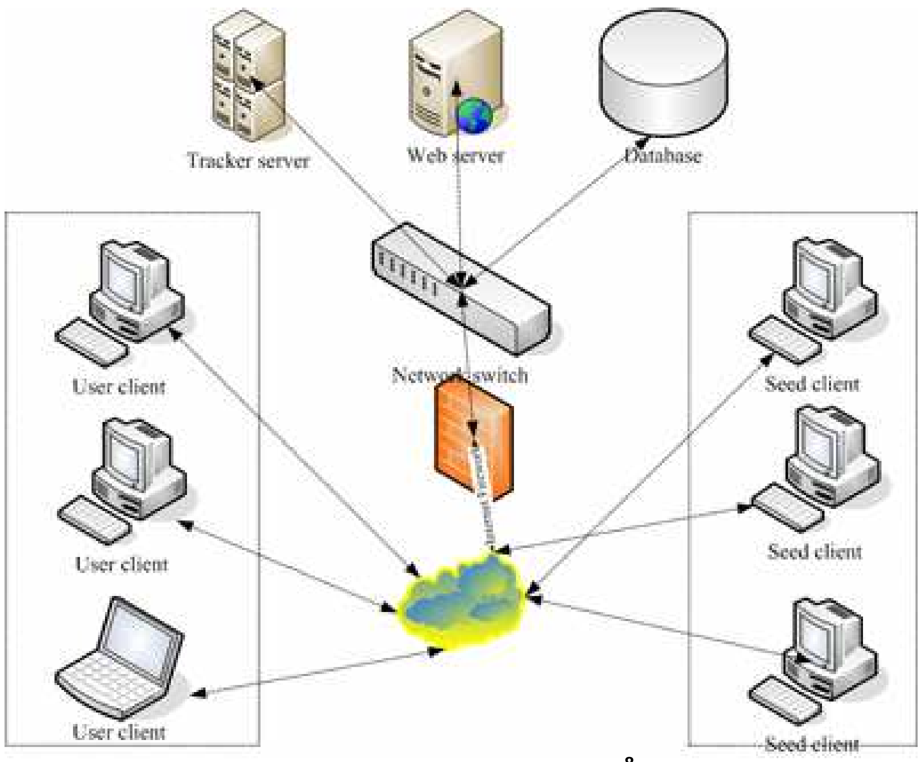

Figure 2 - Bittorrent actors[8]

3.1 Actors in Bittorrent protocol

To accomplish this task, Bittorrent protocol relies on three kinds of actors (Figure 2): (1) the *web servers* to find the appropriate torrent files, (2) the *trackers* to coordinate the peers in the network and (3) the *peers* who share the files in the network by running a Bittorrent client. Each of these actors has a precise function which is presented hereafter.

3.1.1 Peers

Peers are the end-users of Bittorrent protocol. It is among them that pieces of target files are exchanged. In our example, peers are represented by John, his friends and all people running a Bittorrent client to share some files. Among these peers, it is possible to denote 2 subcategories of peers called, in the Bittorrent language, *seeders* and *leechers*.

Seeders are the peers that own the complete target files corresponding to a torrent and share them with other peers. Equivalently, we could say that they only upload these files and do not download from other peers (Figure 3 – thin blue arrows). It is quite clear here that just after files have been

[8] Figure taken from http://www.hwysoft.com/cht/source.htm

published, John is the only seeder, since he is the only one to own the whole pictures and movies.

Leechers are all the other peers, John's friends and people on the Internet, the ones that do not own the complete file and therefore download pieces from other peers (Figure 3 – thin red arrows). They can also serve the pieces of the files that they already have. Once they'll have downloaded all parts of the files, leechers become seeders if they continue to share the files.

Each peer runs a Bittorrent client to share or download files to/from other peers. A peer *chokes* a client to inform it that no requests will be answered until the client is unchoked. Therefore, the client should not send any requests for blocks of data and consider all pending requests to be discarded.

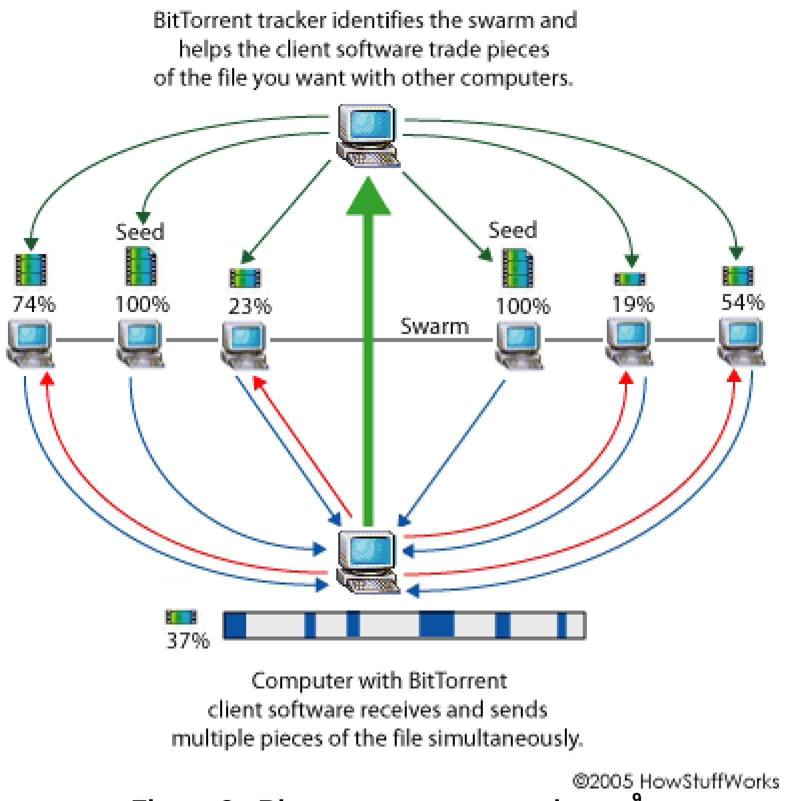
BitTorrent tracker identifies the swarm and helps the client software trade pieces of the file you want with other computers.
Seed
Seed
74%
100%
23%
100%
19%
54%
Swarm
37%
Computer with BitTorrent client software receives and sends multiple pieces of the file simultaneously.
©2005 HowStuffWorks

Figure 3 - Bittorrent message exchange[9]

A peer can be *interested* in another if the latter owns pieces of data that the former itself does not.

Each peer maintains state information for each connection with a remote peer. State information could be:

- **is_choking**: the remote peer is choking the client
- **is_interested**: the remote peer is interested in the client

[9] Figure taken from http://computer.howstuffworks.com/Bittorrent2.htm

- **is_choked**: the client is choking the remote peer
- **is_interesting**: the client is interested in the remote peer

Peers can find the IP addresses and port of other peers by contacting a tracker.

3.1.2 Trackers

In each metadata file that users will download, there will be at least one tracker URL (otherwise the torrent file is not valid…). This URL corresponds to the address of another kind of actor, the *tracker*.

The tracker is responsible of coordinating all peers that request the same target files or, equivalently, use the same torrent file. Basically, the tracker maintains a list of all peers that are currently sharing or requesting the same file, providing them the possibility to contact each other and start sharing new parts of the files.

In order to appear in the tracker’s peer list, a client will have to announce itself to the tracker according to the **Tracker HTTP/HTTPS Protocol** (Figure 3, thick green arrow).

Each torrent created with the announce URL of a given tracker has to be published on the corresponding tracker in order to be available for other peers and we say that the tracker owns the torrent. The tracker can then serve the torrent to download and reply to peer requests with list of other peers sharing the same file.

For John, this means that after having registered his description file on a given tracker, he will run his Bittorrent client to share his files. The client will contact the tracker and remember John’s computer IP address and listening port. Then, every time another client will contact the tracker for information about people sharing John’s files, the tracker will return a list of peers information, within which will be John’s.

3.1.3 Web servers

The first thing users have to do to download a file is to get the metadata file (torrent) that corresponds to the target files users want to download in the end. These files are present on the trackers on which they have been published.

But there is nothing in Bittorrent protocol that prevents any web servers to propose a torrent file to be downloaded from it… Indeed, as it has been said before, the torrent file contains all the information that a client needs to download the target files. Thus there is no need for a torrent file to reside on its tracker. It can therefore be located on any web server that let people download from it and that's an advantage of the Bittorrent protocol: torrents can be everywhere once they have been published.

To find these torrent files, users can simply use their favorite search engine in their browser, files containing both the name of the desired target files and the *.torrent* extension. This way is by far not the most practical, since it will returns tons of answers and maybe not what is really needed.

Hopefully, there are many communities that provide this search services into their own databases of torrents and let users download the torrents that correspond to the target files.
No matter the way users choose, the task is always the same: to get the metadata file, it is necessary to download it from a *web server*, either by HTTP or FTP.

It might be interesting to note that nowadays, many trackers also permit the search of torrents that they do not own and provide them either by direct download or by redirection to a site that provides it, while other web sites only list tons of torrents and redirect the users to the tracker/website that provides the file for download.

3.2 Description file: the "torrent"

Now let's see in more details what a torrent file really is. A torrent file is a text file that contains metadata necessary to a client willing to download a certain file. The encryption of this metadata is called *bencoding* and supports four types of data: byte strings, integers, lists and dictionaries.

1. **Byte strings** are encoded as follows: *<string length in base 10 ASCII>:<string data>*
   Example: *19:Bittorrent protocol* represents the string "Bittorrent protocol"
2. **Integers** are encoded as follows: ***i***<*integer in bas 10 ASCII*>***e***, where i and e are beginning and ending delimiters
   Example: ***i****1664****e*** represents the integer **1664**
3. **Lists** are encoded as follows: *l<bencoded value>…<bencoded value>e*.
   Lists contain any of the four bencoded types.
   Example: ***l****10:Bittorrent****i****1664****ee*** represents the list ["Bittorrent", 1664]
4. **Dictionaries** are encoded as follows: *d<bencoded string><bencoded element>e*
   The keys must be bencoded strings. The values may be any bencoded type, including integers, strings, lists, and other

dictionaries. Dictionaries are sorted in alphabetical order referring to the keys.
Example: ***d**7:torrent**l**3:spai5**eee*** represents the dictionary {"torrent" => ["spa", 5]}

The content of the torrent file is a bencoded dictionary containing the keys listed below.

- **announce**: The announce URL of the tracker (string)
- **announce-list**: (optional) this is an extension to the official specification, which is also backwards compatible. This key is used to implement lists of backup trackers.
- **creation date**: (optional) the creation time of the torrent, in standard UNIX epoch format (integer seconds since 1-Jan-1970 00:00:00 UTC)
- **comment**: (optional) free-form textual comments of the author (string)
- **created by**: (optional) name and version of the program used to create the .torrent (string)
- **info**: a dictionary that describes the file(s) of the torrent. There are two possible forms: one for the case of a 'single-file' torrent with no directory structure, and one for the case of a 'multi-file' torrent (see below for details). In both case, to create the info dictionary, several steps must be taken:
  - Split the target file(s) data into pieces of equal length. In case of multiple files, the data bytes are concatenated and the resulting bytes are split as a single one (Figure 4 below). Number of pieces is equal to
    ceil(total_length/piece_length)
  - For each piece, compute the 20bytes SHA-1[10] hash of the corresponding bytes. The hash uniquely identifies each piece[11] and can therefore provide the client a way to verify that the downloaded piece (i.e. part of the target file(s)) has not been corrupted since the publishing of the original files.
  - Concatenate these hashes to obtain a single string which length will be a multiple of 20 bytes, depending on the number of pieces
  - It is important to note here that:
    - All pieces have the same length (piece_length bytes) with a single possible exception for the last one that could be truncated.

[10] For further information, see http://en.wikipedia.org/wiki/SHA-1
[11] http://www.schneier.com/blog/archives/2005/02/sha1_broken.html, collisions could have been found on the SHA-1 algorithme. But this is not yet of real concern for Bittorrent. See discussion entitled [Bittorrent] SHA-1 Broken at http://lists.ibiblio.org/pipermail/Bittorrent/2005-February/thread.html#560

- A piece can contain data belonging to more than one file, depending on the length of the pieces and the length of the files (see for example *Figure 4, piece 9*)
- Piece length is commonly set to 256kB, 512kB or 1MB. The goal is to keep a rather small torrent size to permit a quick download and no waste of bandwidth for the web servers while still having an efficient swarm for sharing the files. Nowadays, the conventional wisdom is to have a rather small piece size that increases the swarm efficiency, even if the torrent becomes larger. This is because web server bandwidth is becoming less tightly constrained.

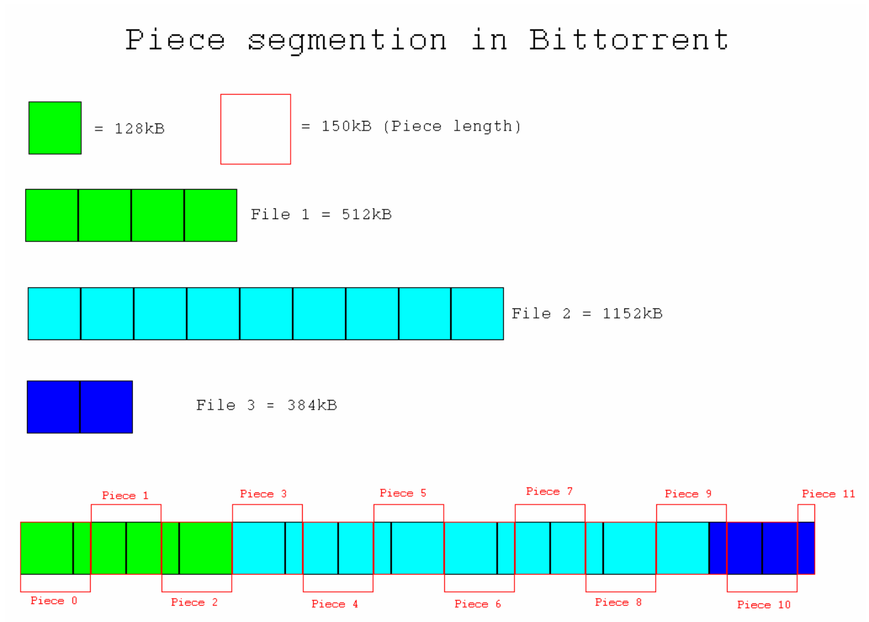

Figure 4 - Piece segmentation

The info dictionary contains the following fields, depending on whether the torrent represent single or multiple file(s).

- **piece length**: number of bytes in each piece - except possibly the last one (integer)
- **pieces**: string consisting of the concatenation of all 20-byte SHA1 hash values, one per piece. The hashes are needed to verify the integrity of the downloaded pieces (byte string)
- **name**: in the case of a single-file download, this represents the (advisory) filename of the file. In the multiple-file download case, this represents the (advisory) name of the directory in which to store the files (byte string)
- **length**: length of the file, in bytes. Present at this level only in the single file case (integer)
- **files**: only in the multiple-file case. Lists of dictionaries, one for each file. The dictionaries have the following keys:
    - **length**: length of the file in bytes (integer)

- **path**: list containing one or more string elements representing the ordered sequence of directories to save the file in. Last element is the file name.
  Example: if the file has to be saved as "dir1/dir2/file.ext", this will be encoded as: ***l****4:dir14:dir28:file.ext****e***

Several optional and non important fields also exist and could be used to perfect the torrent, but we chose to only present the most important ones.

3.3 Communication between actors

At this point we know the different actors in presence and also we have understood the central role of the torrent file in the protocol. Now we will study in more details how the actors interact with each other and especially what are the particularities of their communication protocols.

3.3.1 Torrent files distribution

We have seen that one of the main qualities of the Bittorrent protocol is that it is very easy to make torrent files available. Since they can be downloaded from any web servers, retrieving these files does not require a special protocol. This can be as simple as a HTTP response to a GET request. This can be done by FTP, or by the mean of an email. There is here no general manner to distribute these files and once again this is the strength of Bittorrent. Publishing them require the same protocol. Earlier we have mentioned the existence of communities of users that list tons of torrent files and provide them to download. These communities are particularly useful since they provide in general performing search engines which give the possibility to sort this amount of data and retrieve the exact torrent that users need. They also provide efficient trackers that can handle hundreds of thousands of users. Therefore for making

Now that the torrent file is published on a tracker and that users have access to it from web servers, it is time to start the communication between the client and the tracker. To enable the communication, peer and tracker use a special protocol that is describe hereafter.

3.3.2 Peer – Tracker communication

The tracker is an http/https service which replies to HTTP GET requests. The base URL is the tracker announce URL, specified in the torrent file. Then, parameters are encoded in this URL according to the RFC2396 specification[12] (i.e. a "?" followed by "param=value" sequences separated by "&").

[12] RFC2396: Uniform Resource Identifiers (URI): Generic Syntax. Available at http://www.ietf.org/rfc/rfc2396.txt

When a peer wants to share or download a file, it contacts the tracker using a HTTP GET request and uses the ***Tracker HTTP/HTTPS Protocol***, which requires the following parameter being encoded in the URL:

- **info_hash**: 20-byte SHA1 hash of the *value* of the *info* key from the metadata file. This identity uniquely the metadata file so that the tracker knows which target files the client wants to download
  - It is very important and interesting to note here that a torrent file is identified by its info part only. That means that if one create several torrents with a different announce URL but with the same info hash (i.e. same target files in the same order), the torrents will be the same from a Bittorrent point of view. That provides the possibility to publish the same files on different trackers, making the target files even more available since they can be shared by more peers.
- **peer_id**: 20-byte string used as a unique ID for the client, generated by the client at startup. This is allowed to be any value, and may be binary data
- **port**: The port number that the client is listening on. Ports reserved for Bittorrent are typically 6881-6889 but there is no need for a client to listen on those.
- **uploaded**: The total amount uploaded in base ten ASCII
- **downloaded**: The total amount downloaded in base ten ASCII
- **left**: The number of bytes this client still has to download, in base ten ASCII

The three field *uploaded, downloaded* and *left* may be used by the tracker to maintain some kind of statistics about the peers, for example determine the number of seeders, the download rate of each peer, …

- **compact**: Indicates the client accepts a *compact* response. Value field is set to 1 if client accepts, otherwise it is set to 0. If this field is not present, tracker considers client does not support compact response. Compact response is described below in the tracker response description.
- **event**: If specified, must be one of *started*, *completed*, *stopped*. If not specified, then this request is a request that the client performs at regular intervals to get new peers.
  - **started**: The first request to the tracker *must* include the event key with this value.
  - **stopped**: Must be sent to the tracker if the client is shutting down gracefully.
  - **completed**: Must be sent to the tracker when the download completes. However, must not be sent if the download was already 100% complete when the client started.

The tracker then replies to the request with a plain text document containing a bencoded dictionary with the following keys:

- **failure reason**: If present, then no other keys may be present. The value is a human-readable error message as to why the request failed (string).
- **interval**: Interval in seconds that the client should wait between sending regular requests to the tracker (mandatory).
- **min interval**: Minimum announce interval. If present clients must not reannounce more frequently than this.
- **tracker id**: A string that the client should send back on its next announcements. If absent and a previous announce sent a tracker id, do not discard the old value; keep using it.
- **complete**: number of peers with the entire file (integer)
- **incomplete**: number of non-seeder peers (integer)
- **peers**:
  - In case of non compact response, the value is a list of dictionaries, each with the following keys:
    - **peer id**: peer's self-selected ID, as described above for the tracker request (string)
    - **ip**: peer's IP address (either IPv6 or IPv4) or DNS name (string)
    - **port**: peer's port number (integer)
  - In case of compact response, the peers list is replaced by a peer string with 6 bytes per peer. The first four bytes are the host (in network byte order); the last two bytes are the port (again in network byte order). It should be noted that some trackers only support compact responses (for saving bandwidth) and refuse normal requests. Also note the deprecation here of the peer id, not sent.

3.3.3 Peer Wire Protocol

As soon as the client knows the IP address and port of other peers, either by receiving the list from a tracker or because a remote peer is trying to connect to it, then the client should start the communication with these remote peers. The message exchange between peers is done using the so called Peer Wire Protocol described hereafter. The protocol is symmetric, in the sense that messages can flow in both directions in the same format.

3.3.3.1 Handshake

The communication between peers starts with a symmetric handshake. The peer that wants a piece of file sends a handshake message to the peer that owns this piece. The handshake message has the form:

<pstrlen><pstr><reserved><info_hash><peer_id>

- **pstrlen**: string length of <pstr>, as a single raw byte (default 19)
- **pstr**: string identifier of the protocol (default "Bittorrent protocol")
- **reserved**: 8 reserved bytes (default 00000000)
- **info_hash**: 20-byte SHA1 hash of the info key, same as sent in tracker request
- **peer_id**: 20-byte string used as unique ID for the client sending the message, same as in tracker request, if non compact response is used

3.3.3.2 Messages

All remaining messages take the form of:

<length prefix><message ID><payload>

The length prefix is a 4-byte big-endian value.
Message ID is a single decimal character.
The combination of these two fields uniquely identifies each kind of message according to the following list:

- **Keep-alive <len=0000>**: length 0, no ID, no payload, this message must be sent to avoid certain clients to close connection after a certain period of inactivity.
- **Choke <len=0001><id=0>**: choked the client that receives this message.
- **Unchoke <len=0001><id=1>**: unchoked the client that receives this message.
- **Interested <len=0001><id=2>**: inform a client that a peer is interested in at least one of the piece it owns.
- **Not interested <len=0001><id=3>**: inform a client that a peer is not interested.
- **Have <len=0005><id=4><piece index>**: A piece of the file has just been successfully downloaded and verified. The payload of the message is the zero-based index of the piece.
- **Bitfield <len=0001+X><id=5><bitfield>**: It informs which parts of the file the client has. The payload of the message is the bit representation of the pieces that the client has. High bit in first byte corresponds to piece index 0. Cleared bits correspond to a missing piece, set bits to a valid and available piece. There can be spare bits at the end of the payload and they are set to 0. X corresponds to the bitfield length. It is sent immediately after the handshake is completed. Optional if the client has no pieces of the requested torrent.
- **Request <len=00013><id=6><index><begin><length>**: used to request a block of data in a piece

- **Index**: integer specifying the zero-based piece index
  - **Begin**: integer specifying the zero-based byte offset within the piece
  - **Length**: integer specifying the requested length (in bytes)
- **Piece <len=0009+X><id=7><index><begin><block>**: message that contains a block of data, where X is the length of the block of data
  - **Index**: integer specifying the zero-based piece index
  - **Begin**: integer specifying the zero-based byte offset within the piece
  - **Block**: block of data, which is a subset of the piece specified by index
- **Cancel <len=00013><id=8><index><begin><length>**: cancels a block request

Note that "request" messages can be sent by a client to a remote peer only if the client is *interested* (hence the client has sent an "interested" message to the remote peer) and the remote peer is *not choking* the client (hence the client has received an "unchoke" message from the peer).

4 Implementation of the protocols

There are a lot of implementation of Bittorrent protocol and, so, many Bittorrent clients. But there are only few of them that are written in Java. Actually, the only one that I found is the Azureus client. It is an amazingly complete client, with many features, plug-in and so on. And as it is very complete, it is in the same time very complex, with many interfaces and no real API available. The idea then was to create a new client, i.e. a new implementation of the protocol, as an API with maybe less features, but with a much lower complexity, a client that can be easily understandable and updatable for future work on it and therefore easily embeddable in another application.

Keeping that in mind, I decided to build brand new software that offers the possibility to ease the publication of files, to download and process metadata files (torrents) from any web servers and, according to the downloaded torrent, to start the retrieval of the files described in this torrent. During this work, I also found that it might be difficult for a user to find a tracker where he can easily publish his torrents on. So decision was taken to implement a tracker to ease the publication. We will see in the following sections how these two programs have been implemented.

4.1 Publishing files

As it has been said before, in order to download a file using Bittorrent, users need to get a metadata file, called torrent file. But if they can get it, it is also because someone made it available before… This program offers the opportunity to publish files, by creating torrents containing information about them, and publishing the torrents on a tracker to make it available for other people.

What is interesting with Bittorrent is the fact that in one torrent, one can provide several files to download and not only publish one file at a time. Compared with eDonkey network, when one wanted to publish several files, either one published them one at a time or one compressed them in one folder and then published the compressed file. This feature is interesting in the sense that a user can group files that are related to the same main theme while still leaving the possibility to a remote peer to download only the files he truly wants.

For John, this is in interesting since he can create a single torrent file describing all his pictures and movies and publish it on a tracker. In the same time any user that would want to download only some pictures or only the 2 movies will not have to download all the files but could choose to download only those he really wants.

The next few sections demonstrate how users can publish or retrieve files using our programs, trackerBT and jBittorrentAPI.

4.1.1 Create torrent file

The first thing users have to do to publish files is to create a torrent file containing information about the target files. The torrent creation is mainly handled by the class TorrentProcessor and some of its methods.

Basically what this class do is to take input arguments such as the announce URL of the tracker, the piece size, a list of the files to be published and comment about them and also the name and path of the file the created torrent should be saved into. These parameters are passed to an instance of TorrentProcessor by the mean of ***set*** methods (setTrackerURL, setPieceLength …) and files are added thanks to the add methods (addFile or addFiles).

Once all parameters have been set and all files that should be published have been added, the object can generate the SHA-1 hashes of the pieces of the target files by calling the method generatePieceHashes. At this time the object contains a TorrentFile object that is my JAVA representation of a "physical" torrent file. The last steps to generate this file are to call the generateTorrent method that compute, given the TorrentFile, the bytes corresponding to the BEncoded torrent file. Finally the file is saved according to the directory and filename provided above.

In order to show how this methods can be used and in a thought of providing users an intuitive way to create torrent, the class ExampleCreateTorrent implements these different steps and can be run easily to create any description file. The use of this class is the following:

ExampleCreateTorrent <torrenSavingPath> <announceURL> <pieceLength> <filePath1> ... <filePathN> <..> <creator> <..> <comment>

Then for John to publish his holiday pictures and movies, he'll only have to run the submitted example and provide all these parameters and the paths of his files in order to create the description file.

4.1.2 Publish torrent file

This step is way more hazardous than the precedent one. Indeed, as far as I have seen, there is no official and typical way of publishing torrents on web servers and trackers. Most of time, in order to publish a torrent, you have to create an account on the server, and then authenticate yourself on it (with username and password). Unfortunately, servers often require cookies to be present to authenticate the user and also a *captcha*[13] to avoid bots to publish torrents automatically, and thus prevent automatic authentication. For the moment, we can only publish torrent files on a few numbers of trackers that do not require too much authentication.

In practice this step is circumvented simply by automatically logging into the tracker using a prerecorded username and password and/or sometimes using cookies and session authentication sent by the tracker at login time and then post the torrent to the tracker upload page. Note that jBittorrentAPI only provides the possibility to publish on smartorrent tracker since it is among the relatively big trackers (> 300'000 users) the one with the less authentication needs. No captcha, it only need a username and password to login and upload torrent.

Here if John wants to publish on smartorrent, he'll have to go on the tracker web page http://www.smartorrent.com and register an account. Then after filling up the form and confirming his account creation thanks to the received email, he'll be able to publish on this tracker using the provided example ExamplePublish with the username and password he just created and the trackerURL set to 'smartorrent'. The program will automatically generate the URLs it needs to contact the smartorrent tracker login and upload pages.

But here we can see that it can be quite annoying for John to follow all these steps, he may want to have a faster way of publishing his files, with no need to register an account. And the answer to these needs is the simple trackerBT tracker…

4.1.2.1 trackerBT: a simple Bittorrent tracker

As one can see, finding a tracker that accept anonymous torrent upload is quite hard. That's why I decided to write my own tracker, a very simple one providing all basic functionalities and easy to use for a lambda user.

The tracker is totally independent, in the sense that there is no need for the user to have a web server already installed on his computer. The tracker itself behaves as a web server, with additional services to permit clients to publish torrents and retrieve peer information. The tracker therefore is based on the Simple Web library.

Since the goal here is to keep the program as simple and as light as possible, the tracker does not use a database to store torrents or peers

[13] Captcha is a type of challenge-response test used in computing to determine whether or not the user is human. See at http://en.wikipedia.org/wiki/Captcha

information. All is done using XML files, where data is stored and read by the service to retrieve information about the torrents currently owned by the tracker and the peers currently announced. Let's call these 2 files the ***XML torrents file***, respectively the ***XML peers file.***

The main class Tracker loads configuration data such as the root of the server, the port on which the tracker should listen or the name of the XML files to be used for information storage. The name and path to the configuration files is left to the user who has to specify this information when running the tracker. Once the setup is done, the Tracker class starts the web server on the specified port and waits for incoming requests. Depending on the URL specified in these request, the Tracker class, by the use of Simple Web libraries will instantiate one of the 3 service classes: FileService, UploadService or TrackerService.

The specific URL corresponding to a service is configured in the 'Mapper.xml' file, which maps a given URL to a specific service. To keep the configuration as simple as possible, this file is automatically generated and put in the root directory of the server. Let's see what these services are for and how they work.

4.1.2.1.1 FileService

This service is the standard serving service, i.e. it simply responds to the request by serving the file that is specified in the URL. If the file does not exist, an HTTP404 error page is returned to the client.

4.1.2.1.2 UploadService

The upload service is called when a client tries to upload a torrent to the tracker. For the torrent to be correctly published, the request must be a multipart/form-data request containing several fields:

- name: the name that the user would like the torrent to have on the tracker. Can be different from the real torrent filename. For the moment it is not used, but in the future it could be used to download the torrent from the tracker for example.
- comment: comments about the torrent, description of the files. Again, this is not used yet. This could be used for informing a client of the content of the torrent.
- torrent: the torrent file the user want to publish

Although the name and comment field are not used yet, I decided to keep them since it could be useful to have them if improvement in the tracker are done. For example, it is possible to implement a search service, providing the user a list of torrents published on the tracker, with their name and description. This could prove very useful for users to retrieve the torrent file they need.

Therefore, if all these fields are not present, an error message is sent back to the client, within an HTML page. Otherwise, the torrent file is decoded using BDecoder.java. This decoding is done for 2 reasons. First it is a way to verify that the torrent is valid since the BDecoder class throws an error if the torrent is not correctly constructed. Second it permits to extract the info dictionary which will then be hashed to create the torrent id that will uniquely identify the torrent on the tracker and for future exchange with peers.

Once these steps are completed, the tracker checks if the torrent is already registered. If not, it is saved in a local directory determined in the configuration file and the XML file storing the torrents id, name and comment is updated by adding this torrent's information.

Finally a response is sent to the tracker specifying if the torrent has been successfully added and if not, the reason of the failure. The response is a simple HTML page displaying the error message.

4.1.2.1.3 TrackerService

The last service implemented for the tracker is the TrackerService. This service is called when the tracker receives a request on its announce URL, meaning a peer wants to get the list of peers sharing the same file.

First this service tests if all the required parameters are present in the request according to the Tracker HTTP/HTTPS Protocol (see section 3.3.2. If the request is conform, then the service test if the torrent identified by the info_hash parameter is already owned by this tracker (i.e. it has been previously published) by checking the XML torrents file to contain the info_hash in any of its nodes. If the torrent hash is found, then the service reads from the XML peers file all the peers that are currently sharing the file (removing the too old ones) and then registers the current client in it. Finally the service bencodes the peer list in non compact form and send the created bencoded string to the client as the response.

The nice thing here for John is that this tracker is very easy to use for him. He almost has nothing to do to make it work. The trackerBT provided in the jBittorrentAPI.jar release comes with an example configuration file where John will be able to choose in which directory the torrent root will be set and torrents saved. Then he can just launch it and create torrents with his own IP address.

For the moment, John can choose on which port the server will be listening. This determined the announce URL he has to provide in his created torrent files. For example by default the server listens on port 8081. Let's suppose John has an internet address redirected to his computer, for example, *myaddress.john.com.* Therefore the URL to be used in torrent file has to be:

http://myaddress.john.com:8081/announce

and the URL used to publish file with the ExamplePublish should be:

http://myaddress.john.com:8081/upload

4.1.3 Share the target files

Once the torrent file is published on a tracker, remote peers can have access to it and therefore they may want to get the files. But as one has seen, nobody at the moment is really sharing the target files, only the torrent file is available. It may happen that the tracker only owns the torrent file but not the file itself, and no other peers are sharing the target files either. Then for someone to be able to download the target files, we have to run a Bittorrent client and start "downloading". Yes, this may seem strange that for people to upload from us, we have to start downloading. But actually when the client announces itself to the tracker, it will say that it already owns the whole files (the 'left' value is set to 0 since he owns the whole files data) and is only sharing it with other peers.

Back to the example, John will run his Bittorrent client and select the torrent he just published. The client will then see he already has all pieces and will therefore start sharing files with peers that miss some. Note here that in our example classes, we choose to provide ExampleShareFiles and ExampleDownloadFiles classes, i.e. 2 different classes to hide this strange fact of "downloading for uploading" to the user. But basically they are the same since the ExampleShareFiles just creates an instance of the ExampleDownloadFiles. But again for John no matter what the program is doing, the important thing is that when running ExampleShareFiles, he provides his pictures and movies to his friends.

4.2 Downloading torrents

Once the torrent file has been released on at least one tracker, each user has the possibility to download it and use a Bittorrent client to retrieve the target files. In our software, the client sends an HTTP GET request for the file to the server. Once the file is received, it is directly processed by a specialized class (TorrentProcessor.java), which extracts all the features of the torrent, such as tracker URL, piece length, pieces hashes… The result is stored in a special object called TorrentFile, which will be used by other classes to start the download processes.

Also, during the torrent processing, we could choose the files that we really want to download and skip the undesired ones[14].

In our example, John's friends that want the file will only have to retrieve the torrent published by John. They can get it either directly by John (who could for example send the torrent by email, since it has a very compact size, ~tens of kB) or found it on the tracker and web servers where John published the file.

4.3 Retrieving remote peers information

After the torrent processing, all necessary information is available to start the download process since a TorrentFile object has been created. Therefore the client is able to instantiate the DownloadManager class. According to the TorrentFile object provided, the new created object will create a list of Piece object representing of course the pieces specified in the torrent file. The Piece object creation can last a few seconds since I choose to determined at running time the list of target files each piece belongs to as much as its position in these files. This choice is made in order to speed up the saving and retrieval of pieces from the files during the download or upload process.

Then the client can start listening for incoming connection from other peers by instantiating the class ConnectionListener and starting the listening thread. This class basically tries to create a ServerSocket that listens for incoming peers connection requests. Once a connection is accepted, the thread informs the DownloadManager of the new peer arrival and let it decide if yes or no the connection has to be confirmed by the start of the handshake or just dropped.

Now everything is set up except that we still do not have the list of peers sharing the files described in the torrent. In order to get this list, the program contacts the tracker specified in the TorrentFile object using the ***tracker http/https protocol*** described in Section 3.3.2. To do that, it creates a new Thread by instantiating the class PeerUpdater and starting the thread. The created thread will then regularly contact the tracker to announce that this client is still sharing this torrent and to retrieve new peers. The answer of the tracker will be, as stated before, the list of peers also sharing the same file.

The tracker's reply will be processed and peers information such as IP address, listening port and ID (in the non compact case) will be stored in Peer objects and made available for the DownloadManager object. We can say here that all along the API, a peer is not really identified by its ID field

[14] This features is not yet implemented in our software

but more by the combination of its IP address and listening port. This decision is motivated by the fact that in the case of compact response from the tracker, no ID is provided.

4.4 Retrieving the target files

Now that the client has information on peers, it will try to contact them. This is done in parallel, with the use of Threads. For each new peer found, the program will create a new DownloadTask object. Of course, the number of new connection is limited and this limit can be chosen by setting the maxConnectionNumber value.

This class extends the Thread class and so, there will be as many new threads created as the number of new peers found. The number of new DownloadTask created can be configured thanks to a parameter in the configuration file.

The DownloadTask object will first try to contact the corresponding peer at the given IP address and port by creating a socket. If the connection failed, then the task simply ends. But if a connection can be established, then the client starts the Peer Wire Protocol described in Section 3.3.3 and the corresponding message exchange.

Since all tasks are executed in parallel, we need something to coordinate the different tasks and to manage the message exchange between peers. For example, we do not want the same piece being requested to all peers at the same time, since it will be a waste of bandwidth. My answer to this problem is the DownloadManager object. This object stores all information about requested pieces, active tasks and so on. It implements a listener for all DownloadTask and waits for their messages.

When a DownloadTask receives a request from its remote peer, this request is passed to the DownloadManager. According to the peer state it chooses if the piece block should be sent or not. As we have seen in the protocol specification, a remote peer can request a piece block only if it is unchoked and interested. If these 2 conditions are not satisfied when receiving a request message, the peer is immediately disconnected since it does not respect the Bittorrent protocol. A loop periodically checks which peers should be unchoked and which one should be choked according to their download or upload rates. This decision is called the Unchoking Algorithm and is performed about every 10 seconds to avoid peer state to change too quickly, which is known as fibrillation. The unchoking is done in respect with the following algorithm:

- Sort peers according to their download rate, i.e. the rate at which this client is currently downloading from them. If download is complete, they are sorted according to the upload rate. This last fact is to speed up the time a peer will become a seed, providing therefore more availability to the target files

- Pop the next peer from the list (i.e. the one with the highest DL or UL rate of the peer remaining). If it is not interested, then it should be unchoked. If it is not yet unchoked, send it an unchoke message.
  If the peer is interested, unchoke it if there are not already more than 5 peers that are interested and unchoked. Peers interested and unchoked are called *downloaders.* This limitation is to provide a significant upload rate to the remote peers and rewarding the peers that let us download.
  If it is interested and there is already 5 peers interested and unchoked, then choke the remote peer and add it to a list of peers to be unchoked optimistically
- A peer is unchoked about every 30 seconds. It means that the first peer in the list of choked peers in pop up and unchoked. This is done in order to try to find a better downloader and this is why it is called optimistic unchoking, since this client does not now a priori if unchoking this peer will result in a better download rate.

It is also the DownloadManager that stores the received pieces and saves them into the final target files into the corresponding directories.

For John and his friends, this step is done transparently. The only thing they have to do is launch either the ExampleShareFiles or ExampleDownloadFiles précising the torrent that should be processed and then the retrieval of the files is done automatically. They just have to wait for the download process to end… very quickly!

5 Conclusion

5.1 Achievements

Up to now, we concentrated on the understanding of the protocol and on the development of an application capable of both publishing and retrieving files using Bittorrent protocol. This application is now available, even if some features have not been fully implemented yet. We have now a strong background on how Bittorrent protocol works and how it could be used for web browsing and we are pleased to provide this API for future enhancement and work with it. We are sure it will be very useful for the next part of the project, consisting in transforming a web browser and a web server to make use of Bittorrent protocol qualities.

This first part enabled us to understand the precise operation of P2P networks and particularly Bittorrent protocol. We were able, thanks to the documentation provided on Internet but also with the use of forums and newsgroups, to develop a brand new application within a reasonable time. Also, we looked for existing open source code, reusing some code when available.

A list of all these sites is available under References chapter. According to my opinion this practical work is the only way to properly understand how the whole protocol works and how it can be used and improved. Moreover, the necessity to communicate with other developers was a great pleasure and provides me lots of good ideas.

5.2 Difficulties

Some problems arise along the development of the application.

The first one was related to the development language, JAVA. We thought it would be easy to find libraries, projects or classes to rely on and start with. But the only implementation of Bittorrent that we were able to find was Azureus, an amazingly complete and complex application. We decided therefore to develop a new, simpler application and API so that the new person working on this project can easily understand it and continue its development without too much loss of time.

Another problem was related to the network topology. Firewalls and NAT causes a lot of problem during the development, when trying to test the application with our own means. Many peers are not reachable directly, since their users did not configure their firewalls to forward requests and therefore this caused lots of host to be unreachable.

5.3 Future works and improvements

The next step will be to develop the framework to adapt web browsing to Bittorrent network. Now that the publish/retrieve program is ready, it should be possible to use it along with the whole background acquired during this first part to make web browsing capable of using Bittorrent network when it is required.

Concerning the applications developed during the first part, we could also improve it by implementing some features that could be very useful or that have not been implemented yet, for example the up/down rate management by using a better (optimistic) unchoking algorithm.

Also for the moment our client works using the official specifications, which uses TCP for message exchanges. Nowadays extensions to the official specifications exist that permit the use of UDP exchanges and also provide some kind of NAT Transversal using UpNP

There can be also improvement in the use of XML files which is not optimal at the moment.

And finally, the last thing we could improve is the use of the Simple Web server, which is simple but maybe still too complete for the tracker application. There are many unused features of it that could be removed and certainly libraries that are not needed for the tracker to work. So maybe in this part is it possible to simplify further the program, making this API as light as possible for its use within web browser or server…

5.4 Final word

This project has been a real pleasure to go through. It required quite a large amount of work but finally considering the amount of experience acquired along it, it was worth it. Working with other developers on a subject as popular as Bittorrent was truly a great experience and I hope the resulting API will satisfy the needs of the supervisors and future developers. I'd like to thank David Portabella Clotet for the time he spent helping me, especially in the end of this project.

I finally hope that this project will continue to evolve and reach the desired objective as soon as possible.

6 References

- http://sourceforge.net/projects/bitext: official website of this project
- http://www.Bittorrent.com : official website of Bittorrent
- http://www.Bittorrent.org : a forum for Bittorrent developers
- http://www.wikipedia.org : online encyclopedia providing huge amount of information. See in particular http://wiki.theory.org/BittorrentSpecification
- http://lists.ibiblio.org/pipermail/Bittorrent/ : list of archives about Bittorrent developments. Huge database and many articles, questions and answers about all features of Bittorrent
- http://dessent.net/btfaq/ : many interesting information about Bittorrent related applications
- http://azureus.sourceforge.net: official web site of the Azureus project, another java based Bittorrent client
- Some trackers and torrents databases:
  - http://www.thepiratebay.org
  - http://www.smartorrent.com