Increasing the Usability of BitThief: A Free Riding BitTorrent Client

A Dissertation
Submitted In Partial Fulfilment Of
The Requirements For The Degree Of

MASTER OF SCIENCE

In

NETWORK CENTERED COMPUTING,
High Performance Computing

in the

FACULTY OF SCIENCE
THE UNIVERSITY OF READING

by

Ben Hennig

28/09/2008

University Supervisor: Ismail Mahomed Bhana
FHTW Berlin Supervisor: Prof. Wolfgang Schebesta
ETH Zurich Supervisor: Thomas Locher and
Prof. Dr. Roger Wattenhofer

Acknowledgements

I would like to thank my supervisors, Thomas Locher who always offered me advice and support, further Prof. Dr. Roger Wattenhofer and all other members of the DCG at the ETH Zurich. Thanks also my supervisor, Prof Wolfgang Schebesta from the FHTW Berlin for the help by this report. In addition my supervisor Ismail Mahomed Bhana from the University of Reading.
Especially, I would like to thank my family who supported me during and further in writing this thesis.

Abstract

This master thesis gets an overview about the *BitTorrent* protocol and it shows the concept of the *BitThief* client, what is an ongoing project from the Distributed Computing Group at the ETH Zurich. The *BitThief* client is a free riding *BitTorrent* client which is uses for scientific work for analyzing P2P systems and to improve and/or advance the *BitTorrent* P2P protocol. The concept of the *BitTorrent* protocol and the *BitThief* client is described in the first part of this thesis.
My work on this project is to increase the usability of the client and further the developing of new unique features to increase the user community. The increasing of usability and the developing of unique features are important for the whole project, because it should appeal more users and so the client is spread more. This point guarantee that all statistical analysis are more precise. My own work on the project is described in the second part of this thesis.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

ADSL	Asymmetric Digital Subscriber Line
ASCII	American Standard Code for Information Interchange
CD	Compact Disc
CGI	Common Gateway Interface
DHT	Distributed Hash Table
DNS	Domain Name System
DoS	Denial of Service
DVD	Digital Versatile Disc
FTP	File Transfer Protocol
GB	Gigabyte
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identifier
IP	Internet Protocol
ISO	International Standard Organisation
Kademlia	An algorithm for DHT
KB	Kilobyte
LAN	Local Area Network
LTS	Long Term Support
MB	Megabyte
MD5	Message-Digest Algorithm 5
NAT	Network Address Translation
OS	Operating System

OSI	Open Systems Interconnection
P2P	Peer-to-Peer
PC	Personal Computer
RFC	Request for Comments
RPC	Remote Procedure Call
SHA	Secure Hash Algorithm
TCP	Transmission Control Protocol
TEN	Text Email Newsletter
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VM	Virtual Machine
WAN	Wide Area Network
XML	Extensible Markup Language
XOR	Exclusive OR

1. Introduction

The influence of the Internet has increased enormously over the last years. Many people in the world are working together in teams and sharing their content via the Internet. The industry, like content providers and software developers or the open source communities, would like to distribute its products and also many private users of the Internet would like to exchange material over the world-wide web. Therefore the distribution of content is important and so is the development of sophisticated tools are devoted to this purpose. Nowadays, different approaches exist.
One approach for distributing content in a fair way is to use a P2P (peer-to-peer) network. Bram Cohen[1] developed a P2P file sharing communication protocol called *BitTorrent*[2] in 2001. This protocol was implemented in the *BitTorrent* client. Today many other clients exist where the protocol is implemented. The *BitTorrent* protocol should be shown that it is feasible to share a huge amount of data in a fair way. The idea behind it was that every user who is downloading a file, uploads its already downloaded part to other users. The meaning of fair in this context is that the costs are split between all users, because in the Internet the network provider have to pay for the upload and not for the download. Another advantage of this protocol is, if many users are a part of this kind of distribution the speed of distribution is very high.

The Distributed Computing Group at the ETH Zurich[3] had developed his own *BitTorrent* client which is called *BitThief*[4]. With this client it was shown that the *BitTorrent* protocol is not as fair as described by Bram Cohen original protocol suite [1]. The *BitTorrent* protocol in the *BitThief* client is modified and pretend the tracker and other peers that *BitThief* is fair and especially powerful.[2] Thereby the download rate is increased on the *BitThief* side enormously, although there is no real upload.
The client is written in Java 5 and should work independent of the chosen OS. Note that programs written in Java cannot be guaranteed to work independent of the OS, because some methods depend on the system and it is the task of the programmer to guarantee the correct functionality. The *BitThief* program is used for scientific analysis and currently in a rudimental condition. At the beginning of my thesis, with the *BitThief* version 0.1.7, the most features which are implemented can be called only

[1]http://bitconjurer.org/
[2]www.bittorrent.org
[3]Swiss Federal Institute of Technology Zurich
[4]http://dcg.ethz.ch/projects/bitthief/

from the command line. The GUI at this version is very elementary and supported sparsely feature. Many parameters which can be set in the command line are not supported in the GUI version.

The aim of my work is to make the GUI more attractive for users, not directly interested in the scientific aspect of the BitTorrent protocol, to collect data to show that our findings are correct. That means to show that the BitTorrent protocol must be extended in some direction to be real fair in the sense of a fair upload of the participants. The client is a scientific project and collect data for statistical analysis of this client, therefore it is require that many users uses this client and to guarantee a big user community *BitThief* needs an easy user interface and unique features.
This thesis is split into different chapters. At first I will describe, in Chapter 2, the basics of *BitTorrent* with some concepts of this protocol. In Chapter 3, I will describe the concept of the *BitThief* client how it works now, and I will introduce the new way of the client. Chapter 4 shows the state of *BitThief* before I started my work. In the following chapters, I will describe my assignments on the project.
My assignment is to increase the usability of the *BitThief* client. That includes different aspects, at first it should increase the look and feel of the program, so that the client is more attractive for users (see Chapter 5). Further, to guarantee that users works with the latest version of the client an update mechanism must be implemented. This function is describe in Chapter 6. A nice look and feel and an update function are only the basics to use a program nowadays, to get a greater community a program needs some unique features which convince users to use this client. For this purpose I developed different features. In Chapter 7 is a multi-torrent function described where the user can selects several files from a torrent to download only this subset. To disenthrall the user from a web browser a torrent search function was implemented. This functionality tries to rank all search results and gives the user an overview about good torrents for downloading, so that it will be avoided to download corrupt or dead torrents. The search function is described in Chapter 8. Many users uses *BitTorrent* to watch and/or download TV shows, therefore it is very useful to subscribe to a TV show. This subscription service searches automatically for new shows and offer these results. Further, this functionality can be used for all other torrents what represent an automatic search function. The subscription service is describe in Chapter 9. The last chapter (Chapter 10) of this thesis describe different error analyzes which must be performed during my thesis. This part was needed to create a stable version of the *BitThief* client.

2. The BitTorrent Protocol

The *BitTorrent* protocol was developed and implemented by Bram Cohen in 2001. The aim of the protocol was to distribute huge amount of data and/or files in a clever and efficient way. Before it, data were distributed via HTTP or FTP from special file servers. This technique has a main disadvantage, if downloading a huge amount of user data from a server at the same time then the bandwidth reduced enormously. The result of it could be that the server crashed, nowadays this phenomenon is also called DoS attack. With the *BitTorrent* protocol it cannot happen, because the protocol is based on a P2P technology. If many users are downloading the same file concurrently then the load is shared of many peers and/or clients. With *BitTorrent*, the load is distributed among all peers which results in an increased throughput. The *BitTorrent* protocol is a collaborative file sharing protocol which is implemented on layer 7 of the OSI-reference model.

2.1. Terminology

The *BitTorrent* protocol introduces new terminology. I will itemize the core keywords first and then I describe them briefly in detail.

- Torrent
- Tracker
- Peer
- Seeder
- Leecher
- Swarm
- Choke
- Unchoke

Torrent A *torrent* has two different meanings depending on the context. It can mean either a .torrent metadata file or all files described by it. The torrent file contains metadata about all the files that it offer, the file contains:

- names and paths of the files

- file lengths
- piece length
- piece hashes (checksums)
- tracker announce URL

If a user wants to share some files using BitTorrent, at first he needs to create a torrent metafile.
Tracker A *tracker* is a server that keeps track of which seeds and peers are in the swarm. The clients report their information to the tracker periodically and in exchange they receive information about other clients to which they can connect. The tracker is not directly involved in the data transfer and does not keep a copy of the transferred file.
Peer A *peer* is an instance which is using the BitTorrent protocol. The program is running on a computer on the Internet to which other peers are connected and transfer data. Usually a peer does not have the complete file, but only parts of it. However, in the colloquial languages, "peer" can be used to refer to any participant in the swarm, in this case it is synonymous for a "client". A peer is also called a node.
Seeder A *seeder* is a peer that has a complete copy of the torrent (note not a .torrent file) and still offers it for upload. The more seeders there are, the better the chances are for completion of the download.
Leecher A *leecher* is a peer which is not a seeder. That means the *leecher* only has a part (or nothing by startup) of the download and is still downloading.
Swarm A *swarm* consist of all peers (including seeders) which are sharing the same torrent. For example, six ordinary peers and two seeders make a swarm of eight.
Choke The *choke* mode limits the bandwidth, what means that the *BitTorrent* client shared the content with a fixed number of nodes.
Unchoke *Unchoke* is the opposite of choke and remove the limitation. The client can share with all available peers.

2.2. Mode of Operating

In comparison with conventional methods to download a file via HTTP or FTP the *BitTorrent* technique uses the normal upload capacities of the downloader, even if the torrent and/or file is incompletely downloaded. The files are not only distributed by one server but also from client to client (see Figure 1). This is the *BitTorrent* concept of peer-to-peer. Overall the download load is not lower but it will be shared about a lot of individual users. For popular files this technique prevents the collapse of the network and server.

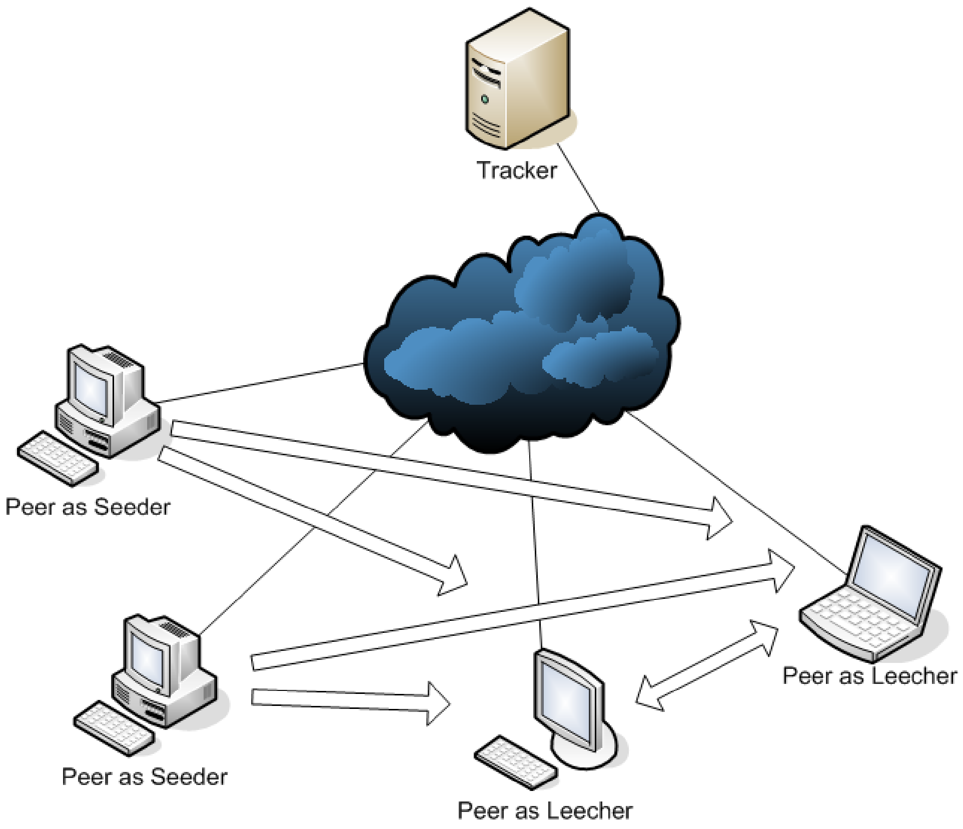


Figure 1: Concept of BitTorrent: All clients are registered on the tracker. The tracker tells all clients who is currently in the swarm. The seeders send packages to the leechers. Also the leechers could send packages to other leechers.

2.2.1. Design of a Torrent File

To download a file the client need a *torrent file* with the extension *.torrent* or *.tor*, it can be understood as a description file and is called the torrent metainfo file. This file has a special encoding form which is called *bee-encode* or in short form *bencode*, so content of a metainfo file is a bencoded dictionary. The metainfo file consists of IP-addresses and/or host addresses of the tracker as well as the file name, file size and the checksum of the download file. The size of the description file is typically a few kilobyte and is usually placed on a webpage by the provider.
Also the *.torrent* file is a control file which controls the communication between the

seeders, *leechers*, and the tracker. The content of the file can be seen with a standard text editor. The first rows of the example file ubuntu-8.04-desktop-i386.iso.torrent is shown below (Listing 1).

```
d8:announce39:http://torrent.ubuntu.com:6969/announce7:comment
29:Ubuntu CD releases.ubuntu.com13:creation datei1209021836e
4:infod6:lengthi733079552e4:name28:ubuntu-8.04-desktop-i386.iso
12:piece lengthi524288e6:pieces27980:
```

Listing 1: Content of a *.torrent* file with one tracker and one download file.

These rows show the *BitTorrent client* that it has to connect to the tracker *torrent.ubuntu.com:6969/announce* and request the file *ubuntu-8.04-desktop-i386.iso*. At the beginning of the first row there is a command *d*, which means that the following entry is a so-called dictionary. The number before the colon defines the length of the following string. For example the *39:* shows the client that the following string, in this case the tracker URL, has a length of 39 characters. The size of the file to be downloaded is described by the keyword *length*, whereas the following *i* indicates that the next string is an integer value. All number strings end with an *e*. For example i730e corresponds to 730, i-730e corresponds to -730 or i0e corresponds to 0. Integer values have no size limitation, but there exist invalid formats e.g. i-0e or i03e. In the torrent file example the downloading file has the size of 733,079,552 Byte (699.12 MB). In general, all elements like strings, lists or numbers end with an *e*, the *e* is the terminate sign. However, the described file(s) will not download as one file, it is split into many parts. Every part is called a piece and in these example every piece has the size of 524,288 Byte (512 KB). The most common sizes are 128 KB, 256 KB, 512 KB, and 1 MB.
Note that the piece length specifies the current piece size of each piece and is usually a power of 2. The typical chosen piece size based on the total amount of file data in the torrent. But it is to keep in mind that if the piece size is too large it could cause an inefficient download and otherwise if the piece size is too small, the torrent file is very large because each hash sum of each piece is a part of this file. So the typically chosen piece size depends on the total amount of file data in the torrent and the download size. The best choice is to keep a piece size of 512 KB or less for torrents under 8 GB, even if that results in a large torrent file but it is guaranteed to have a more efficient swarm for sharing files.
The key word *pieces* indicates in how many pieces the download is split into. Every piece is of equal length except for the final piece, which is irregular, that means the last piece consist of the missing bytes which are mostly smaller as a piece. In the example

from listing 1 the Ubuntu image consist of 27,980 segments and/or pieces.
To get the correct file size we have to divide the pieces by 20.

$$27980 \div 20 = 1399$$

Dividing by 20 is a protocol specification, because pieces map to a string whose length is a multiple of 20. It is to be subdivided into strings of length 20, each of them is the SHA-1 hash of the piece at the corresponding index[3]. Now if we multiply this result with the piece size in KByte then we have the correct file size.

$$1399 * 512 = 716,288 \quad KB \quad (699.5 \quad MB)$$

In another example, which can be seen in Listing 2, should show that a torrent can be administered by more than one tracker. The different trackers are described by an announce list. The indicator for a list is a *l* followed by its elements. Also the torrent file can consist of more information than one target file. This can be called a multi-torrent file and is also described in a list with the individual file name and size. An example for a multi-torrent file is given in Listing 2 as well.

```
d8:announce43:http://tracker.torrentbox.com:2710/announce
13:announce-listll43:http://tracker.torrentbox.com:2710/announcee
l44:http://vip.tracker.thepiratebay.org/announcee
l44:http://tpb.tracker.thepiratebay.org/announcee
l30:http://tracker.prq.to/announceee
13:creation datei1213452878e4:infod5:filesld6:lengthi73233130e
4:pathl47:Madonna - Give It 2 Me [2008][SkidVid]_XviD.aviee
d6:lengthi63404e4:pathl26:Madonna - Give It 2 Me.jpgee
d6:lengthi205782e4:pathl15:ScreenShots.jpgeed6:lengthi2206e
4:pathl14:Video Info.txteee4:name
43:Madonna - Give It 2 Me [2008][SkidVid_XviD]
12:piece lengthi262144e6:pieces5620:
```

Listing 2: Content of a multi *.torrent* file with different tracker URLs.

All single files in a multi-torrent file are described as one long continuous stream. The number of pieces and their boundaries are determined in the same manner as the case of a single torrent file. The order of pieces in the multi-torrent file composed of the concatenation of each file in the list, which means that a piece boundary may overlap a file boundary or in other words one piece can consist of a file end point and starting point or more files (if the files are smaller than the piece size).

2.2.2. Distributing of Content via Tracker

BitTorrent consists of two parts, the server program which is called tracker and the client program, which is installed on a PC by the user and serves as a remote station. The tracker administrates information for one or more files. The downloading client learns from the tracker who else is downloading and distributing the file. If a client has received a segment or chunk of the desired file and has verified the checksum, then the client sends a response to the tracker. The tracker informs the other clients in the swarm and the current client can distribute this chunk to the others.
If the provider removes the torrent out of the tracker then the contact to the tracker is lost and the clients cannot contact other new clients. But the clients can carry on exchanging pieces with the known clients.
In contrast to other well known file sharing systems *BitTorrent* does not exchange any other data from the user. In fact, the swarm distributes only the files which the user of the torrent file has specified for downloading. Also the provider of the tracker can determine which downloads it manages. The individual trackers are not interconnected, therefore a joint network does not exist. Every tracker creates a virtual and temporal network of involved clients.

2.2.2.1. Tracker HTTP/HTTPS Protocol

The tracker is a HTTP/HTTPS service which responds to HTTP GET requests. A request from a client includes information from that client to help the tracker to create a statistic about the existing peers in the swarm. The response from the tracker includes a list of all current peers in the swarm. This list helps the client to participate in this swarm. Both, the request and the response is in a bencoded dictionary form. The base URL for the tracker consists of the *announce URL* which is defined in the torrent description file. All parameters are then added to this *announce URL* using standard CGI methods (i.e. a '?' after the announce URL, followed by 'param=value' sequences is separated by '&'). CGI is a standard for data exchange between a web-server and third party software which handles the request.

All binary data in the URL must be properly escaped, especially the info_hash and the peer_id, for this purpose it is uses the URL encoding style. This means that any byte which is not in the set 0-9, a-z, A-Z, '.', '-', '_' and ' ', must be encoded. It uses the hexadecimal value of the byte therefore. An example for the coding is shown below, where a 20 byte hash sum will be encoded [4].

$$\backslash x12\backslash x34\backslash x56\backslash x78\backslash x9a\backslash xbc\backslash xde\backslash xf1\backslash x23\backslash x45$$
$$\backslash x67\backslash x89\backslash xab\backslash xcd\backslash xef\backslash x12\backslash x34\backslash x56\backslash x78\backslash x9a$$

Table 1: Coding table: Example for an encoded 20 byte hash sum.

Hex value	Sign 8 bit ASCII	URL code
12	DC2 (CTRL R)	%12
34	4	4
56	V	V
78	x	x
9A	´	%9A
BC	$^1/_4$	%BC
DE	Þ	%DE
F1	ñ	%F1
23	#	%23
45	E	E
67	g	g
89	‰	%89
AB	«	%AB
CD	í	%CD
EF	ï	%EF
12	DC2 (CTRL R)	%12
34	4	4
56	V	V
78	x	x
9A	´	%9A

The whole URL encoded sequence is:

$$\%124Vx\%9A\%BC\%DE\%F1\%23Eg\%89\%AB\%CD\%EF\%124Vx\%9A$$

2.2.2.1.1. Tracker Request

If the client sends a request to the tracker the following parameters are part of the GET request. Some parameters are optional and all others are mandatory. At first I describe the obligatory part and then the optional part.
peer_id: The peer ID is always a 20 byte string used as an unique ID for the client. The client generates this ID at startup. Any value is allowed for the ID and may be

binary data. To generate this peer ID currently no guideline exist. But one popular procedure is using the process ID and a time stamp at launching. Other strategies exist and are called the *Azureus-style* and the *Shadow's-style*.
The *Azureus-style* is encoded as follows: it starts with a '-', then two characters for the client id (and/or program id) followed by four ASCII digits for the version, a separator '-', and then a random numbers to filled in this string. An example for an Azureus client is:

$$-AZ2060-\dots$$

Further client IDs are for example for μTorrent "UT" or for the KTorrent "KT".
The *Shadow's-style* uses another encoding for the *peer_ id*. The client identification is a one ASCII alphanumeric sign followed by up to five characters for the version number. If the version number is shorter than 5 signs then pad it with $'-'$. After this version, three characters follows, mostly $'---'$ but not always and the rest of the string will be filled up with a random sequence. Each character in the version string represents a number from 0 to 63. For example:

$$0=0,\quad \dots,\quad 9=9$$

$$A=10,\quad B=11,\quad \dots,\quad Z=35$$

$$a=36,\quad b=37,\quad \dots, z=61$$

$$.=62, -=63$$

An example for the Shadow 5.8.11 client is shown below.

$$S58B-----\quad+\quad\dots$$

port: The port number describes the port the client listens to. The standard ports for BitTorrent are typically between 6881 to 6889 but it is possible to choose a port out of this range.

info_hash: The info hash is a URL encoded 20 byte SHA-1 hash of the value of the info key which is described in the torrent file.

event: The client must specify its event, therefore three different states exist:

started, stopped, and completed. These values are key values of the event. The first request of the client to the tracker must include the event *started* which signals the initiation. If the client stops the current download the *stopped* state is sent. And if the client downloads all pieces it sends the *completed* state. This state signals the tracker to increment "completed download" statistic, it is only on this event based. This signal (*completed*) is sent only once. Because if the client restarts and the download was already 100% complete it would be fake the statistic and therefore it does not send again. If no event is specified, this means that this request is once performed at regular intervals.

downloaded: This parameter shows the tracker the total number of bytes downloaded. It is used if the client sends the *started* event. The coding of this part is based on the TEN ASCII standard.

uploaded: This parameter is also sent with the *started* state and is based on TEN ASCII standard. It informs the tracker about the total number of bytes are already uploaded to other peers.

left: The *left* parameter is for the total number of bytes which this client still has to downloaded. This is also encoded in TEN ASCII.

compact: If that value is set, it indicates that the client accepts a compact response and so the peer list is replaced by a peers string with 6 byte per peer. The first four bytes are the host address and the last two bytes are the port number. The bytes are ordered in the network byte order. The compact mode is for saving bandwidth and it is to note that some trackers only support this mode which means that if the mode is not set the tracker refuses this request.

no_peer_id: If the *compact* mode is disabled, then the tracker ignores the peer ID field in the peer dictionary by this option. If the *compact* mode is enabled, then this option is not active.

The following parameters are all optional.
numwant: Is it desired to restrict the client by sharing only with a bounded number of other client, then this value is to be set. With the result that the tracker responses with an available peer list with the respected count only. By default this value is set to

50 peers and it is permitted to set this value to zero.

key: It is an additional identification and is not shared with any other clients. It is intended to allow a client to prove their identity should their IP address change.

trackerid: If the torrent file describes more than one tracker, then it is meaningful to give each tracker its own ID.

ip: This parameter is to be set if the IP address where the request comes from is not the real IP address of the client. In that case, the client is placed behind a proxy server. It is also necessary that the client communicates via a NAT router. Or both, the client and the tracker are in the same local network on the local side of a NAT router. The reason for this is the tracker would give out the local address of the client to other peers and this class C address (in the RFC 1918 space) is not routable. Therefore the client and/or the NAT router must set its external routable IP address in this field to communicate with other peers. A problem of this parameter is that some trackers treat this value differently. Some trackers accept only this parameter if the IP address lies within the RFC 1918 space. Other tracker ignore this value completely. In general this parameter is not necessary if the address of the client can be determined from the IP address from which the HTTP request came.
The *ip* parameter is for both IP versions (IPv4 and IPv6). If the parameter is an IPv6 address (e.g. 2001:0db8:85a3::8a2e:7322) it indicates only that the client communicates via IPv6.

2.2.2.1.2. Tracker Response

The tracker responds in plain text which is in a bencoded dictionary form and can consist of the following keywords:
interval: The *interval* value indicates how long the client is to wait to send a regular request to the tracker. This time value is to be interpreted in second.
It is allowed that the client sends a request to the tracker more often than the specified interval. This happens if an event occurs especially by the *stopped* and by the *completed* event or if the client needs to know about more peers. But in the last case it is a better practice to specify the *numwant* parameter and so request a larger peers list, instead of bombing the tracker with multiple request.

min interval: This is an optional key to specify the minimal announce interval. This value shows the client that it must not reannounce more frequently that this value.

complete: The *complete* key informs the client how many seeders currently exist.

incomplete: This is the opposite of the *complete* key and means how many leechers currently exist. Both values are integers.

peers: This keyword describes the list of all available peers. The default length is 50, which means that maximum 50 different peers are described. If there are fewer peers in the swarm, then the list will be smaller. Otherwise, the tracker chooses peers randomly and adds these to the response list. If the peer is a seeder, then the tracker has to avoid to report other seeders, because these peers are uninteresting for that peer, therefore the tracker needs a more intelligent mechanism to send in that case only a list of leecher peers. For the peers list two different models exist, the dictionary model and the binary model.

- *dictionary model:* In this model the value is a list of dictionaries. The dictionary consists of the following keys.
  - *peer id:* This is the unique self selected client ID which was described above. This value was created by any individual client and is used to communicate with each peer in the swarm.
  - *ip:* This value can consist of different entries. It could be the DNS name which is represented as a string. Also it can consist of the IP address in the IPv4 format as dotted quad integer value or in the IPv6 format as hexadecimal values which are separated by a colon.
  - *port:* Is the listing port number for each peer.
- *binary model:* This model is instead of using the dictionary model. Each peer is described in a string consisting of 6 bytes. The first 4 bytes are the IP address of a special peer and the last 2 bytes are the listing port number. The notation is in the network byte order (big endian). This model is equivalent to the compact mode.

failure reason: If a failure occurred the *failure reason* key is set. If this key is set then no other keys are set and an error message follows in human readable form which presents the reason why the request failed.

warning message: This key is similar to the *failure reason* key, but the response still gets processed normally. This message is shown if a harmful error occurred, what destroyed not the functionality.

tracker id: This value is created by the tracker and if the client sent its next request and/or announcement it should use this ID value. If this value is missing the client has to use the old *tracker id* from the previous request. It means the client should not remove the old value. This ID is always a string.

2.2.3. Trackerless Distributing of Content

In newer versions a trackerless service has been developed. The tracker function will be carried out on the client side. That eases to provide the content, because the tracker is the most complex part of *BitTorrent*. Therefore the client uses a distributed hash table (DHT) algorithm. Modern BitTorrent-clients can be resigned on the tracker server and thereby they can run locally ("trackerless"). But the problem is that different BitTorrent clients have implement different types of DHT algorithms in their protocols, thus these clients are incompatible. Today the most efficient DHT algorithm is the Kademlia algorithm. More and more clients have implement this protocol and thereby these can avoid incompatibilities.

2.2.3.1. Distributed Hash Table

The challenge in P2P systems is the general problem to find the storage place for a searched data object in an efficient way. The data object, a file, is evenly distributed shared among many nodes, this data structure is a so called distributed hash table (DHT). The entry point is independent from any location and it must be feasible to find every responsible node. Each node is a container of an individual hash table. The demand on a DHT is robustness, self-organization, and scalability, because in a P2P system nodes need access, nodes enter and nodes leave the system at any time. The basic of distribution hash tables are the hash functions.

From a linear codomain the hash function generates a key for each data object. The codomain is evenly spread about the nodes of a node set. Each node is leastwise responsible for a subset of the key space, but it is possible that more than one node is responsible for the same key space, whereas the responsibility changes dynamically. The keys have no semantic meaning and therefore the provided content is software independent. This has the advantage that the implemented DHT-algorithm can be replaced by another implementation. The generic interface consist of only the *public(key, content)* function and the *lookup(key)* function.
To get access to existing system an entry protocol is needed, normally a new node will be connected to another node, which is currently part of the system. This protocol performs the connection to neighbor nodes and controls the construction of routing tables. Routing tables are used by DHT-nodes to find out, in an efficient manner, which content is provided by other nodes.
The definition of the distance to a neighbor is connected to the structure and topology of the current system and can vary in different implementations. The distance has no relation to the physical distance in the system, see below.
The content of the data object which is distributed can be placed directly by the node or it will point to an address (link) of another node which holds the data[5].

2.2.3.2. The Kademlia Algorithm

The Kademlia protocol was developed by Petar Maymounkov and David Mazières at the New York University in 2002. It is a P2P system based on DHT and uses the IP protocol with the connectionless UDP protocol.
Kademlia is used for file sharing and to find information for downloading in the network. A central instance does not exist, which performs the indexing of the available data (like eDonkey), the protocol shares this task with all clients. A node, which has an information, calculates a clear bit sequence with a defined length. This sequence, which is called hash, characterizes the information. The length of all hashes and IDs in the network have to be equal. The protocol looks up only for that nodes which ID (calculated in bit) have the smallest distance to the hash and transfers the contact data.

The algorithm constructs a decentralized virtuell network structure on the LAN/WAN, that is called DHT and it is the general approach as in other DHT. Kademlia defines the nature and the construction of the network and also it regulates the communication and the exchange of information between the nodes. Each node in the network has a unique number, so-called Node-ID, in the 160 bit key space. This ID is a randomly calculated number and not only used for identification. Every information content (value) which is placed or searched in the network has assigned a 160 bit hash value (key) from the DHT. Therefore every key represents an information content and builds a $\langle$key, value$\rangle$ pair. Each pair is stored on nodes whose IDs are close to the key according to some notion of closeness. To publish or find these pairs in the network *Kademlia* uses the distance between both bit sequences.

A node that wants to enter the network has to perform a bootstrapping process. In that phase the algorithm gets, from a user or a stored list, the IP address of a node which was currently known in the *Kademlia* network. Than to find this initial entry IP number it is not specified by the *Kademlia* protocol. If is the node a new one then the algorithm calculates a random node-ID, if that value used by another node then the algorithm performs the calculation again. This procedure ends if the algorithm finds a clear ID. The ID is used for the whole time until the node leaves the network.

The "distance" described above, has no relation to geographical distance, it marks the distance within the ID field. It could be that a node from Europe and a node from Australia are neighbors. Kademlia defines the distance between two IDs as their bitwise exclusive or (XOR) interpreted as an integer, $d(x, y) = x \oplus y$.

The protocol consists of 4 RPCs: *PING*, *STORE*, *FIND_ NODE* and *FIND_ VALUE*. The *PING* RPC probes a node to see if it is online. *STORE* instructs a node to store a $\langle$key, value$\rangle$ pair for later retrieval. *FIND_ NODE* takes a 160 bit ID as an argument. The recipient of the RPC returns $\langle$IP address, UDP port, Node ID$\rangle$ triples for the k nodes it knows about closest to the target ID. *FIND_ VALUE* behaves like *FIND_ NODE* with one exception. If the RPC recipient has received a *STORE* RPC for the key, it just returns the stored value[6].

2.2.4. Peer Wire Protocol

To share the pieces of a torrent between different peers a special protocol is required. This protocol is called the *peer wire protocol* and uses TCP for transmission. The torrent

metafile consists of a description for each piece which facilitates the peer protocol. The information of this chapter comes from [4], that is the wiki side of the BitTorrent project. To get more information in detail please read this website.
Each connection with each remote peer must consist of a state information (*choked* or *interested*).

- **choked:** If the remote peer chokes the client then the requests will not be answered until the client is unchoked. In the *choked* mode the client should not try to send requests for pieces, and otherwise the remote peers has to consider that all pending or unanswered requests will be discarded. Every client has two variables for each connection to keep track of the state.
  - *am_ choking:* This means my client chokes the remote peer.
  - *peer_ choking:* That is used for the remote peer if they chokes my client.
- **interested:** The *interested* state means the client is interested in a piece from another peer (what the remote peers offers). The interested peer will start this request if the remote peer unchokes it. Also as the *choked* mode the *interested* mode has two variables for saving the current state.
  - *am_ interested:* My own client is interested in a piece of that remote peer.
  - *peer_ interested:* The remote peer is interested in a piece which my client has.

The client can download a piece if it is interested in that piece which the remote peer has and further, if the client is not choked by the remote peer. For uploading a piece the remote peer must be interested in that piece and my client is not choking that peer. It is important for the client to save the peer states and to know if the peer is interested or not. Also if the client is *choked* it must refresh this state information to know if the client will begin downloading if the state changed to *unchoked* and vice-versa.
By starting a new connection it is set the *choked* and the *not interested* state, for this purpose the variables are set with the following values.

- am_choking = true
- peer_choking = true
- am_interested = false
- peer_interested = false

2.2.4.1. Handshake

To start the communication between two different peers an initialization phase is required. This is the handshake message and must be first transmitted by the client. The initiator of a connection sends its handshake immediately. This message has the abstract body which is shown below.

$$< pstrlen >< pstr >< reserved >< info\_hash >< peer\_id >$$

The denotation of these fields are the following:

- **pstrlen:** It is the string length of the following string (*pstr*), as a single raw byte.
- **pstr:** This string is the identifier of the protocol and is currently always *BitTorrent protocol* so that the string length is 19. This identifier identified the version 1.0 of the BitTorrent protocol.
- **reserved:** A bit field of eight bytes is reserved for changing the behavior of the protocol. Currently all implementations of the BitTorrent protocol do not modify this field and therefore they set all bytes to zero.
- **info_hash:** That is a 20 byte field for the SHA-1 hash of the info key which is described in the metafile. This is the same *info_hash* that is transmitted in tracker requests (see above Section 2.2.2.1).
- **peer_id:** Also the *peer_id* field is 20 bytes long and represents the unique client ID. Normally, it is the same ID which was sent to the tracker (see above Section 2.2.2.1), but some implementations use an anonymity modus which sets another ID for each remote peer (e.g. Azureus).

Altogether the handshake message has a length of 49 bytes plus the length of the protocol identifier, which is in this case 19 bytes and so the total handshake length is 68 bytes.
A remote peer has to listen to a handshake message and must respond to this message if it recognizes the *info_hash* of the wanted torrent. The *info_hash* is the identifier for a torrent because it is possible that a peer is able to handle more than one torrent at the same time. If a client receives a handshake with an *info_hash* that it is not available, then the remote client has to drop the connection.
If the requested peer receives a handshake response package in which the *peer_id* does match with the expected ID, then the peer can drop this connection. Note the client

must not save its handshakes, because it holds a peer list which was received by the tracker request before.

2.2.4.2. Message

All of the used messages in the protocol take the form of

$$< lengthprefix >< messageID >< payload >$$

The *length field* is a four byte value in the big endian format. The *message ID* is a single decimal byte. The length of the payload depends on the message. Now I will shortly described which message types are available in the BitTorrent protocol and how the body is built.

Keep Alive Message

The *keep-alive* message is a message with zero bytes, it is specified by that the length field is set to zero. This message has no ID and also no payload. If a peer receives no message (*keep-alive* or any other message type) from a participating peer for a certain period of time it can close the connection. This period is generally two minutes. The *keep-alive* package must be sent if no other command is sent in the last two minutes. This message has the following form:

$$< len = 0000 >$$

Choke Message

The *choke* message has a fixed length with the message ID '0' and has no payload. The form of this message is shown below.

$$< len = 0001 >< id = 0 >$$

Unchoke Message

The *unchoke* message has the same form as the *choke* message but the message ID is '1'.

$$< len = 0001 >< id = 1 >$$

Interested and Not Interested Message

The *interested* message and the *not interested* message is also a package which has no payload and a fixed length. The *interested* message has the ID '2' and the *not interested* has the ID ’3’. The *interested* message is used to tell the remote peer that my client is interested in its content and/or pieces. The *not interested* message is the opposite.

$$< len = 0001 >< id = 2 > \qquad\qquad < len = 0001 >< id = 3 >$$

Have Message
The *have* message is the first package with a real payload. The payload is the piece index, which was successfully downloaded before and verified by the hash sum. The piece index is zero based, which means the first index of a torrent starts with ’0’. The length of this message is fixed and the message ID is ’4’. This message offers other leechers in the swarm "I have a new piece".

$$< len = 0005 >< id = 4 >< piece\ \ index >$$

Bitfield Message
The length of the *bitfield* message depends on the current torrent, which means in how many pieces the torrent is split. The variable length is indicated by the ’X’ in the example, whereby the ’X’ means the length of the bitfield. This message should be sent immediately after the handshaking process is completed and verified and before any other messages are sent. If the current client has no pieces of the wanted torrent, then this message does not need to be sent. If the declared length of the message is unequal to the length of the real bitfield, then it is considered as an error. In that case the receiving client can drop the connection. The ID of this message is ’5’ and the payload is the described bitfield. The bitfield is the environment where the pieces are set if they exist otherwise the entry is zero.

$$< len = 0001 + X >< id = 5 >< bitfield >$$

Piece Request Message
If a peer requests a piece or a block, which is a subset of a piece, then it has to use this *request* message. The length of this package is fixed and the payload consists of the following information:

- *index:* It is the zero based index of the wanted piece and is an integer value.
- *begin:* If a block is requested, it specifies byte offset within a piece and is also an integer value. If the client downloads whole pieces, this field is ’0’.

- *length:* This field is also used by block transfer and indicates the block length.

The package ID is '6' and the body is shown below.

$$< len = 0013 >< id = 6 >< index >< begin >< length >$$

Piece Response Message

The *piece* package transmits a piece or block which was requested before. The length of this package depends on the length of the piece and/or block, which is indicated by the 'X'. The ID of this message is the '7'. This package consist of the following information:

- *index:* Is an integer value that contains the zero based piece index.
- *begin:* It is the offset of the wanted block from a requested piece.
- *block:* It contains the real data of a block or piece which was specified by the index.

The package construction is shown below.

$$< len = 0009 + X >< id = 7 >< index >< begin >< block >$$

Cancel Message

The selection of wanted pieces is random and therefore it is possible that a peer requests a piece by multiple peers. If the client has received a piece which was requested by different peers, it has to signalize all other peers that this piece is no longer needed. To signalize this state the *cancel* message is needed. This message has the ID '8' and the payload has the identical construction of the *request* message which was described above.

$$< len = 0013 >< id <= 8 >< index >< begin >< length >$$

Port Message

If the client uses a DHT algorithm for BitTorrent (see Section 2.2.3), it must publish its listen port number in another way as was described in the Section 2.2.2.1. Therefore the *port* message is used. This package has a fixed length, an ID of '9' and the number of the listen port.

$$< len = 0003 >< id = 9 >< listen\ port >$$

2.2.4.3. Message Flow

In this chapter I show two examples to get an idea of the message flow by BitTorrent,

both figures are simplified.
In the first figure (Figure 2) it is shown how a new leechers registers itself at a tracker. The tracker responds with a list contains all available peers, which is one seeder in

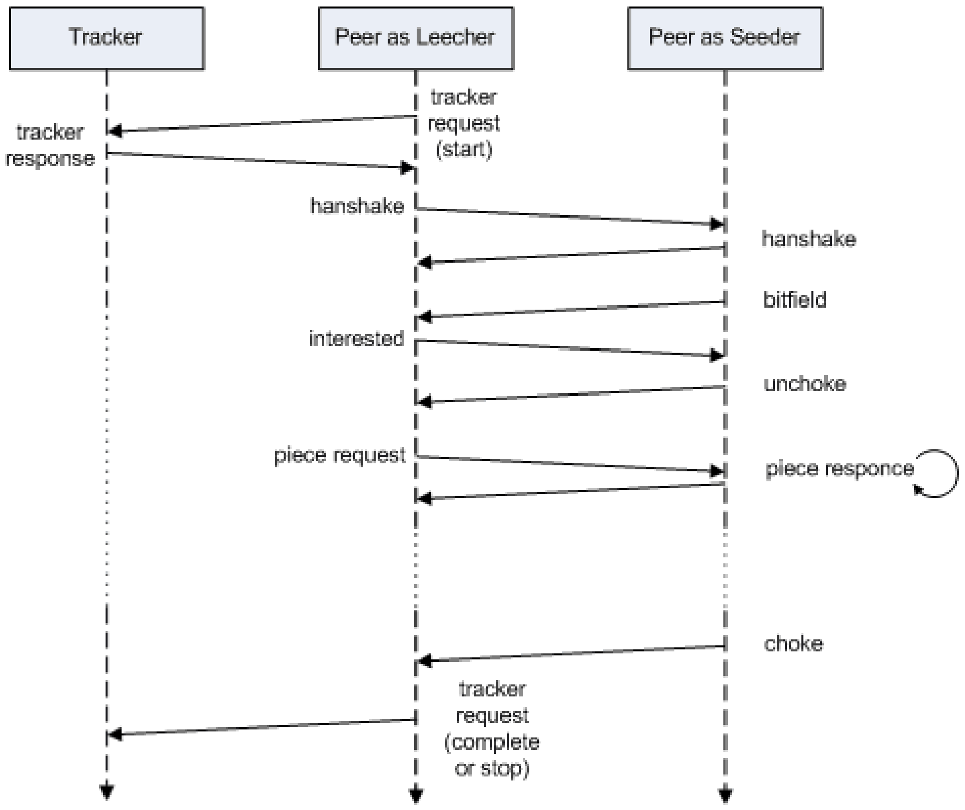


Figure 2: Message flow in BitTorrent: It shows the sequence of the registration at a tracker and further the download procedure with a remote peer.

this example. The leechers calls with an *handshake* message this seeders. The seeder responses with the *handshake* and the *bitfield*. The leecher answered on it with the *interested* message and the seeder will unchoke the new peer. Then the normal download will start, therefore the leecher requests a piece and the seeders sent this piece. This procedure is in that case a loop. In the end the leecher stops, once it has all pieces or the user has shut down the program and/or download, the leecher peer sends a new request to the tracker with the current event (stopped or download).
The second figure shows (see Figure 3) the torrent distribution between two leechers and one seeder. In this figure the communication with the tracker is not shown. The transaction between the first leecher with the seeder is identical to the example above. The second leecher starts its work after the other so that the first leecher has some pieces. Further, the second leechers call the seeder and the first leecher with the *handshake* message. Then both peers answer with the handshake response and the *bitfield*. So far

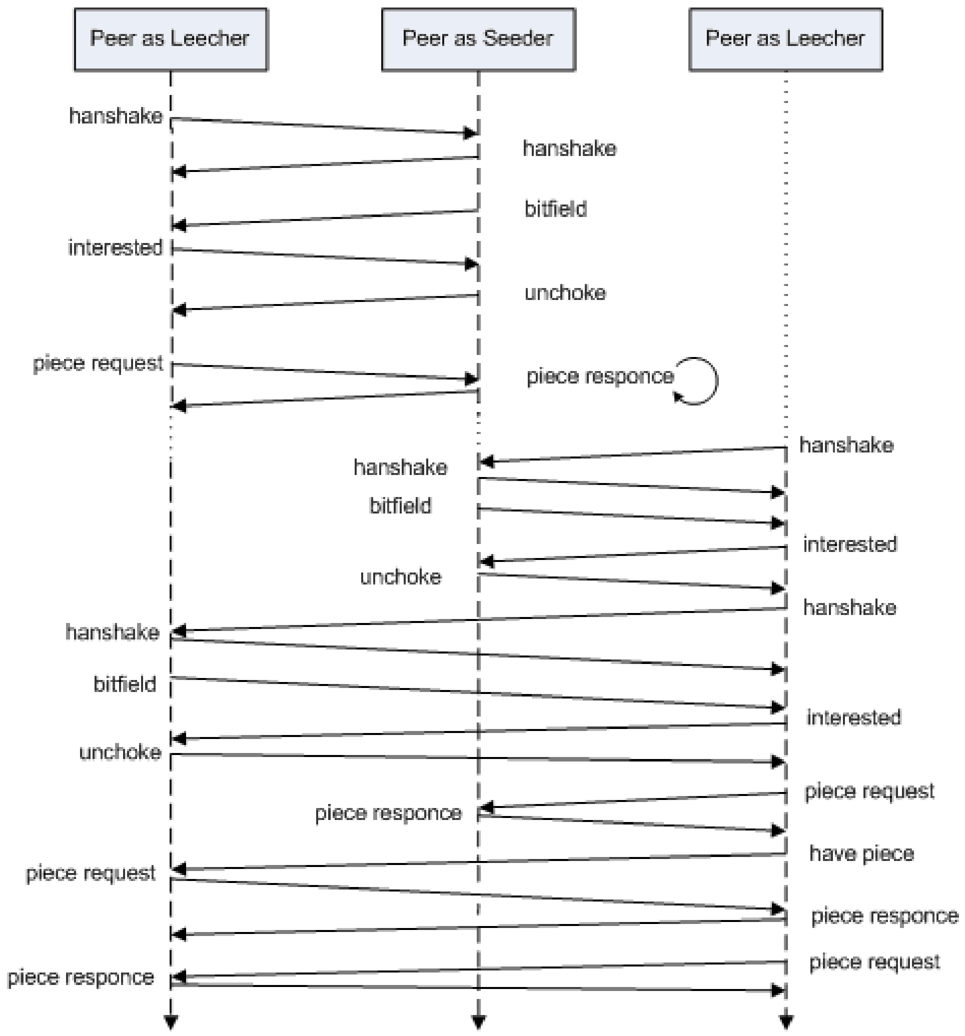


Figure 3: Message flow by BitTorrent: It shows a sequence of action between two leechers and one seeder.

it was the same procedure as before. Now, the second leecher requests a piece which the first does not have, because the request of a piece index is random. If the second one downloads that piece completely it offer it to the first leecher with the *have* message. And now the first leechers can download pieces from the second one.

3. The BitThief Client

The Distributed Computing Group[5] of the ETH Zurich, Institute of Computer Engineering and Networks Laboratory (TIK)[6], has developed its own *BitTorrent* client which is called *BitThief*. The client is written in Java and is based on the official implementation[7] (written in Python, also referred to an official client or mainline client), and the Azureus[8] implementation.
The *BitThief* client was deployed for scientific analysis of the BitTorrent protocol. With this client was shown that it is possible to manipulate the protocol. The result of it is a higher download rate as all other peers in the swarm with an unfair behavior. This client does not upload any pieces to other peers. Note it was not deployed for downloading illegal material, it was deployed to show that the original protocol is not fair.

3.1. Functionality of BitThief

The *BitThief* implementation is as simple as possible and contains a lot of analyzing tools to measure the performance of the client. Also different techniques were implemented to improve the download rate.
BitThief does not perform any chokes or unchokes of remote peers, and it never announces any pieces. In other words, a remote peer always assumes that it interacts with a newly arrived peer that has just started downloading. Compared to the official client, *BitThief* is more aggressive during the startup period, as it re-announces itself to the tracker in order to get many remote peer addresses as quickly as possible. The tracker typically responds with 50 peer addresses per announcement. This parameter can be increased to at most 200 in the announce request, but most trackers will trim the list to a limit of 50. The tracker announcements are repeated at an interval received in the first announce response, usually in the order of once every 1800 seconds [2]. The *BitThief* client ignores this number and ask the tracker more frequently. The client does not impose a limit on the number of active connections, in fact, it tries very aggressively to open as many connections as possible. This is achieved by querying the tracker more often for peer addresses. This frequently polling improves the knowledge about the swarm rapidly.

[5]`http://www.dcg.ethz.ch/`
[6]`http://www.tik.ethz.ch/`
[7]`http://www.bittorrent.com/`
[8]`http://azureus.sourceforge.net/`

If you have a large number of open connections it improves the download rate in two ways. On the one hand, connecting to more seeders allows the client to benefit more often from their round-robin unchoking periods. And on the other hand, there will be more leechers in the neighborhood that include *BitThief* in their periodical optimistic unchoke slot. Opening more connections increases the download rate linearly. The remote peers operate independently of the number of the open connections by *BitThief*.

3.2. How BitThief Exploits the BitTorrent Protocol

A peer which receives a sub-piece cannot verify the integrity of it. The reason is the torrent metafile contains only the hashes for whole pieces. The peer therefore has to fetch the whole piece first and can then decide whether the piece is valid or not. Usually the client cannot determine which peer has transmit bad pieces. This makes the protocol insecure when there are peers uploading random garbage.
This serves as an idea for an attack, but it is to say that uploading garbage is not an interesting option anymore, because most client implementations are good at handling this exploit. Instead of pure free riding without uploading any data, *BitThief* considers a more relaxed form of free riding without uploading any valid data. That is the main idea by *BitThief* for free riding. By uploading random garbage the client can cheat the remote peer into thinking that it is a good neighbor who uploads data quickly and are thus unchoked more frequently which enables *BitThief* to download at a high speed in return. Unfortunately, a lot of BitTorrent clients try to download all sub-pieces of a piece from the same peer and hence discover quickly that *BitThief* is cheating. This causes a block of the client IP address for a couple of hours or even days, depending on the implementation.
In trying to improve this idea, *BitThief* has implemented a mechanism which prevents uploading a whole piece to a remote peer. This way, the peer needs to fetch the remaining sub-piece from another node and will in the end not be able to detect which peer sent the garbage data. Unfortunately, this did not work out either, if the peer connection stalled because *BitThief* did not answer certain sub-piece requests [7].

3.3. BitThief will be a Fair Client

The next generation of *BitThief* (version 1.0.0) will implement a new transmission strategy which guarantee a real fair data exchange. Therefore a new protocol is

developed by the DCG[9] which performs a *tit-for-tat* strategy. To avoid unfairness, *tit-for-tat* is the most efficient strategy to enforce collaboration among selfish users.
In a real *tit-for-tat* environment, a peer is only exchange data with another peer if it continues to give something in return. Regrettably, this is often impossible in a file sharing environment, because a newly joining peer inherently has no content to share and therefore it is not able to upload and so to collaborate with other peers. This is the first problem of pure *tit-for-tat* for decentralized and dynamic environments, such as P2P networks, so the peers can rapidly end up in a deadlock situation.
Another problem with pure *tit-for-tat* in file sharing applications is that a peer's neighbors sometimes have nothing to offer which the peer does not already have, this situation block out any exchanges. Therefore, real *tit-for-tat* is not a suitable strategy to guarantee fairness in dynamic, distributed systems.
To avoid this situations and to use the advantage of *tit-for-tat* the DCG developed a way of implementation for *tit-for-tat* strategy which is described in [8]. Now I will summarize the important points of their strategy.

3.3.1. Idea of Tit-For-Tat

The strategy from authors of [8] is based on source coding. Let us assume that different peers are interested in the same file f provided by one or more peers. The file f is divided into m data blocks $b_1, \ldots, b_m$ which are used as the atomic units of trade. The peers still in the process of downloading are commonly referred as leechers and the peers which provided the whole file are called seeders. In the beginning one seeder must provide a block set for a leecher, while existing several seeders, it is useful that one seeder send the first blocks for free. Note that the first blocks are not the first data blocks of the file and these data blocks are not shared directly. The seeders encode them first and provide these to the leechers.

Each data block is considered to be a sequence of elements from a certain alphabet. These elements are elements from a *Galois field* $GF(2^q)$ where q denotes the length in bits of each element in the sequence. A *Galois field*[10] or a finite field is a field that contains only finitely many elements. Finite fields are important in number theory, algebraic geometry, Galois theory, cryptography, and coding theory. The *Galois field* is

[9]Distributed Computing Group of the ETH-Zurich http://dcg.ethz.ch/
[10]The field named in honor of Évariste Galois.

defined on the basic operations addition, substraction, multiplying, and division.
All computations are carried out in a finite field modulo a *Mersenne prime number*[11]. The *Mersenne prime number* is a sub-quantity of the prime numbers with the property that currently only 45 numbers are known. An advantage of *Mersenne prime numbers* is that these numbers are almost a power of two. This characteristic of the basic algebra operations $(+, -, \times, \div)$ which is used in a *Galois Field* is fast and thereby efficient. Suppose the data block size is 128KB and the *Mersenne prime number* is $2^{31} - 1$, then each element of the finite field consists of a sequence of 33,825 elements, each element has a length of 31 bits. When performing mathematic operations on data blocks, the operations are performed on all the elements in the blocks separately.

Seeders never distribute data block directly, they compute a linear combination of k blocks and offered these blocks. k has a defined size and the blocks are chosen randomly. The authors of [8] have analyzed different size of k and they have detect that a good size of k is $k = \log(m) + 2$. A leecher downloads such a block. Secondary, the leechers need to download a vector which contains information about which blocks have been added up to build this new block. This strategy, for small random k, beware the peers for deadlock situations. To reconstruct the original file, a leecher has to download all m encoded blocks which are all build from different linear combinations of the original file. The file can be recovered by solving a linear system of equations.
The exchange between leechers are fair using *tit-for-tat* because every leecher must provide blocks to get blocks itself.

3.3.2. Seeder Strategy

Peers pin their hopes on the goodness of seeders to become blocks to start downloading. Therefore the seeders need a suitable strategy which helps the leechers on the one hand, but on the other hand they have to prevent freeloading behavior. The seeder strategy is based on a strategy whose primary purpose is to boost the download progress by providing newcomers with initial data they can share.
The Allowed Fast message of BitTorrent Fast Extension mechanism defines for any peer a small but specific pseudo-random set of data blocks that this peer can download immediately for free.

[11]A Mersenne prime number P is a prime number for which it holds that there is a number $x \in \mathbb{N}$ such that $P = 2^x - 1$.

Allowed Fast is an advisory message which means "if you ask for this piece, I will give it to you even if you are choked." Allowed Fast thus shortens during which the peer obtains occasional optimistic unchokes but cannot sufficiently reciprocate to remain unchoked. The *Fast Extension* mechanism is a modification of the previous original protocol and is also implemented in the mainline client. It modifies the semantics of the *Request*, *Choke*, *Unchoke*, and *Cancel* messages, and further it adds a *Reject Request* [9]. Furthermore, it adds more interesting features, for example it specifies an algorithm to calculate a piece subset which is based on the Class C IP-Network address where the remote peer is in. The pieces which are in the set can be downloaded for free by the remote peer. This can improve the bootstrapping of new peers which enter the swarm. The new nodes get a start set of pieces, which is chosen randomly, so they can start downloading.

This mechanism has a big problem for fair trade, because the peer can collect all blocks without uploading any blocks in return. The set of data blocks are build in according to the peers IP address and therefore all set are equal for a specific requesting peer by different seeders.
The system from the authors of [8] adopts the idea to provide an initial pseudo-random set of encoded blocks to all peers. In contrast to the original Fast Extension mechanism, only the seeders compute such sets of blocks and they are the only ones giving out blocks from such sets. The reason why only seeders can do this and not the leechers is due to the nature of a leecher, because the leechers do not possess all blocks and therefore they cannot compute a random set of blocks. If a leecher connects to a seeder, the seeder will announce these downloadable linear combinations. Another modification of the original mechanism is that the seeders have the skill to modify the size of the allowed subset of linear combination at run time. The size depends on the number of peers in the network. The authors write that the number of peers can be set to $\min\{m, \max\{\alpha, \beta \times \frac{m}{n}\}\}$, where n is the number of the peers in the network and $\alpha \ \ and \ \ \beta$ are small positiv constants greater than 1.

This strategy of *tit-for-tat* has some advantage in contrast to the original *BitTorrent* implementation. At first, the most peers have different IP addresses which guarantees many combinations of different encoded blocks. That ensures that the available number of distinct linear combination in the swarm is large. Another point is, the leechers are forced to collaborate, because the seeders merely provide a small and specific set of blocks to each leecher and refuse to provide any other blocks. The point is, any leechers should be interested to upload to other leechers as much as possible which uses the bandwidth of the network more efficiently. On the other hand, the seeder can stay in the network and save its own upload capacity and therefore the seeder save costs. A further advantage is the size of variable the free block set: The system is resiliency to freeloaders increases as the network grows. Furthermore, if there are only a few interested peers, each of them gets a reasonable share of the file for free so that they can all finish their downloads by exchanging their blocks amongst each other [8]. One disadvantage of the new system is that the complexity and computing time increases. A further disadvantage is for asymmetric Internet connections like ADSL, because with this technique the upload capacity is smaller than the download capacity. This reason decreases the upload and therefore it has much influence on the whole download time. But also the original BitTorrent protocol has this problem as well as all other peer-to-peer techniques.

4. State of BitThief

In this section I show the state of *BitThief* version *0.1.7*, it should be shown on which point I start. The GUI is very simple and has implemented only a few features. The program at this stage consists of three windows. If you execute the program the first window opens. That is shown in Figure 4. This window is very plain and supports only the load option for a new torrent metafile.

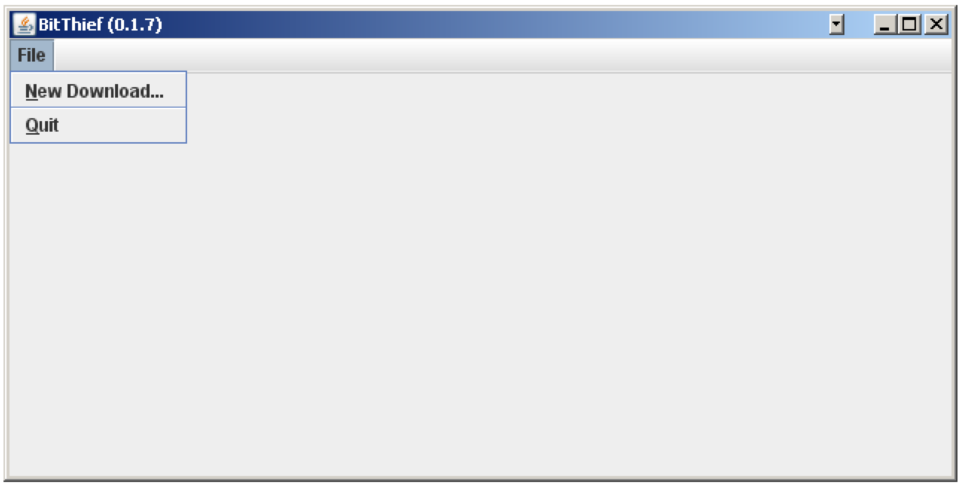
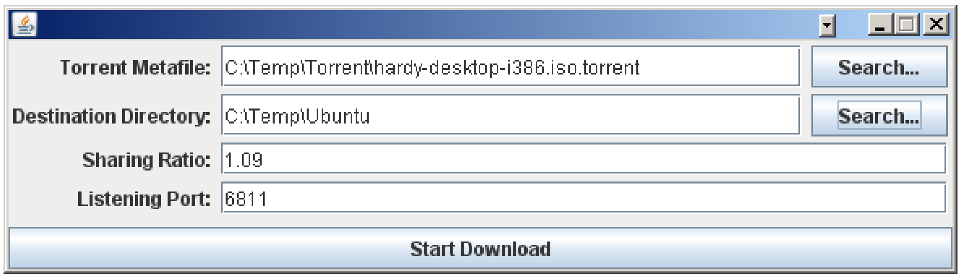


Figure 4: BitThief 0.1.7: Start screen.

If you select "New Download...", which is the natural way, the next window shows a dialog where you can choose the torrent metafile and the destination directory where the download should be stored. Also the window presents the listening port and a calculated random sharing ratio. This window is shown in Figure 5.



Torrent Metafile: C:\Temp\Torrent\hardy-desktop-i386.iso.torrent

Destination Directory: C:\Temp\Ubuntu

Sharing Ratio: 1.09

Listening Port: 6811

Figure 5: BitThief 0.1.7: Choose a torrent metafile and destination directory.

If the metafile and the destination directory is set, the download can start. This is shown in the next window, Figure 6. It comprises two options, the stop download option, "clear", and the option to show the details for analyzing the current download session.

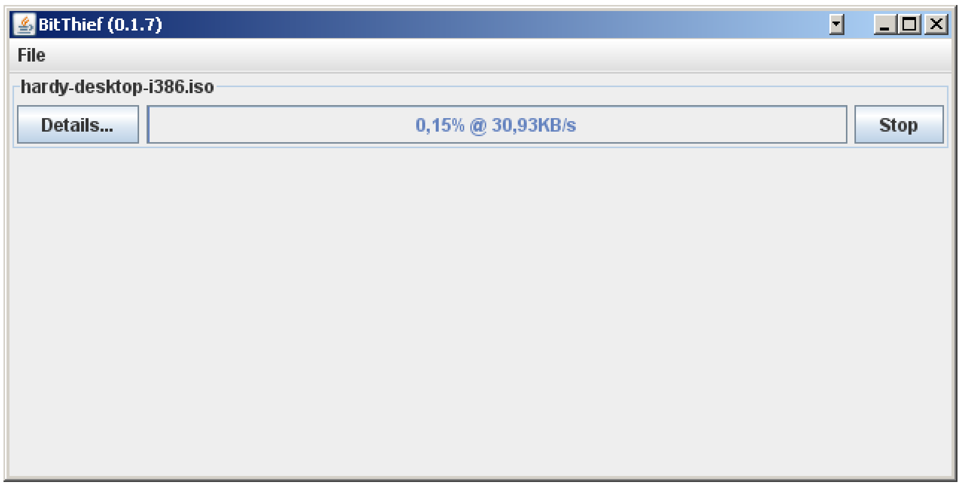


Figure 6: BitThief 0.1.7: Download started.

The last window of the current version is the details window. There you can see different tabs for analyzing and reporting. This is used for different comparisons between the original implementations of *BitTorrent*, like Azureus[12] and others, and the *BitThief* client. The details window is shown in Figure 7.

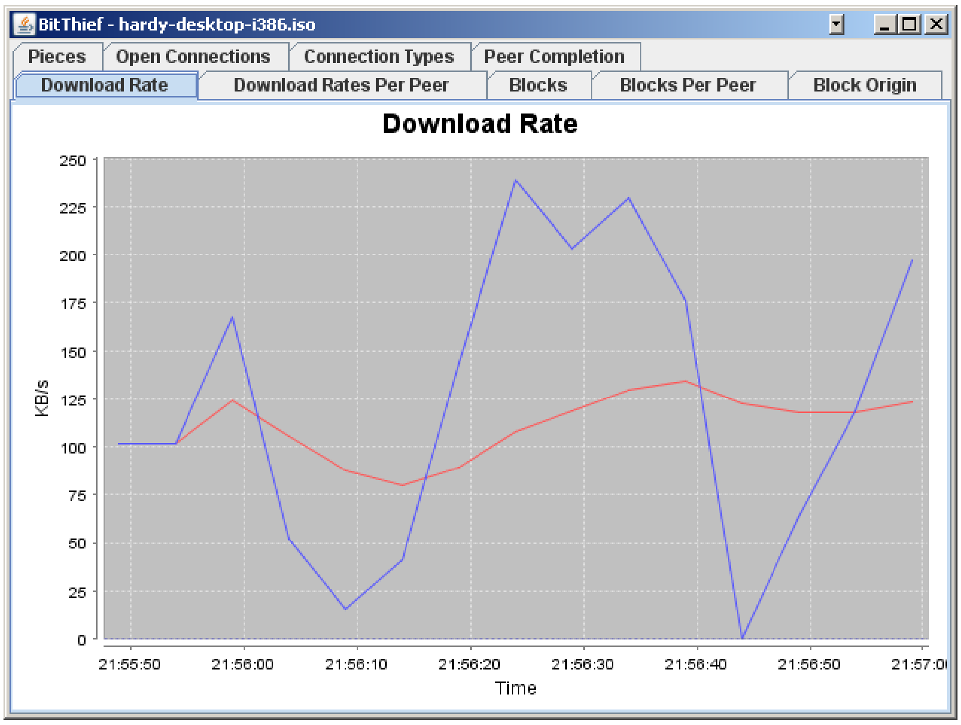


Figure 7: BitThief 0.1.7: Details of the current download.

[12]http://azureus.sourceforge.net/

5. Esthetic surgery

In this chapter is described the new user interface of the *BitThief* client. This chapter and all others are concerned with the current version *0.2.0* of the *BitThief* client.

5.1. Motivation

The version *0.1.7* of *BitThief* has a very simple user interface where only the basic functions are implemented (load and start a torrent, show download details). Also, all system parameters can be set only by a call from the command line. That is unattractive for many users and to approach more users to use this client, instead of any other BitTorrent clients, *BitThief* need a nicer and/or a more friendly user interface. Further, *BitThief* will be used by a greater community and this increase the accuracy for the statistical analysis which is important for the scientific evaluation.

5.2. Concept

A friendly GUI has to be intuitive, so that the user is familiar with the program in short time. Also, it needs well known components (functionalities) and features to control the program.
What is important for a user-friendly graphical user interface? A GUI has to provide standard and advanced hot-keys e.g., save, load, close and further functionalities. This hot-keys allow to control the program in a quick modality from the keyboard. Also for a good recognition of different features a program needs icons, which represent the functionalities behind. Furthermore, all menu titles are named in a short form what could be interpreted in another way by users, so it is useful to describe all titles more in detail. Therefore, tool-tips are very helpfully, this guarantees that the title is not too long, and it describes the functionality behind.
Nowadays, many users like to control a program only with the mouse and they hate it to navigate in the file system to select a wanted file, so it is important that new programs implement a drag and drop functionality to pull a wanted file in the program.

What is important for a professional program interface further? Exclusive of special features a program needs a window where the user can set all important program options which the programmer provide for customer settings. Further, common practice is a window to show the program name including the current program version, the developers of the project and the homepage of the project if it exists. This window is mostly called an "About" window.

5.3. Implementation

First, I have implemented two kinds of hot-keys where the user can call program functions via the keyboard. A kind of hot-keys uses the *CTRL* key plus a character key. Thereby, it was to heed using of standard keys for elementary functions like *CTRL Q* for closing the program, *CTRL O* for open a file (load torrents or others files), and *CTRL S* for save operations. The main window has no save operation and so the *CTRL S* key is free to use for the search function, the *CTRL U* key is used to call the update function manually. I choose this values because it is intuitive for these functions. Only the important functions are call via *CTRL*. The *CTRL* hot-keys call the functions directly. The second kind of hot-key uses the *ALT* key. All functions in the menu support this short-key. The used characters are underlined, so the user knows which combination is to use to call a wanted function. For example, to show the *About* window following combination is used *ALT A B*. To open the *About* menu *ALT A* is used and then *B* to open the window, because the name or title is *BitThief*. To describe each function more in detail each menu title consists of a tool-tip description which informs the user about this function.
All menus consist of an icon for graphically description of the function. All icons have a license for free developing. This is important to guarantee that the client is abidance by the law.
The hot-keys, tool-tips and icons are implemented in different windows in the same manner as described. This elements should increase the main window as well other windows to.

The new main window is shown in Figure 8. In the new version of the window, it includes current downloads with a progress bar and if it is selected the details window with all charts of the selected torrent. Also the menu-bar was improved, it implements all functions which are available for the client (Search Torrents, Subscription Service for Torrents, the Updater, the Option menu as well the "About BitThief" window and the control keys for the select download, which shows details, start, pause, and remove). Further, the window supports a drag and drop functionality to load a new torrent in a

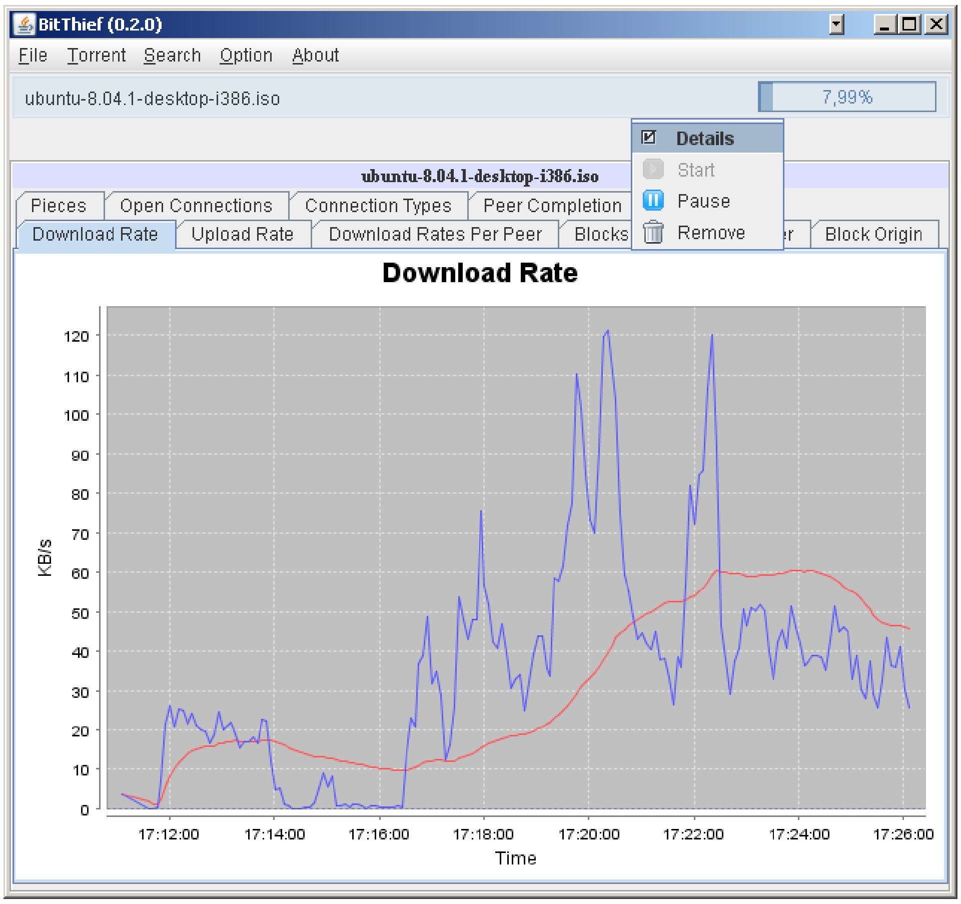
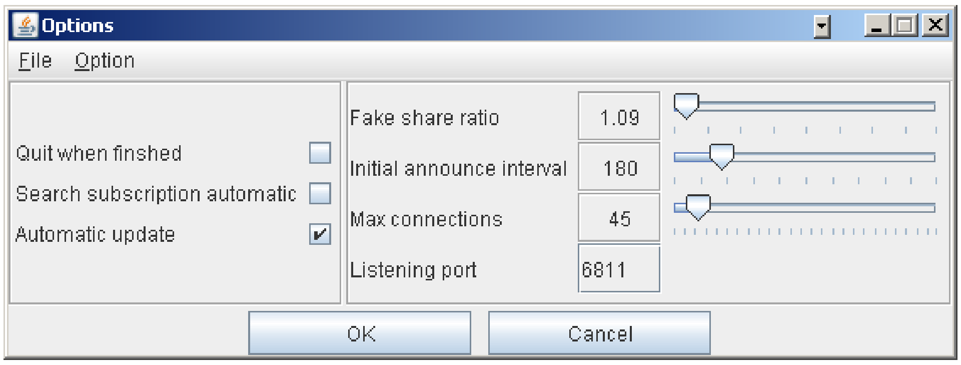


Figure 8: BitThief 0.2.0: The new main window design.

quick way.
The window size is variable, the maximum size is the screen dimension. For this purpose the program ask the operating system about the current display resolution. The minimal size depends on the currently selected torrents and the details container if it is shown. It calculates the minimal screen size, given the count of current torrents plus the detail window height, because the minimal size must guarantee that the detail container do not overlap the torrents container and so the window cuts off some information.

In the next step, the options for the client were implemented in the GUI. This window is shown in Figure 9. On the left hand side are the boolean options with a checkbox including their description and on the right hand side are the editable values. The most values can be changed about a slide-bar to help users to set the right values and further, which avoid falsely input. Only the listing port uses an editable text-field. Because 65535 different ports exist and so it is impractical to choice one of them per slide-bar or combo-box. This field needs a special failure detection to check if the input is a number or not, and to check the range. Did the user a mistake, a message informs him when the port is out of range or if the input is not real number. Also, all values per slide-bar are checked of range failures. In case of users fake the input, e.g., the listening port is set but not committed per enter, the program set the default value.



Quit when finshed ☐
Search subscription automatic ☐
Automatic update ✓

Fake share ratio 1.09
Initial announce interval 180
Max connections 45
Listening port 6811

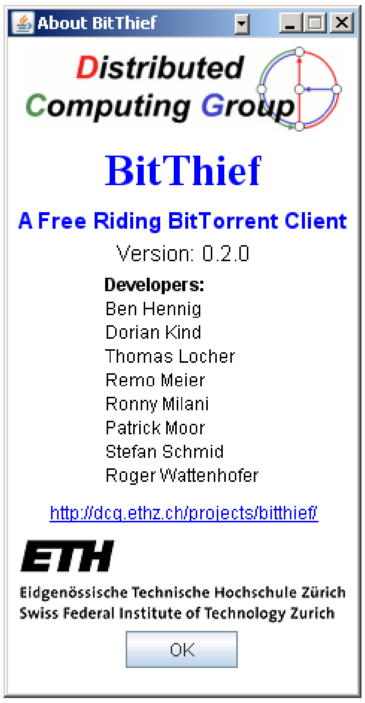
Figure 9: BitThief 0.2.0: The new option window.

The user has the option to save and load different setting files via the file menu and he can also restore the default values. To set the default values the user has to push the menus under *Option*. The "Option window" itself can be called from the main window.

Another important point of a "professional" program interface is to show a window with program name, version, developers and a link to the project homepage this is an "About" window and is shown in Figure 10.
The window will be created dynamically at run time. The developers were read in from a special file as well as the URL from the homepage. The window lists all developers one after other and at the end it calculates the window height. With the shown URL (blue underline) starts the default browser from the system (except Unix/Linux) and the project homepage is shown. Under Unix/Linux, the program looks for an existing browser and starts it, because Java 1.5 has no support for the default browser.
All other windows are shown in the specific chapter.

5.4. Testing

The implementation of GUI elements is not too difficult, but it needs a lot of testing time. *BitThief* must run on most popular OSes (e.g. Windows XP/Vista, Linux Gnome/KDE and MAC OS), that is the reason why *BitThief* is written in Java. Suppose Java is system independent and in general it is true, but in practice, e.g., with GUI functions



About BitThief

Distributed Computing Group

BitThief

A Free Riding BitTorrent Client

Version: 0.2.0

Developers:
Ben Hennig
Dorian Kind
Thomas Locher
Remo Meier
Ronny Milani
Patrick Moor
Stefan Schmid
Roger Wattenhofer

http://dcg.ethz.ch/projects/bitthief/

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

OK

Figure 10: BitThief 0.2.0: "About" window shows project name, version, developers and project homepage.

it is easy to show that the interpretation of the OSes are different. Now I will explain some short examples.
The window-managers for Linux interpret the window width differently than Windows. The result is the content in the window cuts through the border under Linux. The solution is easy, if the window size will be set then ask about the OS before and if it is Linux, add a size constant (e.g. 25 pixel, note the constant depends on the elements which are used).
The drag and drop feature from Java is a beautiful tool but the behavior is different for all window managers. If you load a new torrent file per drag and drop, then you have different effects in different systems. For Example KDE 3.5 recognizes another URL-string as Gnome. That means you have to implement different string filter for any OS or window manager.
The last example is the mouse click behavior. Basically Window and Linux/KDE are a double-click system and Linux/Gnome is a single-click system. The reason is that Gnome has implemented the standard mouse click behavior from X11 (Linux/Unix window manager). To solve this problem the programmer must release a mouse event for the correct interpretation.
The GUI was tested under Windows XP/Vista, Linux KDE 3.5/KDE 4.1 and Gnome

with the Ubuntu distribution Dapper Drake and Hardy Heron (both LTS versions), Open Solares 2008.05 (Unix based on System V), and MAC OS 10.5 (Leopard, Unix based on BSD). In conclusion, GUI programming for system independent programs need a lot of testing time.

6. Update Function

In this section I have analyzed and designed an update function for the *BitThief* client.

6.1. Motivation

The current published version of *BitThief* (version 0.1.7) is available on the project home page[13]. Nowadays, all users of the *BitThief* community have to look on this site if a new version published. To avoid this scenario in later versions an update function is require. Also this functionality should guarantee in future that most users works with the latest version of *BitThief* what is important for statistical analysis of the client. To avoid that users look for newer versions, an automatic update function is useful which detect the new version.

6.2. Concept

In this section I will describe some concepts for an update function. For an update function you have to think about following points.

- Distribution technique
  - distribution via P2P or standard ways
  - incremental or full update
- How can it detect a new update
- How it should works
- What happens in a failure case

6.2.1. Distribution Technique

At first I have analyzed which distribution technique I can use for the client, because it is a P2P client and it could be distributed itself via BitTorrent protocol. This would be a very clever way, but *BitThief* has a big disadvantage for this idea it cannot send data to other nodes (at the current version 0.2.0). So it is only the standard way via FTP or HTTP from the homepage and/or server. Although the question about full

[13] `http://dcg.ethz.ch/projects/bitthief/`

or incremental update is easy to solve. The client is one *.jar* file, that means it is a compressed executable format from *Java* which include all program files. For this kind of program file it is better to distribute the whole program instead of incremental parts.

6.2.2. Detect Update

How can *BitThief* detect an update? For this aim many solutions exist. One solution is to parse the website and look for the current version which is published on the web. This way is insecure because it is easy to fake the content on a website. A better way is to store a *"version.properties"* file in a hidden directory. This file can be downloaded and then the client can read out the existing version from the web. If a newer version is available the program informs the user.

6.2.3. How it should Works

If a new version is available the client looks for a checksum and the link for the new version. After this step the client create a copy of itself and stores it under a new name. This step is fundamental in the failure case in order to go back to the old version. The client has restored itself and can start with the update function. If the program has all data, it computes a checksum and compares it with the published sum on the web and overwrites itself on the hard disc. In the case that both sums are equal, the current program is launching the new version and then it terminates itself. If the new version has started it looks for an old version file and then removes it.

6.2.4. Failure Case

If the update function detects a failure, because the checksums are not equal, then the client follows a backward strategy. That means that it informs the user about the unsuccessful update and restores the current *.jar* file from the existing copy. After this procedure the old version is running.

6.3. Class Model

My concept consists of two classes, the main class *BitThiefUpdater* and the *MDGenerator* class. Figure 11 show the class diagram for the update function. The main

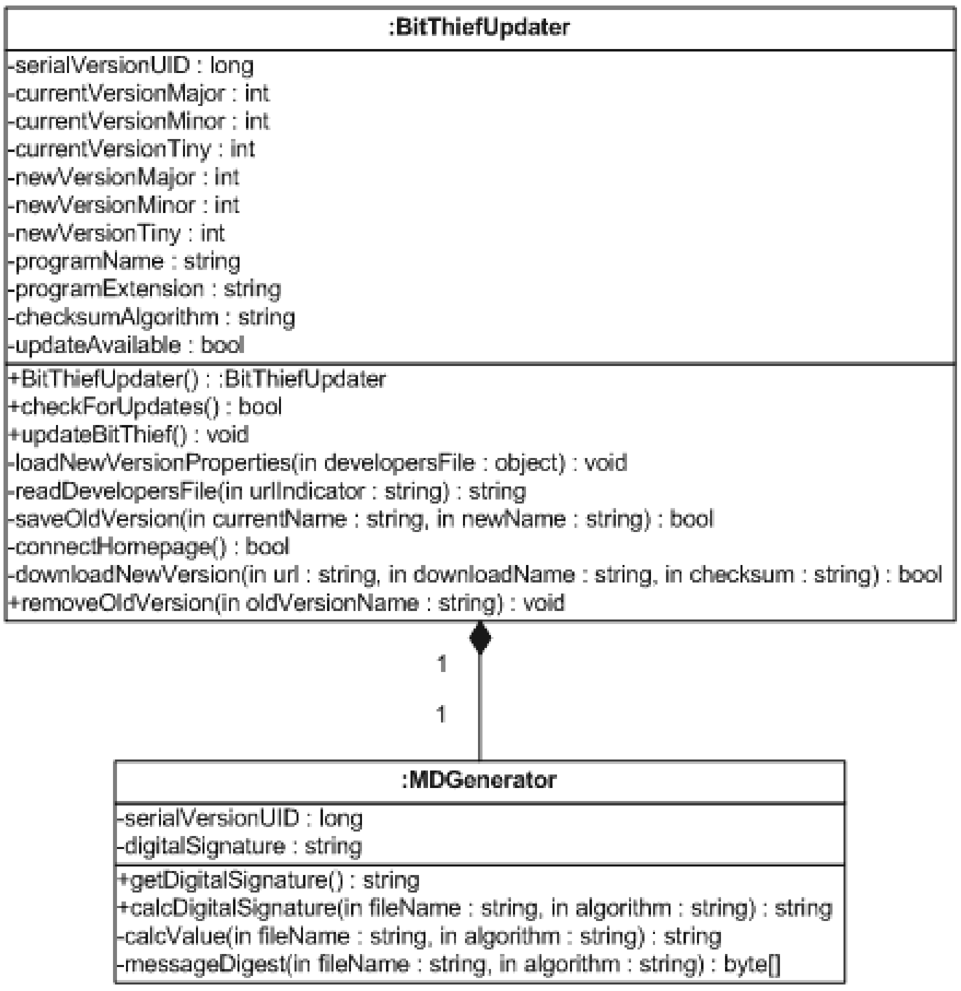

:BitThiefUpdater
-serialVersionUID : long
-currentVersionMajor : int
-currentVersionMinor : int
-currentVersionTiny : int
-newVersionMajor : int
-newVersionMinor : int
-newVersionTiny : int
-programName : string
-programExtension : string
-checksumAlgorithm : string
-updateAvailable : bool
+BitThiefUpdater() : :BitThiefUpdater
+checkForUpdates() : bool
+updateBitThief() : void
-loadNewVersionProperties(in developersFile : object) : void
-readDevelopersFile(in urlIndicator : string) : string
-saveOldVersion(in currentName : string, in newName : string) : bool
-connectHomepage() : bool
-downloadNewVersion(in url : string, in downloadName : string, in checksum : string) : bool
+removeOldVersion(in oldVersionName : string) : void
1
1
:MDGenerator
-serialVersionUID : long
-digitalSignature : string
+getDigitalSignature() : string
+calcDigitalSignature(in fileName : string, in algorithm : string) : string
-calcValue(in fileName : string, in algorithm : string) : string
-messageDigest(in fileName : string, in algorithm : string) : byte[]

Figure 11: Class diagram for the update function.

class, *BitThiefUpdater*, includes all communication, download and compare functions. It consists of three public function for the controlling from outside (*checkForUpdates*, *updateBitThief* and *removeOldVersion*). The first function which is called, except the constructor, is *checkForUpdates* which returns a boolean value. This function is downloading the *version.properties* file. In order to extract the current version, which is currently published, the private *loadNewVersionProperties* function is used. After that the *checkForUpdates* routine compares the versions number and if the version from the web is greater, the function returns *true* else *false*. Had the updater detected a new version, the *updateBitThief* function would have been called. This function has no

return value but it throws an exception in the failure case which can be caught by the caller. At first the *updateBitThief* function creates a copy of itself, therefore it uses the *saveOldVersion* function. Then, the updater calls the next function, *connectHomepage*, this function uses the *readDevelopersFile* which extracts the download URL from the *developers.properties* file. If it has the destination URL, the function parses the web site for the checksum string which is only published there. The string is used to recognize a failure by downloading the new client version. Now all is prepared for the real download. This is performed by the *downloadNewVersion* function. For the download part I use a special library from Apache[14] (commons-httpclient-3.1.jar). This library is more robust and clearly faster than the standard Java tools. The *downloadNewVersion* is a boolean function which returns false in a failure case else it returns true. To recognize a failure during the update the updater compares the checksums which comes from the web site on the one hand and on the other hand it calculates a new value. The returned value are guided through *connectHomepage* to the *updateBitThief* function. There it will be evaluated and is it an error then the backward strategy will be performed. For this purpose the function restores the old version, overrides the failed file and throws an exception to inform the user.
For the error detection I used the second class, *MDGenerator*, which represents a checksum generator to calculate the hash sum. It can be used for different algorithm, e.g., MD5 or SHA-1. To start the generator the updater calls the public *calcDigitalSignature* function. The updater commits the destination file name and the algorithm which should be used. *calcDigitalSignature* calls the next function *calcValue* and this calls the generator procedure *messageDigest*. *calcValue* is a help function because *messageDigest* returns a byte array and this array must be converted to a string, this is the main task of the *calcValue* function. The result string of the hash sum is returned to the caller guide through *calcDigitalSignature*. There also exists a getter function for returning the hash sum in later cases.

6.4. Implementation

The update function is a singular part of this program and could be implemented in other Java programs with minimal changes. Figure 12 shows a program plan of procedure for my concept of the update function.

[14] http://hc.apache.org/

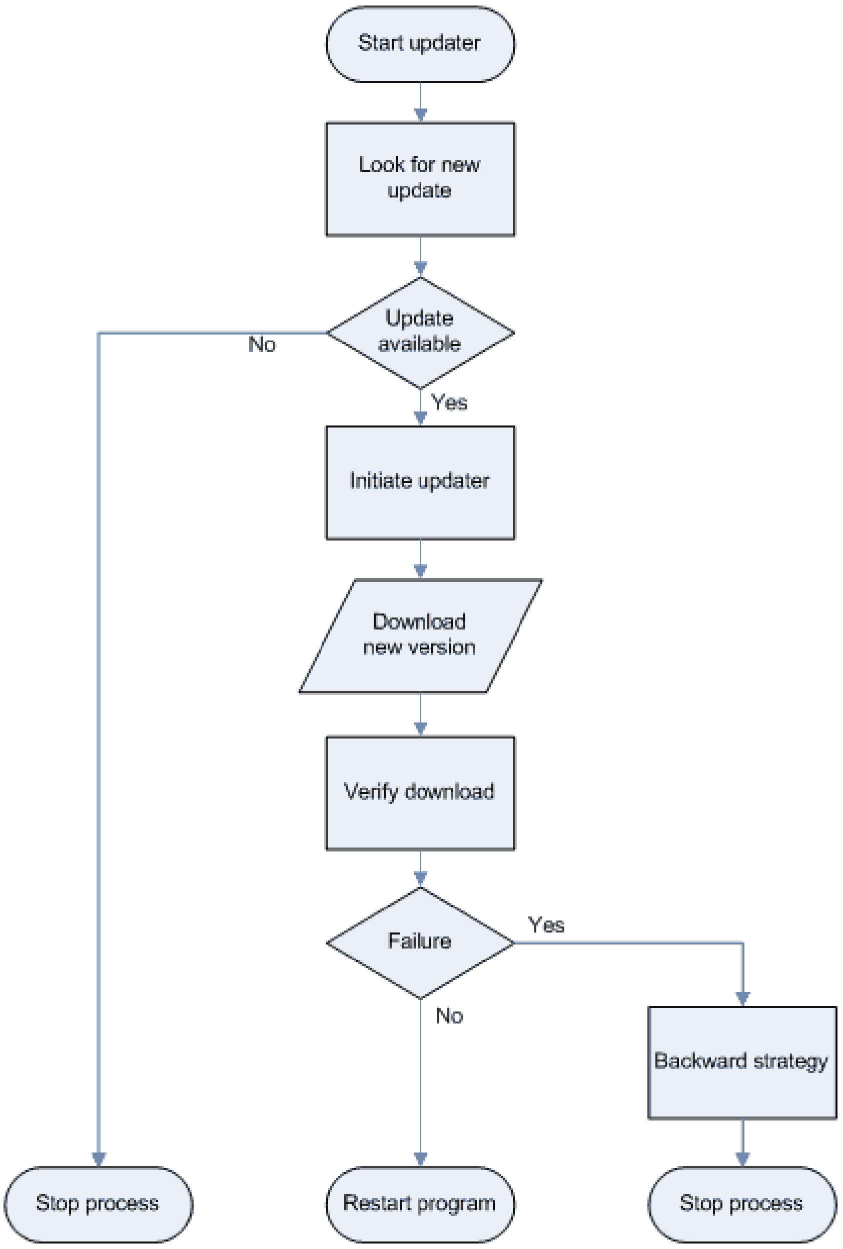


Figure 12: Program plan of procedure for the update function.

The client downloads the *"version.properties"* file from the web and saves this file under the name *NewVersion.porperties* in the "bitthief" folder in the home directory of the user. Then it compares the new version with the existing version as described before. When the updater is started it create the copy of itself with the name *BitThiefOld.jar*. To avoid failure of this process the function calculates both checksums and compares these values. Now the preparation is done and the client can update the source file. If the download is complete, the client has verified the new program version and this version is running now, cleanup routine will be performed. This routine looks for the *NewVersion.porperties* and the *BitThiefOld.jar* and removes these parts. In Figure 13 a sequence diagram is shown which clarified the procedure. The diagram shows the case

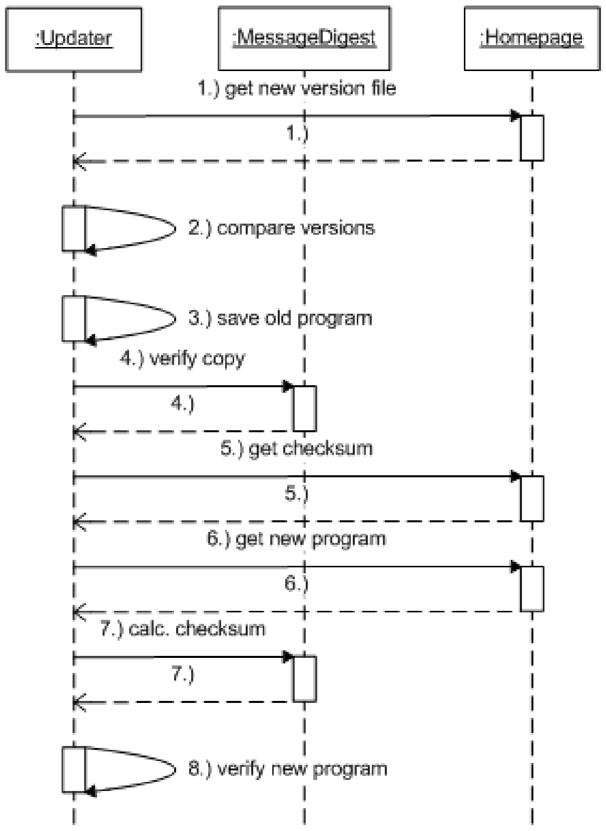


Figure 13: Sequence diagram for the update function, download a new program version.

where a new update is available and no failure is detected.
In case the client detects a failure on the checksums, then the backwards strategy will be started. The client restores its version from the *BitThiefOld.jar*. At the same time the user gets an error message for the crashed update and the client performs the cleanup routine.
The updater is implement in two kinds, automatic or manual. If the user select the automatic option in the option menu (set by default) then the client looks for a new version at launch. If the user has deselected this option then the user can use the update function in the "option" menu-bar from the main window.

6.5. Testing

The version where the updater is implemented is *version 0.2.0*, the latest published version on the web is *0.1.7*. The challenge is to test all parts of the update function. Therefore, I performed different scenarios. In the first scenario I tested the recognition of a newer version with downloading and restarting the client. For this purpose I changed my own *version.properties* file to a lower version as on the web (e.g., version *0.1.0*) and then I created a new *.jar* file. Note this test was also performed with other combinations like *1.0.0*, *0.0.1* etc.. With all combinations the update worked well.
In the second scenario I tried to test the cleanup function, which should remove the old client. I have performed the first scenario and then I have overridden the downloaded file by hand with a newer version, because all downloaded versions are *0.1.7*. This test ran well as well.
In the last scenario I tested the error detection. This was a little bit harder because I have to simulate network failure and the network in Universities mostly works very well. For this test I have tried to cut the network connection by removing the LAN cable.
In conclusion I can say that the updater works well.

7. Multi-Torrent Selection

In this chapter I describe a concept and my implementation of a multi-torrent selection dialog. At first I will explain the basic of multi-torrents and then I introduce the concept and the implementation.

7.1. Basic

A multi-torrent file is a file which describes more than one download file like a music album (included covers and many music titles) or program packages which are not distributed as *iso image* or in compressed form (e.g., zip file). A torrent is split into pieces with a constant size and these pieces are split into many blocks, also with a fixed size. The piece size is defined in the torrent header and the block size is always fixed (16 KByte). This constant size defines a natural bound. But these limits are not the bounds of a single file because a single file can be larger or smaller than the bounds in a piece and/or block. This problem is illustrated in Figure 14. These properties mean that it

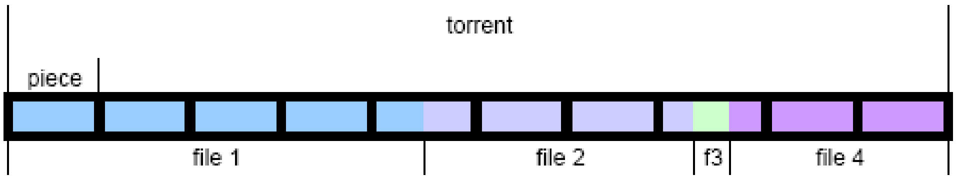


Figure 14: Multi torrent: properties between the block size and the file bounds.

must download more pieces and/or blocks than needed. In the worst case it would be that it has to download all pieces if the single elements are small (smaller than the piece size or a little bit larger as a piece).
The fragmentation of pieces in blocks has the advantage that smaller packages will be sent, which can increase the throughput in the network and further it can receive the content from more nodes.

7.2. Concept

First, it is to be determined if the current torrent is a multi or a single torrent. If it is a single torrent, then the process starts normally. When it is recognized that a multi-torrent was chosen then show all elements in a selection dialog. The user can deselect all unwanted files and then the user start the download process. Note before the real download is started the program has to calculate the needed and/or unneeded pieces also it must save the unused file names. To mark the needed and/or unneeded pieces there are two different strategies. Both strategies have the same program plan of procedures, which is shown in Figure 15.

The first strategy is to save the unneeded pieces in a file and declare these pieces as

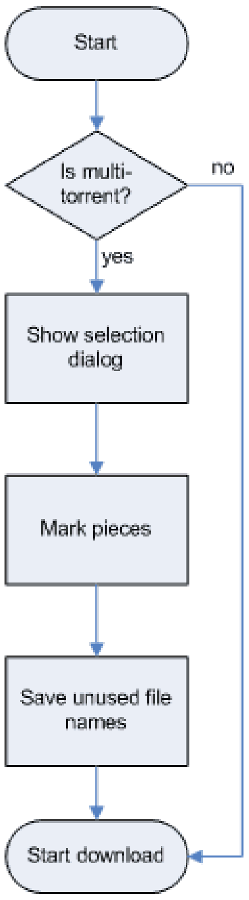


Figure 15: Program plan of procedure for the multi-torrent function.

valid so that the client thinks that these pieces are already downloaded. To save this pieces the program has to calculate *the single file size* divided by *piece size*. The result is the *piece count*. To get the current *piece index* it must only add the piece counts from the files before adding the current piece count. The challenge when calculating the *piece count* is that the result is mostly a floating value, but the piece index is an integer. Therefore, it must at first round down the value and then it can mark these pieces as valid. After this step, it has to round up the value for the next calculation. This rounding procedure should guarantee that it does not override a piece which is used by a file that is significantly smaller than the piece size. To mark the pieces as valid has a big disadvantage, it does not work with fair BitTorrent clients because they want to

upload these pieces to other peers and have the problem that the pieces do not exist. Hence, another strategy is needed.
The second approach is to turn around the first one. That means that it saves these pieces which are needed for our download. This makes it possible working with a fair distributing client.
Both strategies have the same calculation cost, but the order of rounding is vice versa. The cost refers only to the mathematical algorithm, in reality it depends on the wanted pieces or/and unwanted pieces.

7.3. Class Model

Figure 16 shows my class model for the multi-torrent selection. There exist three different classes *MultiTorrentSelection* class, *MetaInfo*, and *BitField* class. The *BitField* class is the environment which handles all pieces of the current torrent download. The

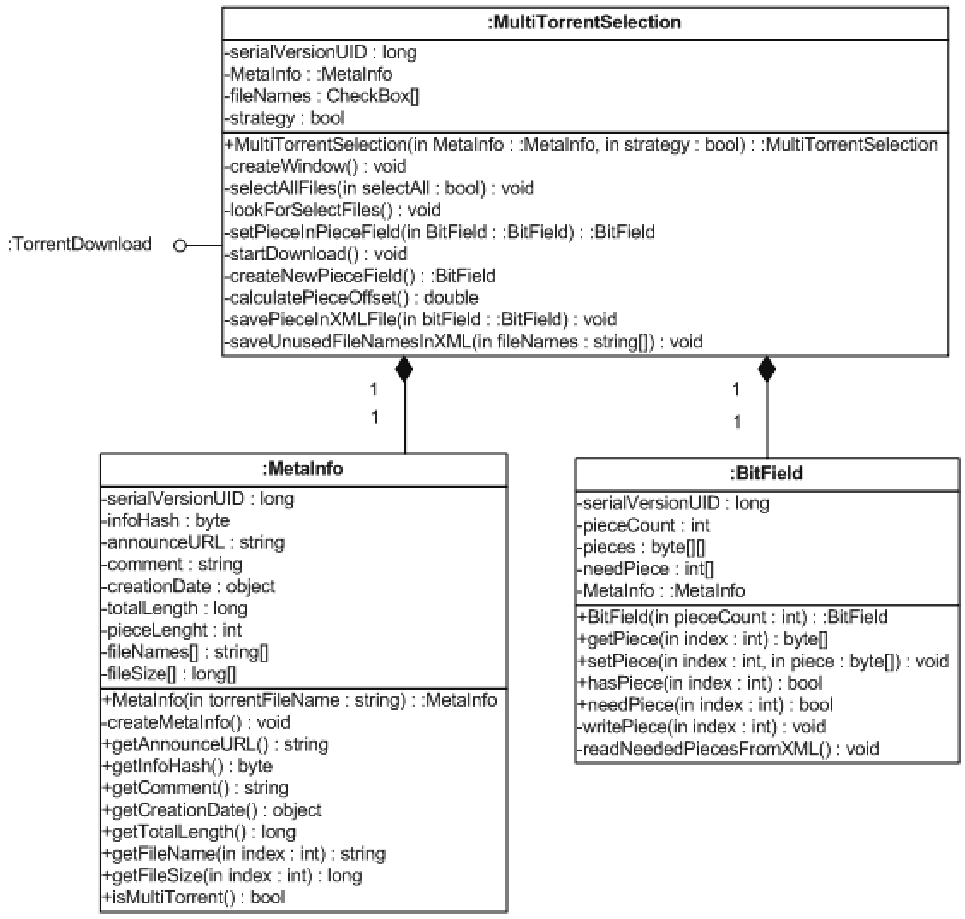

:MultiTorrentSelection
-serialVersionUID : long
-MetaInfo : :MetaInfo
-fileNames : CheckBox[]
-strategy : bool
+MultiTorrentSelection(in MetaInfo : :MetaInfo, in strategy : bool) : :MultiTorrentSelection
-createWindow() : void
-selectAllFiles(in selectAll : bool) : void
-lookForSelectFiles() : void
-setPieceInPieceField(in BitField : :BitField) : :BitField
-startDownload() : void
-createNewPieceField() : :BitField
-calculatePieceOffset() : double
-savePieceInXMLFile(in bitField : :BitField) : void
-saveUnusedFileNamesInXML(in fileNames : string[]) : void
:TorrentDownload
1
1
1
1
:MetaInfo
-serialVersionUID : long
-infoHash : byte
-announceURL : string
-comment : string
-creationDate : object
-totalLength : long
-pieceLenght : int
-fileNames[] : string[]
-fileSize[] : long[]
+MetaInfo(in torrentFileName : string) : :MetaInfo
-createMetaInfo() : void
+getAnnounceURL() : string
+getInfoHash() : byte
+getComment() : string
+getCreationDate() : object
+getTotalLength() : long
+getFileName(in index : int) : string
+getFileSize(in index : int) : long
+isMultiTorrent() : bool
:BitField
-serialVersionUID : long
-pieceCount : int
-pieces : byte[][]
-needPiece : int[]
-MetaInfo : :MetaInfo
+BitField(in pieceCount : int) : :BitField
+getPiece(in index : int) : byte[]
+setPiece(in index : int, in piece : byte[]) : void
+hasPiece(in index : int) : bool
+needPiece(in index : int) : bool
-writePiece(in index : int) : void
-readNeededPiecesFromXML() : void

Figure 16: Class model for the multi torrent selection.

MetaInfo class is build after selecting a torrent file. In this class, all information of the torrent such as title, file(s) with size, piece length, and the total length is extracted. In case the torrent describes more than one download file then it is a multi-torrent file. The *MultiTorrentSelection* class needs at beginning the *MetaInfo* class. On the basis of the file names the selection dialog window is created. Once the dialog is shown, the user can deselect the unwanted files. At the beginning all files are selected.

When the user commits, the *lookForSelectFiles* routine is called. In this routine, it will initialize a *BitField* (array) where the pieces are marked. Therefore, it calculates the piece index and marks these. At the same time, the function stores the used and unused file names in a string array. If the function is done, all piece indices are saved in a XML file for this torrent. Also, the used and unused file names are saved in extra XML files. All file names consist of the hash value of the current torrent, which guarantees that the names can be clearly identified if more torrents exist. To save the pieces and the file names in files has the advantage that the download can restart if the client crashed or the user stopped the download. If the user removed this torrent from the client, all files will be deleted. Now the multi-torrent function has completed all task and starts the download class. This class looks for the piece XML file and read in the piece indices and if the download is done then it reads in the unneeded file names to remove these files from the directory. Note that the download routine is not in the diagram, if I drew all classes the diagram would be too complex.
The implemented class model in *BitThief* is nearly the model that is described in Figure 16. I have simplified the model to show the core concept of this function. It was not useful to show all classes in the model because the class model of the whole *BitThief* project which are connected for the download process is too complex. In the original *BitThief* implementation there exist more "helper classes" and also all GUI elements with their initialization and control functions.

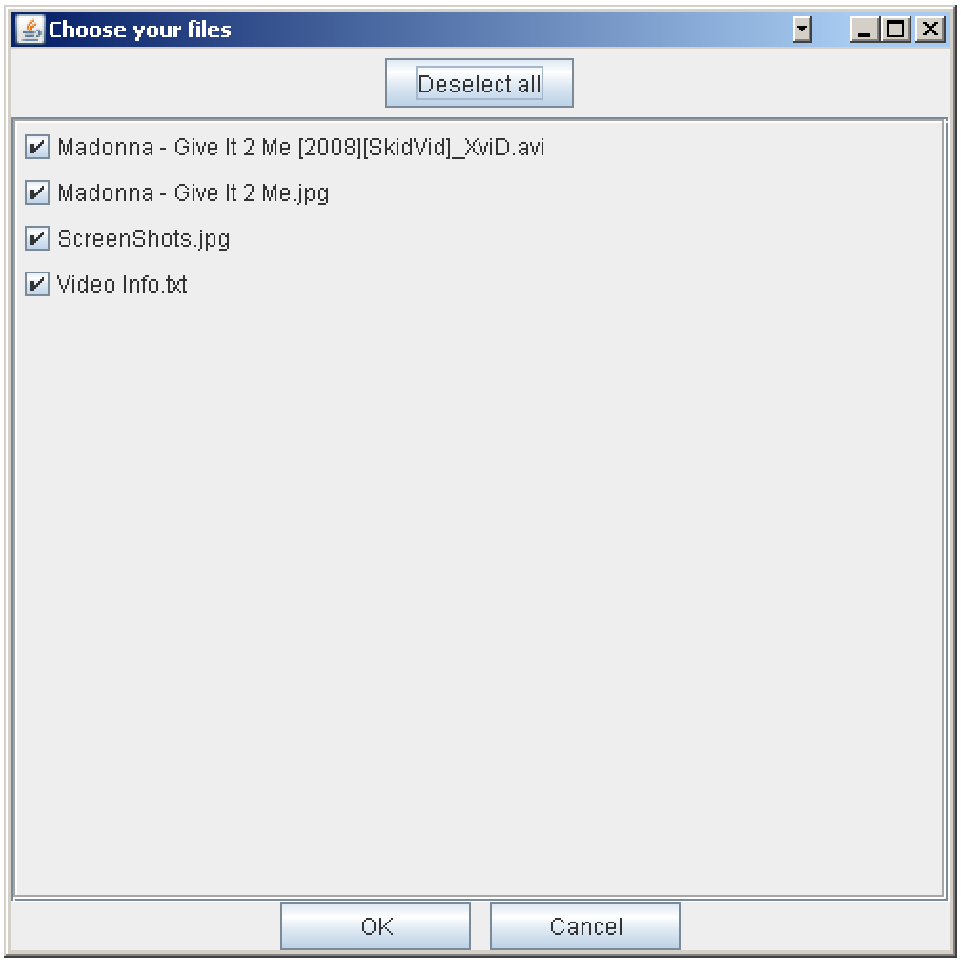
7.4. Implementation

The first implementation of the multi-torrent selection dialog uses the strategy where the pieces not needed for the download are marked. The reason is in version 0.2.0 of *BitThief* an environment does not exist to distinguish which pieces are needed and which therefore are wanted and thus it was simpler to mark all pieces which are unneeded as valid. That means the client is manipulated and thinks to have these pieces. In the current version, this strategy is harmful because this version never

uploads anything to other clients.
To guarantee that the multi-torrent selection works with a fair protocol I have implemented the second strategy which selects only the needed pieces. This implementation was harder because it must implement further functions to distinguish between having this piece and still needing this piece. The reason is, if the client looks for the next needed piece it asks itself if this piece exists. If the piece does not exist then download it and at this point the client must know if the piece is wanted. Therefore, I implemented an index variable for each piece. To save a little bit of memory, the variable is an integer value and has three different states. If the piece is unneeded for the download, the value is smaller zero (-1), if the piece is wanted but not currently downloaded, the value is equal to zero and if the value is greater than zero (1) the piece exists and was downloaded before.
Now the client works with both strategies which can be selected by a switch. The modification has no influence on the real performance and the user of *BitThief* will not notice this change.

The aim of the selection window for multi-torrent is making the handling as simple as possible for the user. This aim should guarantee an intuitive service. The window is shown in Figure 17 including an example of a multi-torrent file, which presents a Madonna video clip with covers and a description file. Note that the torrent was chosen, because the characteristic of the including files are very good for testing (one large file and the rest files are smaller than a piece).

The window consists of three buttons, a *Select/Deselect all*, an *OK*, and a *Cancel* button. The *OK* button calls the *lookForSelectFile* function, which calculates the used and/or unused pieces. If that is done the real download process can begin and the window is closed. In the following listings, Listing 3, Listing 4 and Listing 5 an example is shown where only the second and fourth file will be downloaded. Listing 3 represents the file where the unused file names are saved. Listing 4 shows all files which are needed for the download process, because the needed pieces overlap the boundaries. In case of that a file is unneeded during the download process, the client knows that and so no memory will be allocated for that file. And Listing 5 shows which pieces are needed for the wanted files. This example is performed with the strategy for fair clients.



☑ Madonna - Give It 2 Me [2008][SkidVid]_XviD.avi
☑ Madonna - Give It 2 Me.jpg
☑ ScreenShots.jpg
☑ Video Info.txt

Figure 17: BitThief 0.2.0: Multi torrent selection window inclusive an example torrent.

```
<?xml version="1.0" encoding="UTF-8"?>
<torrent_download>
  <torrentName>Madonna - Give It 2 Me [2008][SkidVid_XviD]</
     torrentName>
  <filesCount>2</filesCount>
    <fileName>Madonna - Give It 2 Me [2008][SkidVid]_XviD.avi</
       fileName>
    <fileName>ScreenShots.jpg</fileName>
</torrent_download>
```

Listing 3: Example file which saves the unwanted file names.

```
<?xml version="1.0" encoding="UTF-8"?>
<torrent_download>
  <torrentName>Madonna - Give It 2 Me [2008][SkidVid_XviD]</
```

```
     torrentName>
  <filesCount>4</filesCount>
    <fileName>Madonna - Give It 2 Me [2008][SkidVid]_XviD.avi</
       fileName>
    <fileName>Madonna - Give It 2 Me.jpg</fileName>
    <fileName>ScreenShots.jpg</fileName>
    <fileName>Video Info.txt</fileName>
</torrent_download>
```

Listing 4: Example file which saves the needed file names for the download.

```
<?xml version="1.0" encoding="UTF-8"?>
<torrent_download>
  <name>Madonna - Give It 2 Me [2008][SkidVid_XviD]</name>
  <need_pieces>
    <piece>279</piece>
    <piece>280</piece>
  </need_pieces>
 </torrent_download>
```

Listing 5: Example file which shows the needed pieces for the current download.

The *Cancel* button closes the window and sets the allocated storage free. For a simplified handling there is the *Select/Deselect all* button, by default all files are selected and the button shows the *Deselect* name. This default setting is chosen because it can be assumed that the user will download all files in most cases. If the button is clicked, the name changes to the opposite state. With that button the user can select or deselect all files so that a user who wants for example one file or only few he must not click every file. Every file has a check box for selection they are all ordered in a check box array. If the box contains the check mark than the file is chosen and the client will download it. Otherwise, if the box has no check mark, the file will not be downloaded.
The window is constructed in a dynamic manner, which means the window size can be set by the user and if the file description is larger than the view port a scroll bar for horizontal and vertical adjustments is shown.

7.5. Testing

I have tested different scenarios with several torrent files. At first I have tested the recognition of multi-torrents and single-torrents, therefore I used an Ubuntu torrent

which is a single torrent consisting of an ISO image. And for multi-torrents I used music albums from several artists as well as complete seasons of TV shows and videos. The recognition works well and the window is only shown if it is a multi-torrent file.
In the next step I tested if the download works well with single elements (not the whole content). For this purpose I selected every second file from a multi-torrent and then I controlled the beginning and the end of the file. For this test it was feasible to use a music album, because these files are not too large and the controlling is easy and fast.
In the last test scenario I simulated the worst case (described in section 7.1). This was a little bit harder because the challenge was to find a torrent which consists of some small files. Useful for that test was the torrent of a music clip which is shown in Figure 17. The whole torrent is split into 281 pieces, whereas the first file (video clip) is using approximately 279 pieces and a little bit of the next piece. So the last three files are split into two pieces. File 2 has 63.404 Bytes and file 4 has 2.206 Bytes, which is significantly smaller than the piece size with 262.144 Bytes. File 3 has 205.782 Bytes which is also less than the piece size but this file has the property to use the last pieces which means that this file overlaps the piece bound. With this three files I have performed many tests. At first I have selected every single file with the result that for file 2 and 4 only one piece is downloaded. Both pieces of file 3 are downloaded. Then I selected file 2 and 4 together and also both pieces are downloaded. This test was useful to guarantee that the multi-torrent selection works well.
I have tested both strategies with the same test scenarios and both strategies work well. What I could not test is the behavior when working with a fair protocol, which means that the *BitThief* client shares its pieces with other clients, because a fair protocol was at this time not implemented.

8. Search Torrents

In this chapter, I describe a search function for torrents. The aim of this function is to give the *BitThief* client a new feature which does not current exist in other *BitTorrent* clients and which make the client slightly independent from a browser. The search function should increase the useability of *BitThief*.

8.1. Motivation

The motivation of the search function on the one hand is to increase the usability and to give the client a special feature. On the other hand, many torrents exist which are corrupted or dead. Death means in this case that no seeders are available for the torrent and therefore it is not feasible to get the whole content. Because it can be assumed if no seeders exist that the current leechers in the swarm hold only a part of the content and so the download cannot be finished. Hence it is meaningful to implement a ranking service for good torrents and also a strategy to avoid bad torrents.

8.2. Concept

The concept of the search function is a simple search mask and/or dialog for any provider from which the user can choose. There the user can select a provider, select the search option (e.g. all, films, music, books, games etc.) and an input field for the search word. To guarantee an equal search mask a translation function for the search options is necessary. If all parameter are set, the function connects to the chosen website, commits the search request and waits for the reply. The responses from the provider are always in the HTML format as well the search request must be converted in to the HTML format. To convert the response, a HTML parser is required. The parser transmits the converted results in a data table and provides it for the GUI. A simplified program plan of procedures is shown in Figure 18 which will be concretized later. Before the GUI fetches the data table, it will remove all dead torrents (having zero seeders) from the content, also some corrupted torrents will be removed. Example for this is that no size is available which means that values inside of the seeder or leecher cells are characters instead of numbers, once the data table is adjusted, the GUI can call it.

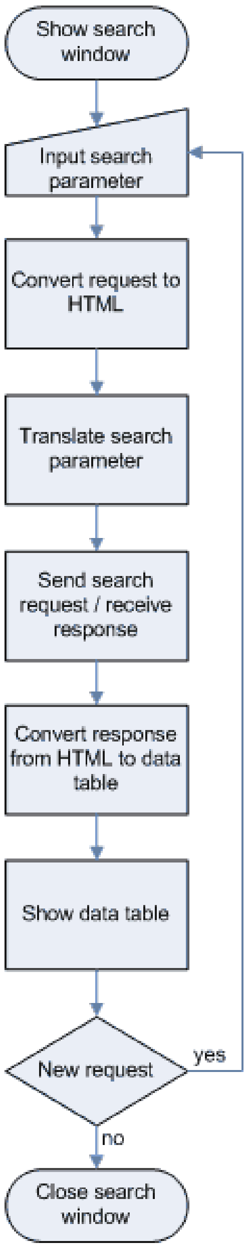


Figure 18: Simplified program plan of procedures for a search function.

8.2.1. Concept of the GUI

The GUI presents the search results in a table with the following columns: Name, Date, Type, Seeds, Leechers, Size, Comments, Download, CommentsURL, Comments-Count, Thanks-Count, Torrent-URL and Index. Only the first eight columns are shown ("Name" to "Download") because the last five columns are hidden and will be needed for background operation (e.g., torrent ranking, the torrent URL and, if they are available, comments from other users). The "Comments" and "Download" column contain a

separate button. With these buttons the user can download the torrent file, and the start dialog for the real download is shown, where the user can choose the destination folder for the download. When clicking the "Comment" button user comments mentioned for that download file should be shown, if no comments are available a short dialog is shown which informs the user that no comments are available.
The data table can be (too) large so that the user needs a sort function for every column. This function has to handle different data formats like strings, date, integers and strings with numbers, therefore a special function is useful.

8.2.2. Concept of Ranking

The first strategy to avoid corrupted or dead torrents was described before. But for a better hit rate it is not enough. Therefore, it needs a better strategy which can guarantees a good torrent. The basic requirement for ranking is that the torrent provider supply information for the torrent which can be analyzed by the program. Now I will explain two different approaches.

At first I will describe a simple idea, where the "Thanks" are analyzed. For that the count of "Thanks" is the indicator. The torrents with the highest count is the best hit, because it can be assumed that a fake or bad torrent has no "Thanks". The advantage of this strategy is it needs not much sophisticated intelligence by the program. Furthermore many torrent provider supply this information by a search request and/or search response. This avoid additional requests, saves costs and is very fast.

The second idea uses the comments. The count of comments is not an indicator for good or bad torrent. Because users can write anything in a comment good and bad or unimportant stuff. Thus a good parser is needed which is able to analyze the comments and then to create a hit rate. The challenge for the parser is to recognize the right information which is important, for example, it is a fake or not. To look for a *keyword* like "fake" is not too hard, harder is to implement a mechanism for all kinds of the negation ("is not" or "isn't" etc.). Harder is the languages, because who says that the language of the comment is English, it could be written in German, Spanish or in the worst case Chinese or Arabic. Note worst case based on the letters (or signs) in Chinese and Arabic.
Another problem is the handling of two comments, where the first says it is a fake and the second says the opposite. Then both comments are in balance and no ranking is

possible.
It is feasible to avoid the problems with the linguistic feature. Most users in the web added to the written text Smiley icons to show their emotion. It is easier to use these icons for the ranking than the language, because the variety is not too large. But this approach has the problem with the balance of good and bad smiles also. In addition, some users write comments about a mistake in the name of the torrent name or they expected something else by the title. These facts destroy any statistic for a good ranking. A further disadvantage are the network cost, because in the worst case, every single torrent has a comment, the program floods the provider with requests in very short time with a good connection. Otherwise, with a bad network connection the time which is needed to create the ranking is long.

8.3. Class Model

In this section I describe the class model for the search function. Figure 19 shows the conceptual class model of this function. For simplification I split the model into two

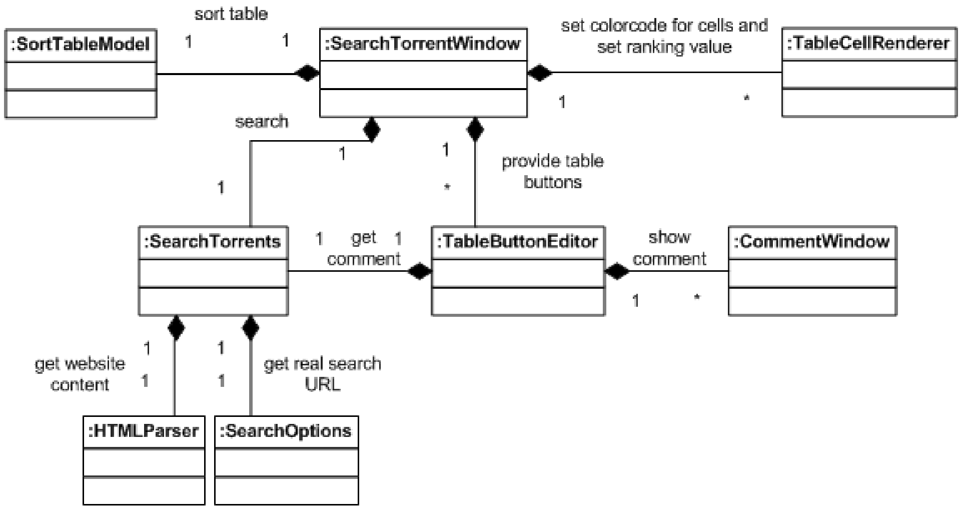


Figure 19: The conceptual class model for the search function.

diagrams. The first model describes the real search function and the second model describes the user interface.

8.3.1. Class Model for Searching

The search function consists of three classes, a *SearchTorrents*, *SearchOptions*, and the *HTMLParser* class. The main class is the *SearchTorrents* class, the other two are utility classes. This class model is shown in Figure 20.

The caller from *SearchTorrents* delivers the following information from the selected

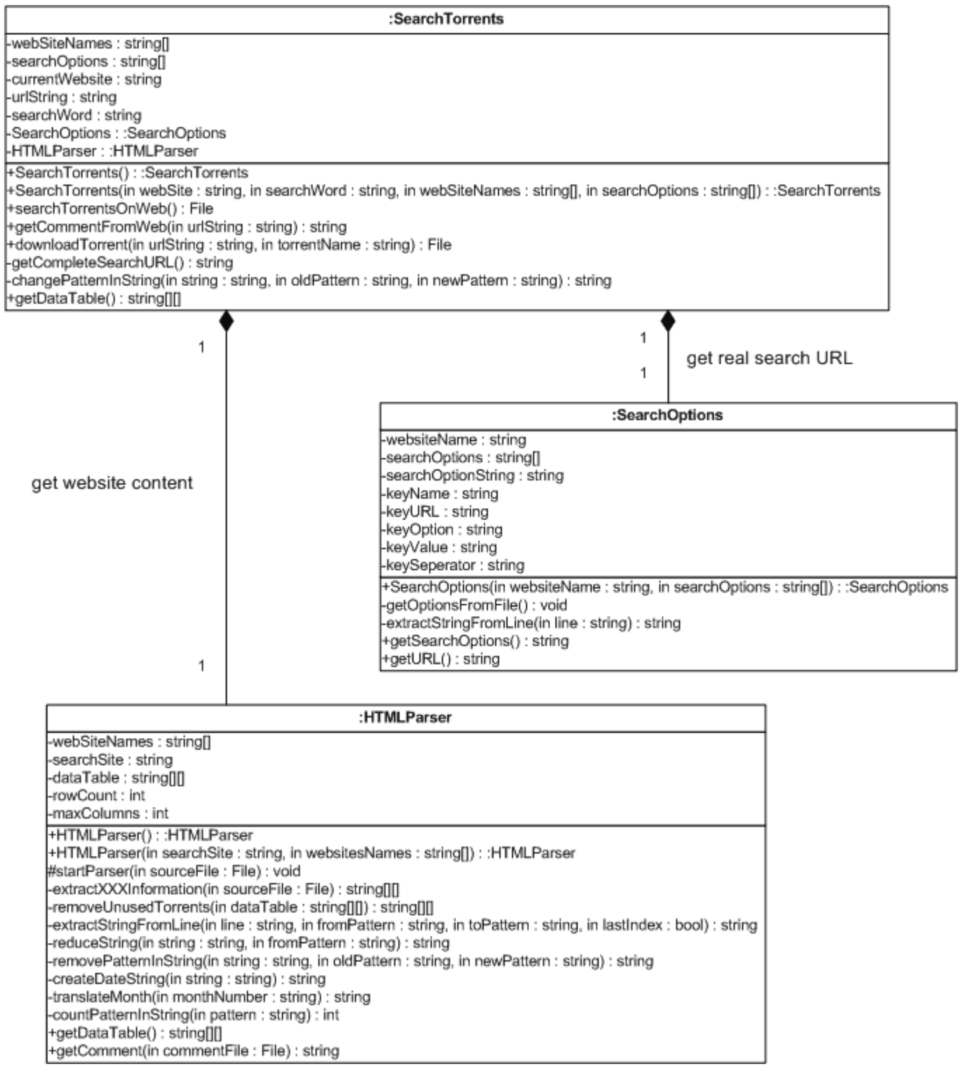

:SearchTorrents
-webSiteNames : string[]
-searchOptions : string[]
-currentWebsite : string
-urlString : string
-searchWord : string
-SearchOptions : :SearchOptions
-HTMLParser : :HTMLParser
+SearchTorrents() : :SearchTorrents
+SearchTorrents(in webSite : string, in searchWord : string, in webSiteNames : string[], in searchOptions : string[]) : :SearchTorrents
+searchTorrentsOnWeb() : File
+getCommentFromWeb(in urlString : string) : string
+downloadTorrent(in urlString : string, in torrentName : string) : File
-getCompleteSearchURL() : string
-changePatternInString(in string : string, in oldPattern : string, in newPattern : string) : string
+getDataTable() : string[][]
1
1
1
get real search URL
get website content
:SearchOptions
-websiteName : string
-searchOptions : string[]
-searchOptionString : string
-keyName : string
-keyURL : string
-keyOption : string
-keyValue : string
-keySeperator : string
+SearchOptions(in websiteName : string, in searchOptions : string[]) : :SearchOptions
-getOptionsFromFile() : void
-extractStringFromLine(in line : string) : string
+getSearchOptions() : string
+getURL() : string
1
:HTMLParser
-webSiteNames : string[]
-searchSite : string
-dataTable : string[][]
-rowCount : int
-maxColumns : int
+HTMLParser() : :HTMLParser
+HTMLParser(in searchSite : string, in websitesNames : string[]) : :HTMLParser
#startParser(in sourceFile : File) : void
-extractXXXInformation(in sourceFile : File) : string[][]
-removeUnusedTorrents(in dataTable : string[][]) : string[][]
-extractStringFromLine(in line : string, in fromPattern : string, in toPattern : string, in lastIndex : bool) : string
-reduceString(in string : string, in fromPattern : string) : string
-removePatternInString(in string : string, in oldPattern : string, in newPattern : string) : string
-createDateString(in string : string) : string
-translateMonth(in monthNumber : string) : string
-countPatternInString(in pattern : string) : int
+getDataTable() : string[][]
+getComment(in commentFile : File) : string

Figure 20: The class model for the torrent search function.

website: The searched word, all supported websites, and the chosen search options. With these values the class will be initialize. The array for supported websites is needed for case differentiations because any provider has its own rules for searching. In the next step, the *SearchTorrents* class calls the *SearchOption* class. This class extracts all important information, for the chosen website from a XML file. This file contains information of the URL, the real search options and the sort options. If the *SearchOption* class has extracted the wanted data, then it constructs a string which can be recalled by the caller class. The strings are on the one hand the URL from the provider and on the other hand the real option string. These strings are interconnected by the caller class

SearchTorrents and in addition it adds the search word.
The search word can consist of more than one word which are separated by a space character. This character must be replaced by a special character which depends on the provider. Furthermore, the search word must translate in HTML format which means that it has to parse for other characters and replace the original HTML code like "[" to "%5B" or "]" to "%5D". For this purpose no special parser is needed, that will be performed by the single function *changePatternInString* which also changes the space character.
If all preparation are done, then the class sends the search request to the provider. For that, the *SearchTorrents* class uses the function *searchTorrentsOnWeb*. Also, this class is waiting for the response and saves it in a file. Now the *HTMLParser* class can be called.
Also the parser class (*HTMLParser*) has to know the provider name for case differentiation. Because the response text differs enormously by several providers. Therefore, extra parse functions for every provider are required. The parse function *extractXXXInformation* copies the result in a string table (two dimensional array), because all information in HTML are strings. It is useful to remove all corrupted or dead torrents in this class, because it saves a little bit of storage for all callers and the handover is slightly faster. Note that this will be noticeable by huge data volumes. The data table is now available for the caller class and can be recalled by the getter function *getDataTable*.
Also the *SearchTorrents* and the *HTMLParser* class is needed for downloading the commentaries. While in this case the comment URL exists the *SearchOption* class is not necessary. The *SearchTorrents* class requests the website with a special URL for the torrent, which was received by the search request, described before, and saved in a file. This file is handed over to the *getComment* function of the *HTMLParser*. This function removes the header information and parses the comment body. This body is saved in a string as HTML format, the reason for that will be described later.

8.3.2. Class Model for the User Interface

The class model for the user interface is separated from the search function, because the search hits are displayed in a table and a table needs some utility classes for the handling. This class model is shown in Figure 21. The class *SearchTorrentWindow* is the main class of the whole search process. At the beginning the user has to choose a torrent provider, write its search term and, if wanted by the user, select special search options. With these values the *SearchFunction* is called. Additionally all provided

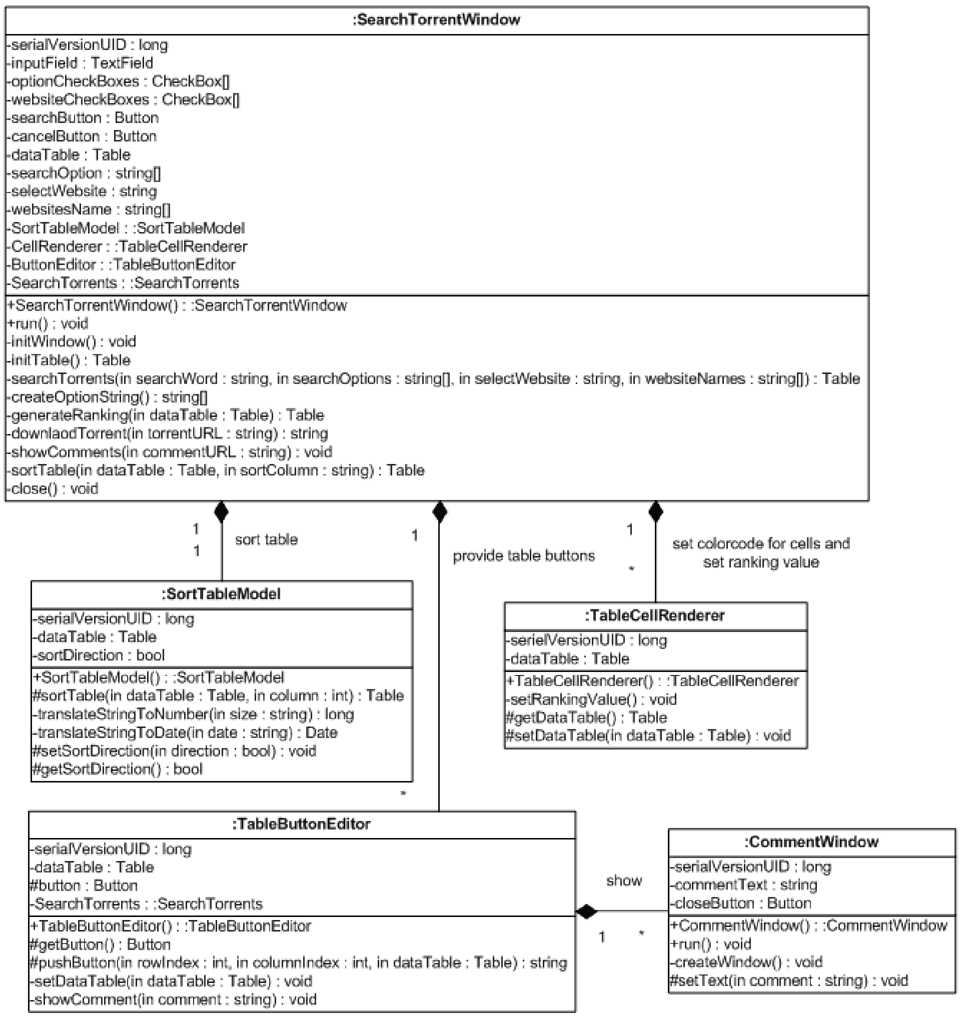

:SearchTorrentWindow
-serialVersionUID : long
-inputField : TextField
-optionCheckBoxes : CheckBox[]
-websiteCheckBoxes : CheckBox[]
-searchButton : Button
-cancelButton : Button
-dataTable : Table
-searchOption : string[]
-selectWebsite : string
-websitesName : string[]
-SortTableModel : :SortTableModel
-CellRenderer : :TableCellRenderer
-ButtonEditor : :TableButtonEditor
-SearchTorrents : :SearchTorrents
+SearchTorrentWindow() : :SearchTorrentWindow
+run() : void
-initWindow() : void
-initTable() : Table
-searchTorrents(in searchWord : string, in searchOptions : string[], in selectWebsite : string, in websiteNames : string[]) : Table
-createOptionString() : string[]
-generateRanking(in dataTable : Table) : Table
-downlaodTorrent(in torrentURL : string) : string
-showComments(in commentURL : string) : void
-sortTable(in dataTable : Table, in sortColumn : string) : Table
-close() : void
1
1
sort table
1
provide table buttons
1
*
set colorcode for cells and set ranking value
:SortTableModel
-serialVersionUID : long
-dataTable : Table
-sortDirection : bool
+SortTableModel() : :SortTableModel
#sortTable(in dataTable : Table, in column : int) : Table
-translateStringToNumber(in size : string) : long
-translateStringToDate(in date : string) : Date
#setSortDirection(in direction : bool) : void
#getSortDirection() : bool
:TableCellRenderer
-serielVersionUID : long
-dataTable : Table
+TableCellRenderer() : :TableCellRenderer
-setRankingValue() : void
#getDataTable() : Table
#setDataTable(in dataTable : Table) : void
*
:TableButtonEditor
-serialVersionUID : long
-dataTable : Table
#button : Button
-SearchTorrents : :SearchTorrents
+TableButtonEditor() : :TableButtonEditor
#getButton() : Button
#pushButton(in rowIndex : int, in columnIndex : int, in dataTable : Table) : string
-setDataTable(in dataTable : Table) : void
-showComment(in comment : string) : void
show
1
*
:CommentWindow
-serialVersionUID : long
-commentText : string
-closeButton : Button
+CommentWindow() : :CommentWindow
+run() : void
-createWindow() : void
#setText(in comment : string) : void

Figure 21: The class model for the user interface of the torrent search function.

websites are committed. The returned value is a string array in a table format, that means the array is two dimensional. This array is put in a table which must pass through different processes for a better look and feel as well as the ranking, for adding buttons and a sort option.

For a nice look and feel the *TableCellRenderer* class is responsible. There a background color is set for every second row. Also, it implements a tool-tip which presents the comment and the thanks count if they are available. Furthermore, it performs a visual hit ranking which is split into three levels. Level one, lowest level, the class sets a special foreground color for the whole line (row). On level two it adds a simple Smile icon in the tool-tip. In case of a super hit, level three, it sets a big Smile icon.

To add buttons in the table cell the *TableButtonEditor* class is used. The buttons are needed to show the comments, if this functionality is available, and to download the torrent file and then to start the download of the torrent content. If the user pushes a comment button, the *TableButtonEditor* class calls the *SearchTorrents*. If no comment is available, the source cell of the URL is "null" and then a dialog will be shown to inform the user. Otherwise, the *SearchTorrents* class answers with the commentary text which is in HTML format. Then, a new window is created in a separate thread which shows the comment text. The HTML format was retained because the comments have many icons and to show this in HTML format is useful. Also the look and feel is nicer, than in case it is converted to a sterile text format.

To sort the different columns in the table the *SortTableModel* is necessary. All values in the table are strings, therefore, the class must convert some columns to the real data type to ensure the right sorting. For the columns "Name" and "Type" no additional functions are needed. Both cells contain real string types. The "Date" column must convert all values to a real date object and furthermore in some cases it has to add the current year or generate the whole date because very new torrents have entries like "yesterday" or "day.month + time". To set a date for an entry "yesterday" is quite hard, because there exist month boundaries, year boundaries and the leap year, so a differentiation for many cases is required. A converter for string to an integer or float is used by the columns "Seeds", "Leechers" and "Size". The complexity for the seeds and leechers count is low but for the size a unit exists (GB, MB or KB), which must be considered. Therefore, before the converter can start the function, it has to parse the unit and store this value for the next step. If the number is convert to a double the value must translate into the same unit. For that the "KB" base is useful, because it can be assumed that torrents smaller than 1 KB do not exist. When the user wants to sort the table for a special column, the user has to push the header column. After the sorting, the table is transmitted to the main class, with an intermediate step to change the cell colors and to reorder for the hit ranking and then the new table structure is shown.

8.4. Implementation

In the first version of the search function I have implemented for two different providers which are the most popular torrent providers in the Internet (Mininova[15] and The Pirate

[15] http://www.mininova.org/

Bay[16]). At first I have analyzed the different search option with their terminologies. These values are saved in a *searchOptions.properties* file in the XML format. The file is hidden from the user because it is a component of the *BitThief.jar* file. The file consists of the search option including the separator to connect some options, the URL, and the sort options to sort the reply by the provider. This file is shown in Listing 7 on page 85 in the Appendix A. I chose this kind for more flexibility because if the provider changed its key words and further to easily add another provider.
For the implementation of the parse functions in the *HTMLParser* class I created for every considered provider a separate function. For this purpose I have analyzed the HTML code of the reply from the websites. Separate parse functions are important because the order of data and the offered data are different. For example Mininova provides comments and thanksgivings which The Pirate Bay website does not do. Also, the kind of source code is different and on the other hand the key words and separators for the entries and also the code arrangement. This means The Pirate Bay provides every data row in a line which is separated with the line end character. In contrast, Mininova presents the whole data as only one line. Therefore, the read in of the data is different.
In Figure 22 the sequence diagram for the search action is shown. It illustrates the steps from creating a search request to showing the reply.

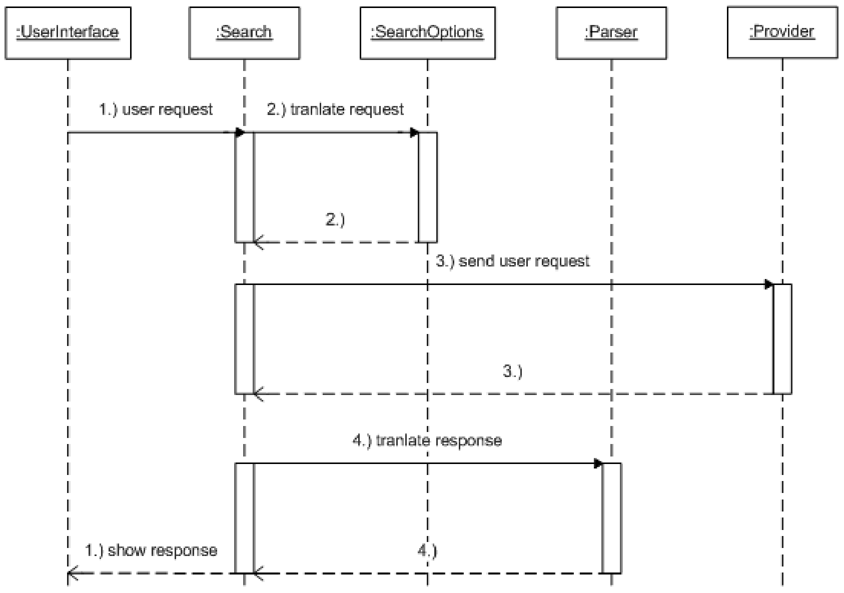


Figure 22: Sequence diagram for the search function. This shows the sequence between request and reply.

[16]http://thepiratebay.org/.

8.4.1. Implementation of the Torrent Ranking

The whole ranking service of torrents is split into five levels. At first, the search request is sorted by the seeder count on the provider side. This guarantees that the client responds with the most frequently distributed torrents. In the second step all corrupted and dead torrent will be removed. In the last three steps the *BitThief* client performs its own ranking. For that the client evaluates the thanks count, because this technique is robust, easy, and fast. The visualization was described before in detail, the client colored the foreground of the row and adds to kinds of icons (if it is a hit or super hit).
The other technique to evaluate the comment is rejected because the complexity with the languages are too big and the counting of smile icons is insecure, because some users adds more than one icon to underline theirs emotions. And further the additional request has the disadvantage that it could make the function useless for the user, because if the user has a slow Internet connection, the time to create the data table including the ranking is too large and therefore useless.

8.4.2. Implementation of the GUI

The search function can be called from the main window. There exists a "Search" menu in the menu bar where the function is callable. Also a short cut exists to start the search dialog (*CTRL S*). In Figure 23 the start display is shown. There the user has

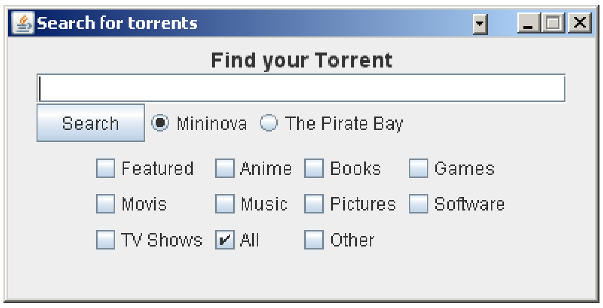
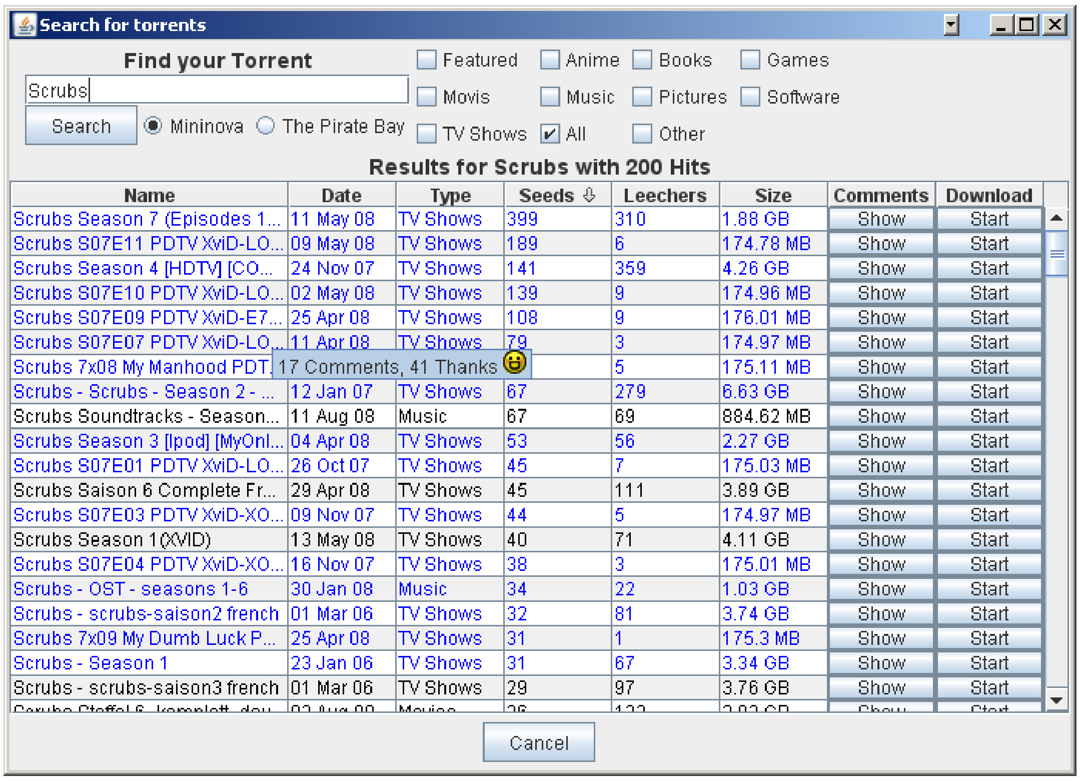


Figure 23: BitThief 0.2.0: Start screen of the search function.

to input the search word or sentence, select the wanted provider and to chose a special sort option. Then the user can validate the entry with the "Enter" key or with the "Search" button. Also an abort option is available either to perform with the "ESC" key or with the cross on the top of the window. The default value for the sort option is always "All" and the default provider is Mininova.

If the user has committed its request, the next window is shown (Figure 24). Now the


Search for torrents

Find your Torrent

Scrubs

Search [✓ Mininova] The Pirate Bay

Featured, Anime, Books, Games, Movis, Music, Pictures, Software, TV Shows, [✓ All], Other

Results for Scrubs with 200 Hits

Name	Date	Type	Seeds ⇩	Leechers	Size	Comments	Download
Scrubs Season 7 (Episodes 1...	11 May 08	TV Shows	399	310	1.88 GB	Show	Start
Scrubs S07E11 PDTV XviD-LO...	09 May 08	TV Shows	189	6	174.78 MB	Show	Start
Scrubs Season 4 [HDTV] [CO...	24 Nov 07	TV Shows	141	359	4.26 GB	Show	Start
Scrubs S07E10 PDTV XviD-LO...	02 May 08	TV Shows	139	9	174.96 MB	Show	Start
Scrubs S07E09 PDTV XviD-E7...	25 Apr 08	TV Shows	108	9	176.01 MB	Show	Start
Scrubs S07E07 PDTV XviD-LO...	11 Apr 08	TV Shows	79	3	174.97 MB	Show	Start
Scrubs 7x08 My Manhood PDT	17 Comments, 41 Thanks 😃			5	175.11 MB	Show	Start
Scrubs - Scrubs - Season 2 - ...	12 Jan 07	TV Shows	67	279	6.63 GB	Show	Start
Scrubs Soundtracks - Season...	11 Aug 08	Music	67	69	884.62 MB	Show	Start
Scrubs Season 3 [Ipod] [MyOnl...	04 Apr 08	TV Shows	53	56	2.27 GB	Show	Start
Scrubs S07E01 PDTV XviD-LO...	26 Oct 07	TV Shows	45	7	175.03 MB	Show	Start
Scrubs Saison 6 Complete Fr...	29 Apr 08	TV Shows	45	111	3.89 GB	Show	Start
Scrubs S07E03 PDTV XviD-XO...	09 Nov 07	TV Shows	44	5	174.97 MB	Show	Start
Scrubs Season 1 (XVID)	13 May 08	TV Shows	40	71	4.11 GB	Show	Start
Scrubs S07E04 PDTV XviD-XO...	16 Nov 07	TV Shows	38	3	175.01 MB	Show	Start
Scrubs - OST - seasons 1-6	30 Jan 08	Music	34	22	1.03 GB	Show	Start
Scrubs - scrubs-saison2 french	01 Mar 06	TV Shows	32	81	3.74 GB	Show	Start
Scrubs 7x09 My Dumb Luck P...	25 Apr 08	TV Shows	31	1	175.3 MB	Show	Start
Scrubs - Season 1	23 Jan 06	TV Shows	31	67	3.34 GB	Show	Start
Scrubs - scrubs-saison3 french	01 Mar 06	TV Shows	29	97	3.76 GB	Show	Start
Scrubs Staffel 6 komplett deu	03 Aug 08	Movies	26	[illegible]	[illegible]	Show	Start

Cancel

Figure 24: BitThief 0.2.0: Search window inclusive suggestions.

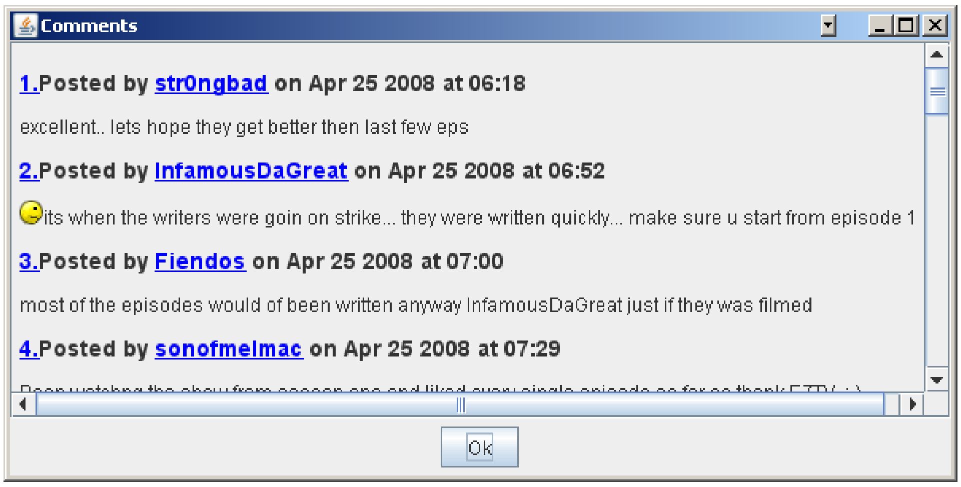
user can search again for a new term or the user can set limits by choosing the options or the user searches at another provider for the same term.

If it is possible to sort the table by other criteria by pushing the table header on the wanted column. For the first six columns the meaning is clear. By pushing the "Comments" header all rows are sorted by the counts of comments. If the "Download" header is pushed then all rows are sorted by the "Thanks" count. The sorting is possible for upside and downside. The direction is switched at the next click.

If the user moves the mouse over the rows, slowly and not too fast, then ranking is displayed in a tool-tip. It is also shown in Figure 24 with a super hit.

The window size is resizable, but the minimum value is given by the table and the maximum value is given by the screen dimension and/or the resolution of the display. If the window is broadened, a new table size will be calculated (column width). Nearly all columns have a small range where they vary, but the "Name" column get the rest of the width because the name needs most of the space in the table and all other columns have nearly a fixed size.

To show the comments from a torrent, the user has to push the "Show" button in the "Comments" column. If it is performed and a comment is available, then the following window is shown (Figure 25). Otherwise, a failure message is shown. The comments are



1.Posted by str0ngbad on Apr 25 2008 at 06:18

excellent.. lets hope they get better then last few eps

2.Posted by InfamousDaGreat on Apr 25 2008 at 06:52

its when the writers were goin on strike... they were written quickly... make sure u start from episode 1

3.Posted by Fiendos on Apr 25 2008 at 07:00

most of the episodes would of been written anyway InfamousDaGreat just if they was filmed

4.Posted by sonofmelmac on Apr 25 2008 at 07:29

Figure 25: BitThief 0.2.0: Comment window including user comments.

all presented in the original HTML format, to ensure a nice look and feel including the icons. This window is started in a separate thread, so the user can show different comments and it is useful to compare the content. The sequence diagram for the comments is shown in Figure 26, it illustrates the action to get and visualize the user comments.

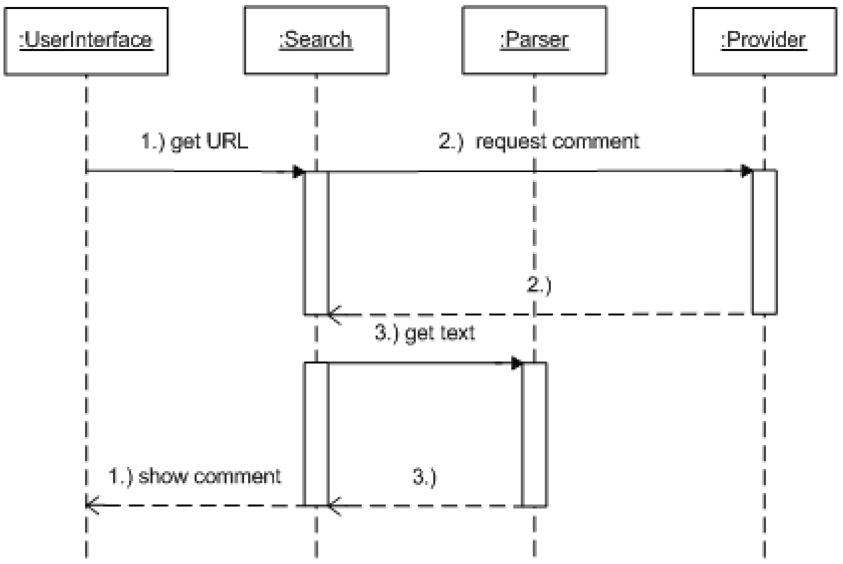


Figure 26: Sequence diagram for the search function. This illustrates the actions to show the user comments.

To start a torrent download, the "Start" button in the Download column is needed. If the user pushes the button, the select torrent will be downloaded in a special directory. The download folder, also called "Download", is in the "bitthief" folder in the home directory of the user. After that the start procedure for downloading the content is called and the torrent file is handed over. Now the user has to select a destination folder and can start the download process. The action which will be performed at the start of the download process is shown in the sequence diagram Figure 27.

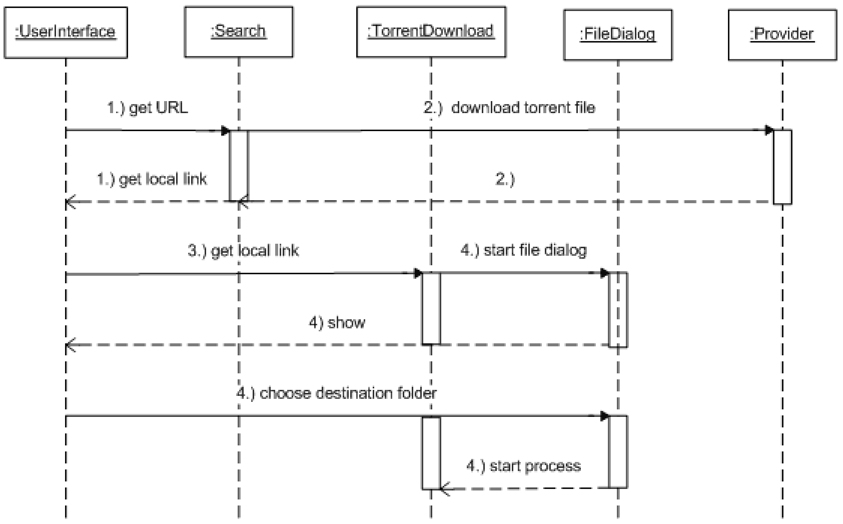


Figure 27: Sequence diagram for the search function. Download a selected torrent file and start the download process.

8.5. Testing

For testing the whole search function, I split the test into different stages. In the first stage, I tested only the search and parse functionality with different terms. The received results were compared with the content of the website. The result of the test was that I must improve the parse function, because some information has included HTML characters which were not converted. Another problem was, especially for Mininova, some information were corrupted and/or in a wrong column after parsing. The reason is the Mininova website sends its information in different orders and the challenge is to recognize the right one. This is not indicated by a column specified HTML tag, on the contrary most parts have the same start and end tag. Therefore it must be recognize by the content. To find these failures in the source code it was a little bit tricky because with each failure the source code had to be analyzed again. The last failure that I have recognized was that, if the user searched for a special torrent type, for example "Movies" or "TV Shows", the received answer does not consist of this type, so it must

added manually and/or a function has to recognize that this field is null and then it has to add the type. This problem was only by Mininova.
In the second test phase I checked the sort functionality. For that I have sorted every column in both direction. This test has showed that Java sorts the view port of the table and not the real data container. A further problem was discovered. All references are corrupted which use the other table classes, in particular the reference for the torrent URL, the comments, and consequently the hit ranking. The reason for the failure is Java converts the table structure in a vector automatically, and the problem is that no back converter exists. Thus, a separate converter is necessary. This function returned the new sorted table for all other classes.
Next I tested the download function for torrents and comments. This test generates again a new problem with defective URLs which means that the client received nothing. The failure was the URLs were received in clear text, which contains space characters and brackets, since this have to be converted into HTML code for a request. After converting this character before sending the request, the functions did work in all cases. These types of failures, which were described before, are not failures in the concept and/or this kind of problems was not to be foreseen.
In the last step of the testing I checked correctness of the hit ranking. This was not difficult, because I have the right visualization after my rules. The rules are at first if there is one "Thanks", set a foreground color, at second if there are more than two "Thanks", add a small smile. And in the end, add for a super hit, with more than five "Thanks", a big smile. This test was free of failure.
After this test phase I can say the search function is reliable and works well.

8.6. Implementation of more Provider

The development and the implementation showed that would be it easy to implement new provider, if it were necessary. For that only minimal changes are needed. The programmer has to expand the *searchOptions.properties* file and add the new provider name in the source code. Further, the programmer has to develop a new parse function for the *HTMLParser* class. These three steps are only required to improve the search function.

9. Subscription Service

In this chapter I describe a subscription service for TV shows, which are distributed via torrents. This function should increase the usability and should implement a new unique function for the *BitThief* client to set itself apart from other clients.

9.1. Motivation

Many users of BitTorrent clients use this technique to distribute TV shows and also many users like that to organize their own TV program. A lot of popular TV shows are available, at all torrent providers, shortly after the live stream is done the users like to watch these promptly. Also the users which are far away from home do not want to do without their home TV. So it is useful and interesting to develop a subscription service. Further, many users are sluggardly to search for a title manuell again and again, if it is current not provided, so an automatic search function is required also what can be understand as a subscription. Another reason for the subscription service is to get more benefit of the search function which was described in Chapter 8.

9.2. Concept

At first one has to think about what a subscription service needs and what is required for a subscription. The basics for a subscription service is a search function to search for subscribed torrents. This function is similar to that function which is already described in detail in chapter 8.
Further a subscription service needs tools for management where the user can list all subscriptions and where the user can add new, edit or cancel a subscription manually.
What is required to subscribe to a torrent? To subscribe to a torrent it needs some important information for searching. This information are the torrent name, the provider and the type (movie, book, game etc.) for better results. Further, to subscribe to a single TV show the torrent needs an indicator for the current season and episode count. This indicator has to be extracted from the name to increase it for the next episode. That guarantee that the user get the next TV show and it realized a real subscription service. Thereby the challenge is that different patterns exist to describe such torrents and further a complete season consist of a part of this pattern (series count). The most common pattern are *"SxxExx"*, *"Sxx EPxx"* or *"yyXyy"* (x and/or y represent real numbers) also

this pattern can be written in small letters and can begin with a zero. Also it is required to remove all unimportant words from the torrent name, because it increases the hit rate by searching for new torrents. Further, some titles consist of the series name plus the episode name. That part, the episode name, is unimportant and/or it could destroy the subscription functionality especially the search functionality.
If the function has detected new torrents, the results should be shown in the same GUI concept as the search function. Because too many different interfaces could confuse users and it is useful to use well known components, also it guarantee a intuitive handling. For the management dialog another interface concept is needed, for that the concept of the multi-torrent dialog could be useful. Only the dialogs for editing and creation a new subscription need a new design, but both dialog interfaces should be equal. In conclusion, the subscription service should be easy and intuitive to use, hence I will use well known components for the user interface.

9.3. Class and Sequence Model

The class model for the subscription service is similar to the search function. But it needs extra classes for its own process (management of subscriptions, search and show subscribed torrents). To illustrate the whole concept of this service I have create a conceptional class diagram which is shown in Figure 28. The search function as well as the sorting of the results and the hit ranking will not be described in this chapter. These functionalities are equal to the process of the search function and are described in detail in the Section 8.3.1 (page 56 et seqq.) and 8.3.2 (page 58 et seqq.). I will only explain their differences. In Figure 29 it is shown the class model of the subscription service. Only the classes for the subscription service (*SubscriptionService*, *SearchSubscriptionsWindow*), *SubscriptionServiceWindow*, *NewSubscriptionWindow*, *EditSubscriptionWindow* and *CancelSubscriptonWindow* are shown, all other classes are denoted as interfaces.

The core of the subscription service is on the one hand the search function and on the other hand the *SubscriptionService* class. This class saves all subscribed torrents in the file *Subscription.properties*, which is placed in the *bitthief* folder in the home directory. The structure of this file is XML. In that file the name, the season indicator (if exist else "null"), the torrent type and the provider name for each subscription is saved.
The class *SubscriptionService* has an interface to the search window. The search window calls the *SubscriptionService* and commits the torrent name. Then the function

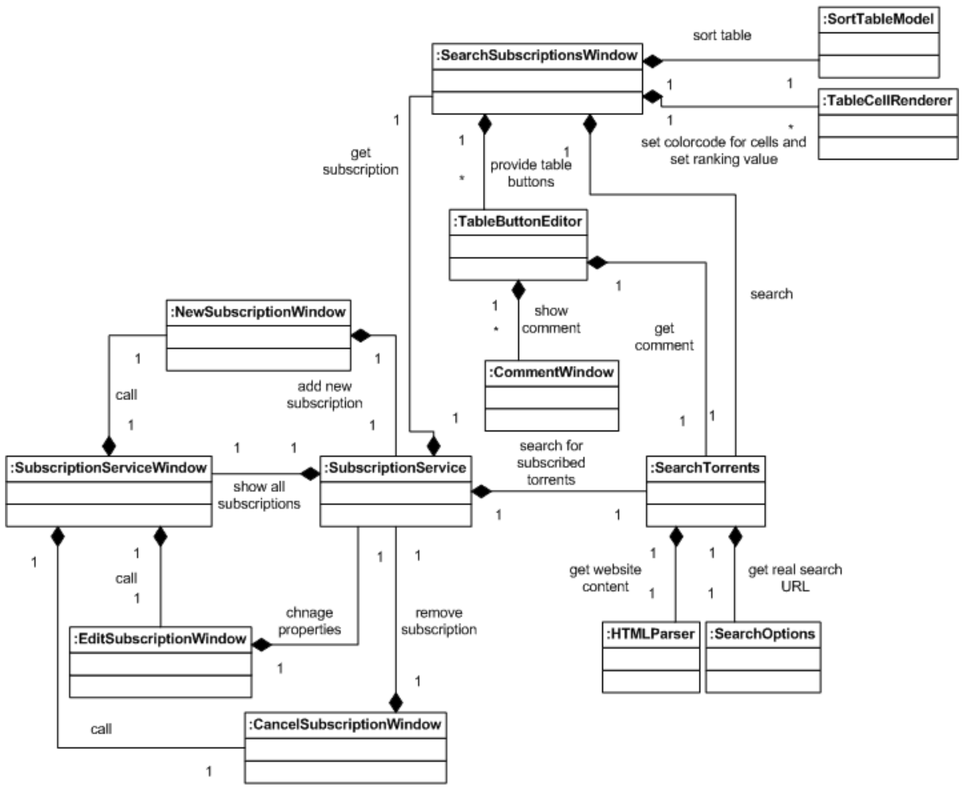


Figure 28: Conceptual class model for the subscription service.

isTorrentSubscriptionable checks this value if a pattern exists to subscribe this torrent automatically, which is a feature for TV shows. If the torrent can be to subscribed to the function returns with true else with false. If it is detect a subscribeable TV show the user gets a message which asks him if he wants to subscribe to this torrent otherwise no message will be shown. The sequence diagram in Figure 30 shows the activity to recognize a torrent which can be subscribed to automatically.
If the user wants to subscribe to a torrent, the *saveSubscription* function is calls. This function calls the private function *saveSubscriptionInFile* and saves the important information, if no failure appears, the functions returns true. To avoid multiple entries by the same torrent description a test routine is needed. This is performed by the private function *existSubscription.* if this function returns true, it means that the subscription already exists, the save function knows that no write operation will be performed, otherwise it writes the subscription in the file. If the subscription exists and the function returns true, it is no indication of a failure. That is hidden from the user because it should guarantee a clean subscription file and also to avoid too many unneeded search request.
If the user looks up new subscribed torrents the *SearchSubscriptionsWindow* is called

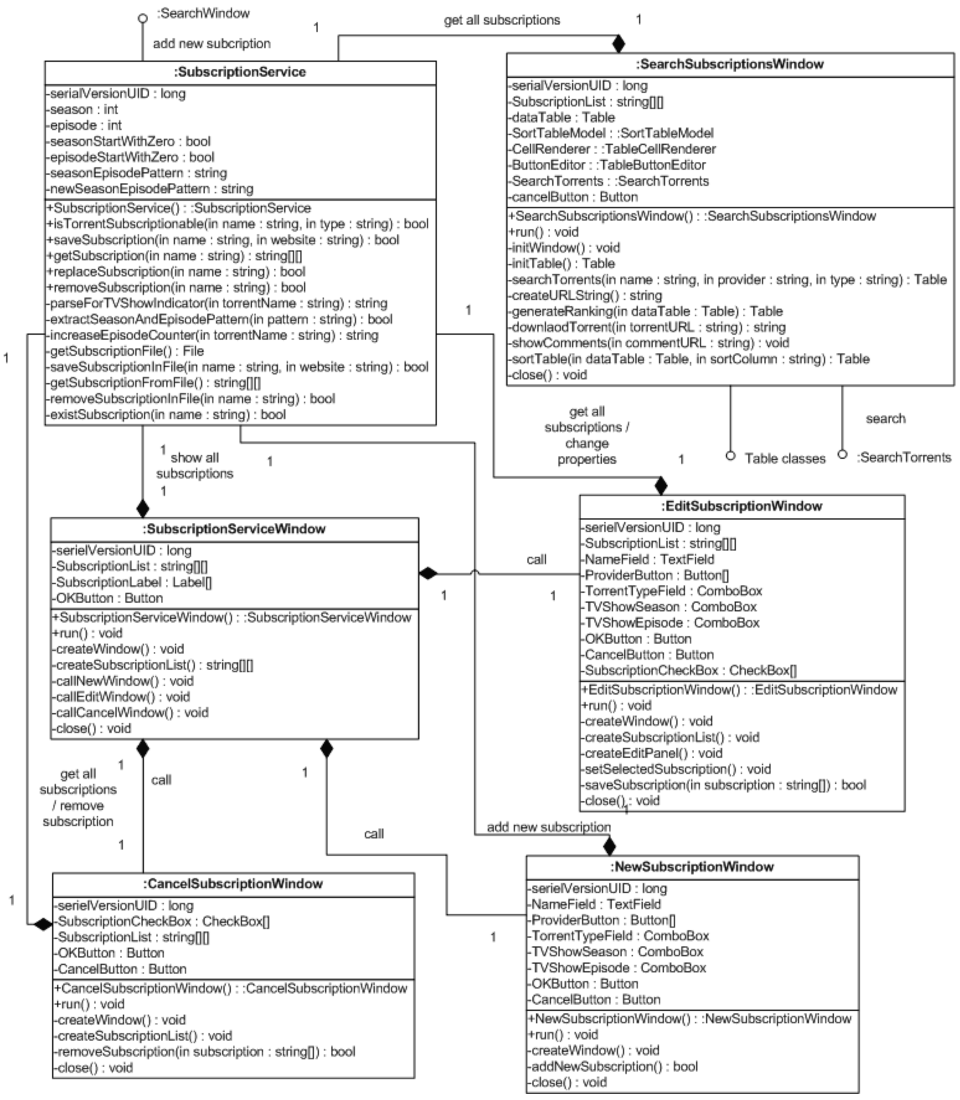

:SearchWindow
add new subcription
get all subscriptions
:SubscriptionService
-serialVersionUID : long
-season : int
-episode : int
-seasonStartWithZero : bool
-episodeStartWithZero : bool
-seasonEpisodePattern : string
-newSeasonEpisodePattern : string
+SubscriptionService() : :SubscriptionService
+isTorrentSubscriptionable(in name : string, in type : string) : bool
+saveSubscription(in name : string, in website : string) : bool
+getSubscription(in name : string) : string[][]
+replaceSubscription(in name : string) : bool
+removeSubscription(in name : string) : bool
-parseForTVShowIndicator(in torrentName : string) : string
-extractSeasonAndEpisodePattern(in pattern : string) : bool
-increaseEpisodeCounter(in torrentName : string) : string
-getSubscriptionFile() : File
-saveSubscriptionInFile(in name : string, in website : string) : bool
-getSubscriptionFromFile() : string[][]
-removeSubscriptionInFile(in name : string) : bool
-existSubscription(in name : string) : bool
:SearchSubscriptionsWindow
-serialVersionUID : long
-SubscriptionList : string[][]
-dataTable : Table
-SortTableModel : :SortTableModel
-CellRenderer : :TableCellRenderer
-ButtonEditor : :TableButtonEditor
-SearchTorrents : :SearchTorrents
-cancelButton : Button
+SearchSubscriptionsWindow() : :SearchSubscriptionsWindow
+run() : void
-initWindow() : void
-initTable() : Table
-searchTorrents(in name : string, in provider : string, in type : string) : Table
-createURLString() : string
-generateRanking(in dataTable : Table) : Table
-downlaodTorrent(in torrentURL : string) : string
-showComments(in commentURL : string) : void
-sortTable(in dataTable : Table, in sortColumn : string) : Table
-close() : void
get all subscriptions / change properties
search
Table classes
:SearchTorrents
show all subscriptions
:SubscriptionServiceWindow
-serielVersionUID : long
-SubscriptionList : string[][]
-SubscriptionLabel : Label[]
-OKButton : Button
+SubscriptionServiceWindow() : :SubscriptionServiceWindow
+run() : void
-createWindow() : void
-createSubscriptionList() : string[][]
-callNewWindow() : void
-callEditWindow() : void
-callCancelWindow() : void
-close() : void
call
:EditSubscriptionWindow
-serielVersionUID : long
-SubscriptionList : string[][]
-NameField : TextField
-ProviderButton : Button[]
-TorrentTypeField : ComboBox
-TVShowSeason : ComboBox
-TVShowEpisode : ComboBox
-OKButton : Button
-CancelButton : Button
-SubscriptionCheckBox : CheckBox[]
+EditSubscriptionWindow() : :EditSubscriptionWindow
+run() : void
-createWindow() : void
-createSubscriptionList() : void
-createEditPanel() : void
-setSelectedSubscription() : void
-saveSubscription(in subscription : string[]) : bool
-close() : void
get all subscriptions / remove subscription
call
call
add new subscription
:CancelSubscriptionWindow
-serielVersionUID : long
-SubscriptionCheckBox : CheckBox[]
-SubscriptionList : string[][]
-OKButton : Button
-CancelButton : Button
+CancelSubscriptionWindow() : :CancelSubscriptionWindow
+run() : void
-createWindow() : void
-createSubscriptionList() : void
-removeSubscription(in subscription : string[]) : bool
-close() : void
:NewSubscriptionWindow
-serielVersionUID : long
-NameField : TextField
-ProviderButton : Button[]
-TorrentTypeField : ComboBox
-TVShowSeason : ComboBox
-TVShowEpisode : ComboBox
-OKButton : Button
-CancelButton : Button
+NewSubscriptionWindow() : :NewSubscriptionWindow
+run() : void
-createWindow() : void
-addNewSubscription() : bool
-close() : void

Figure 29: Class model for the subscription service, only the subscription classes are shown.

and a window will be shown, but before the window is shown, the class is called the *getSubscription* function from the *SubscriptionService* class. The whole process of searching for subscribed torrents to downloading a wanted torrent is shown in Figure 31. The *getSubscription* function calls the private function *getSubscriptionFromFile* which returns a string array (two dimensional) that includes all information of the subscriptions. With that string array, if it is not null, the *SearchTorrents* class is called with each entry one after the other, if more subscriptions exist. All found results are collected in a table and will be presented to the user at the end. All results are presented according to the same rules from the *SearchTorrentWindow* which was described in Chapter 8 (hit ranking, sorting etc.). If the result table is null, a message will inform the user. When

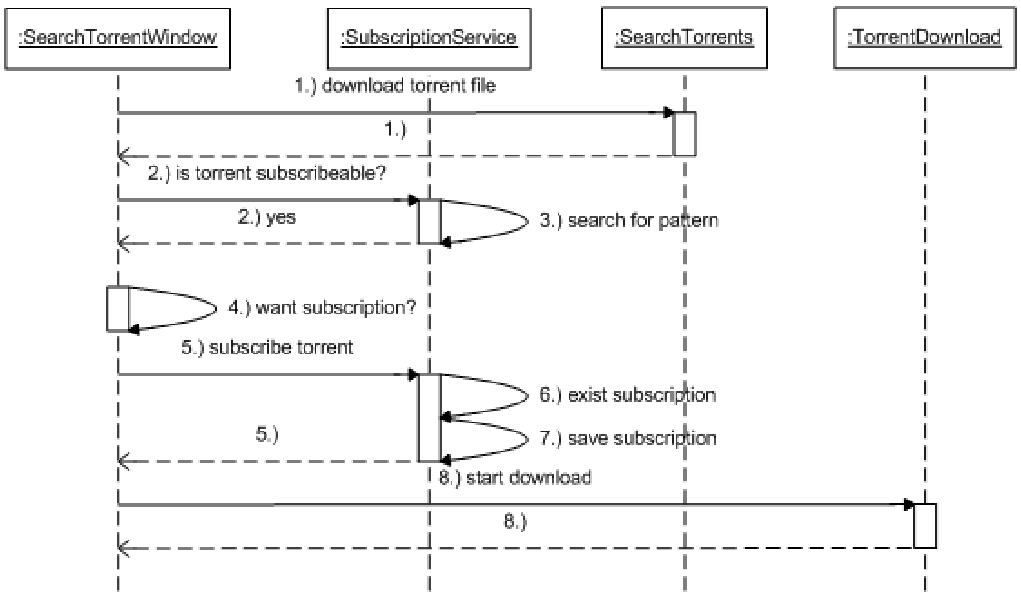


Figure 30: Sequence diagram shows the recognition of a subscribeable TV show.

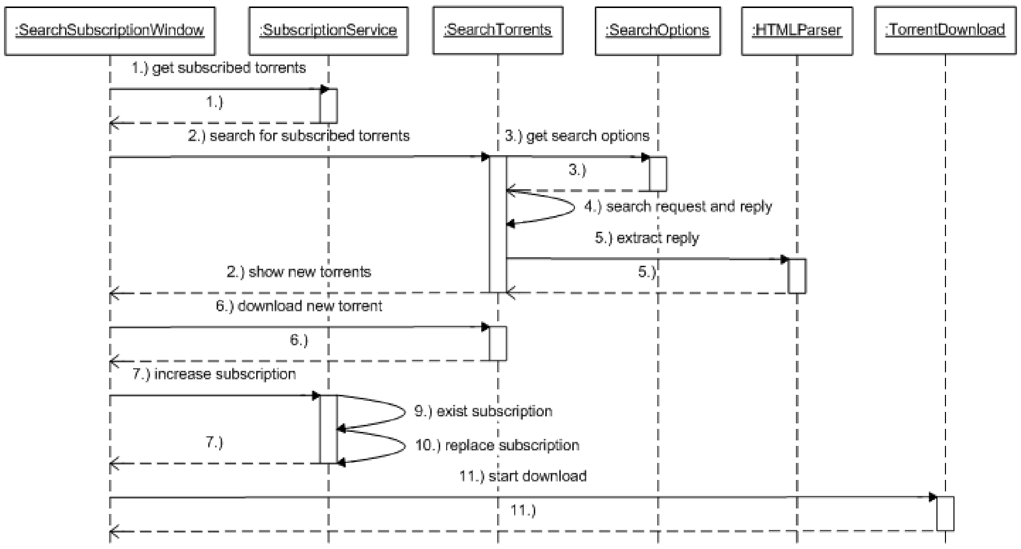


Figure 31: Sequence diagram shows the activity for searching subscribed torrent and downloading one.

new torrents are detected and the user starts to download a TV show, which is subscribe automatic, the subscription file must modified. In this step the subscription indicator will be increased and replaces the entry in the subscription file. This will be performed by the *replaceSubscription* function. During this step the function looked up if the new entry exists, because it is possible that a user subscribes to older torrents, for example the same season but different episodes which can lead to double entries. If a further entry exists with the same description, the current entry will be removed from the file.

To subscribe to torrents manually the *NewSubscriptionWindow* is needed. Its provides a dialog to describe the search parameters like search name, torrent type (Movie, TV show, Book etc.) and the wanted torrent provider. Has the user set all parameters the entry will be saves in the *Subscription.properties* file via the *SubscriptionService* class and than it returns to *SubscriptionServiceWindow* what present all exist subscriptions. The sequence diagram for this activity is shown in Figure 32.

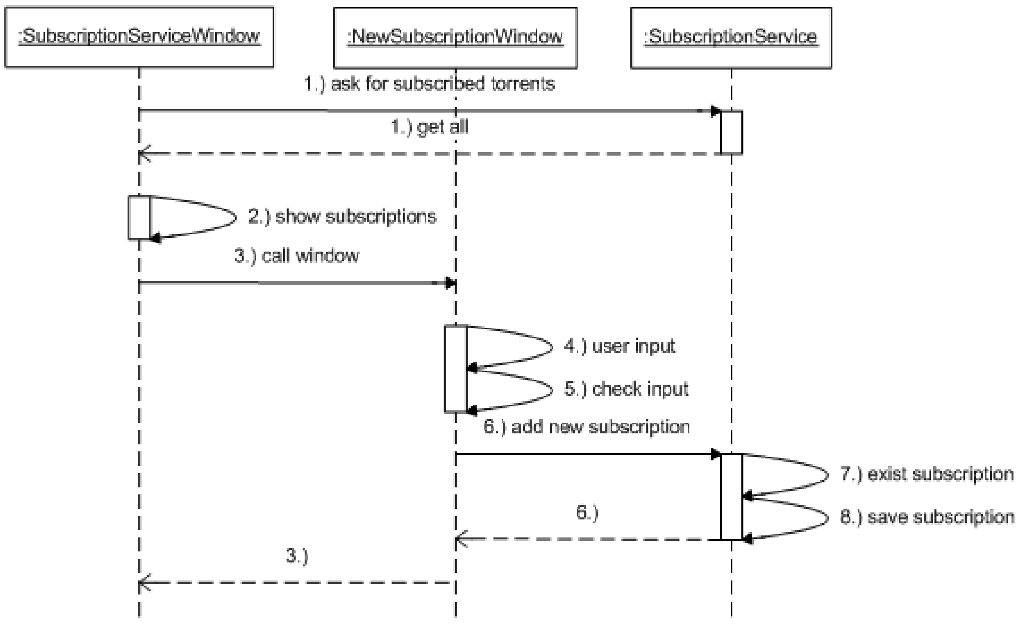


Figure 32: Sequence diagram to add a new subscription manually.

The edit window is needed to change properties of a torrent e.g. did the user a mistake or to change the TV show counter manually. This class calls the *SubscriptionService* class to gets all exist subscriptions and than the user has to select the wandet entry for changing. Note the call of the *SubscriptionService* class to gets all subscriptions instead of hand-over the subscriptions from the *SubscriptionServiceWindow* is needed to guarantee that this window shows all available subscriptions, because all windows runs in a separate thread and it is possible that users add or removed subscriptions in another window (threat). Has the user submits the change entry the window returns to the *SubscriptionServiceWindow*. The sequence diagram for this function is shown in Figure 33.

The *CancelSubscriptionWindow* is used to remove and/or cancel subscribed torrents (one or more). To get all subscriptions are obtained in the same manner as described by the edit class. The activity to remove a subscription is shown in the sequence diagram in Figure 34. In that case, the user has to select the wanted torrent and then

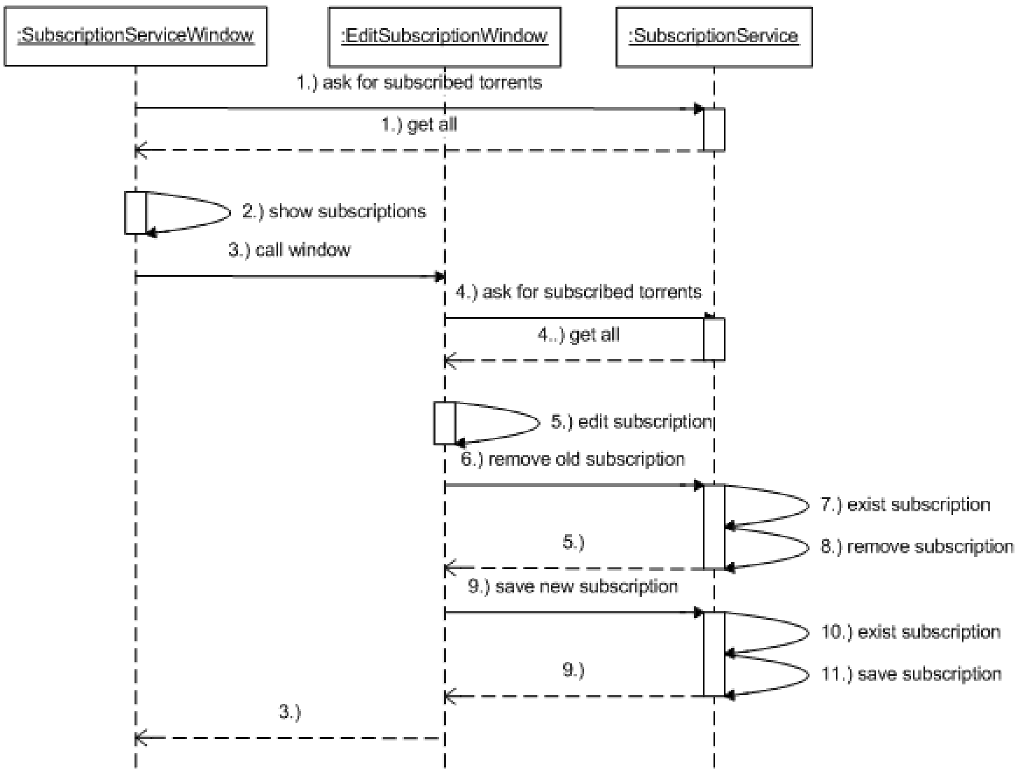


Figure 33: Sequence diagram to change properties of a subscription.

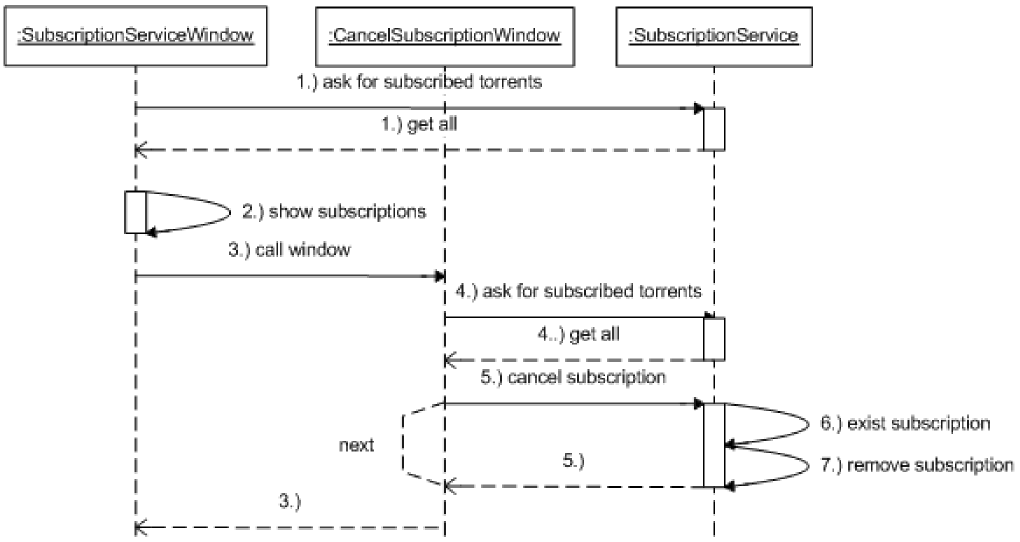


Figure 34: Sequence diagram for canceling a subscription.

to commit it. With the description of the torrent the *removeTorrent* function from the *SubscriptionService* class is called. The remove function calls at first the test routine *existSubscription* and then if it returns true the internal (private) function *removeSubscriptionFromFile* which performs the deletion.

9.4. Implementation

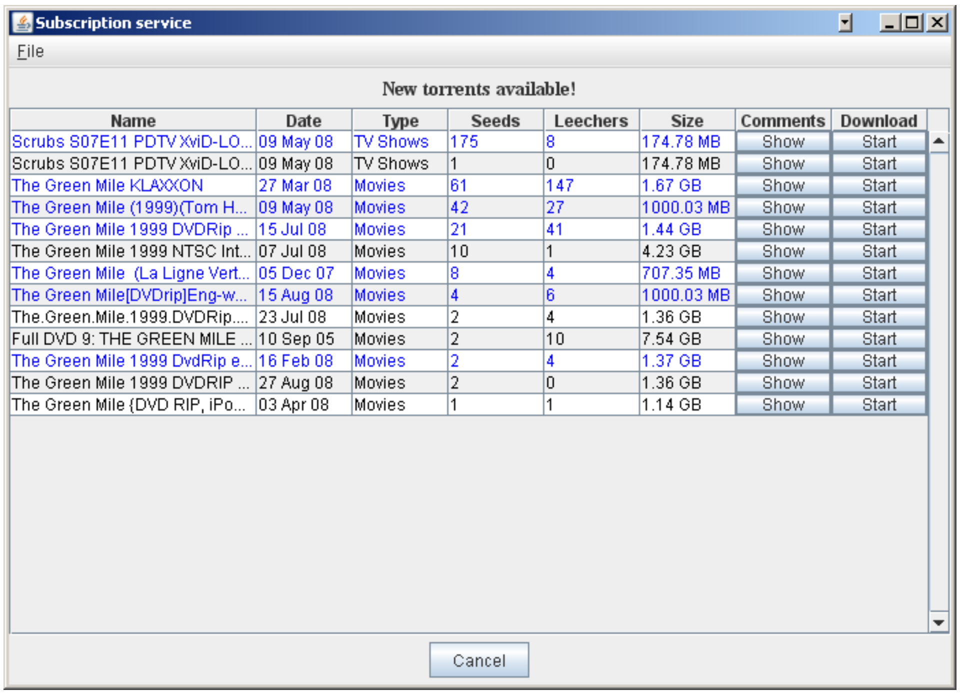
The file where all subscriptions are saved is shown in Listing 6, the listing shows two subscribed torrents. Additional to the name, series pattern, type, and the provider name the file has a counter. This counter represents the number of current subscriptions and is used inside the classes for initialization and search requests.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
 This is the subscription file for BitThief (0.2.0)
 created on Sun Aug 31 18:32:38 CEST 2008
-->
<begin>
<subscriptions>2</subscriptions>
<name>Scrubs S07E11</name>
<serie>S07E11</serie>
<type>TV Shows</type>
<url>Mininova</url>
<name>The Green Mile</name>
<serie>null</serie>
<type>Movies</type>
<url>Mininova</url>
<end>
```

Listing 6: Structure of the *Subscription.properties* file including two subscribed torrents.

The subscription service is implemented in different places. The search functionality is placed in the *Search* menu in the main window. An example for this window is shown in Figure 35, in this figure it are see new subscribed torrents that are available. If the user calls this window, it starts the searching for subscribed torrents. Also, the option exists to search automatically for new torrents. This option must be set in the "Option → Options" menu, this functionality is disabled by default. If the automatic searching is enabled, then the *BitThief* client looks up new torrents at startup. To guarantee a clean launch (for initializing running torrents) the searching sleeps at first and starts after the initializing phase.

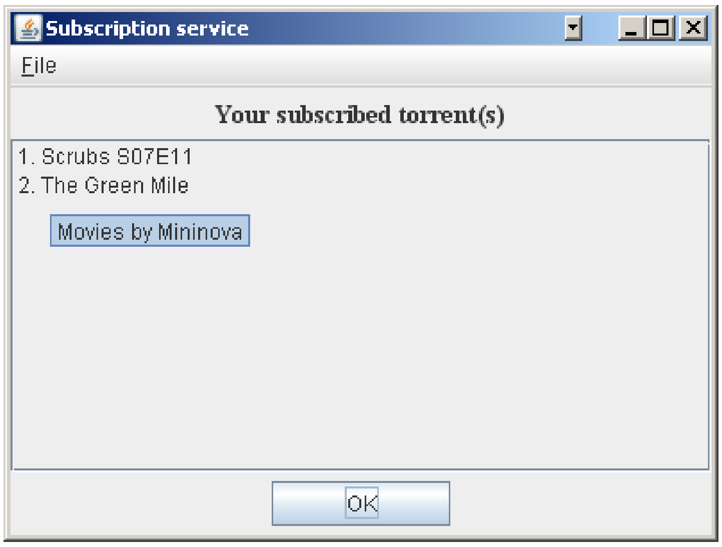
The window for subscription management, where are all subscriptions list and the user has the option to add, edit or cancel a subscription is placed in the "Option →



Name	Date	Type	Seeds	Leechers	Size	Comments	Download
Scrubs S07E11 PDTV XviD-LO...	09 May 08	TV Shows	175	8	174.78 MB	Show	Start
Scrubs S07E11 PDTV XviD-LO...	09 May 08	TV Shows	1	0	174.78 MB	Show	Start
The Green Mile KLAXXON	27 Mar 08	Movies	61	147	1.67 GB	Show	Start
The Green Mile (1999)(Tom H...	09 May 08	Movies	42	27	1000.03 MB	Show	Start
The Green Mile 1999 DVDRip ...	15 Jul 08	Movies	21	41	1.44 GB	Show	Start
The Green Mile 1999 NTSC Int...	07 Jul 08	Movies	10	1	4.23 GB	Show	Start
The Green Mile (La Ligne Vert...	05 Dec 07	Movies	8	4	707.35 MB	Show	Start
The Green Mile[DVDrip]Eng-w...	15 Aug 08	Movies	4	6	1000.03 MB	Show	Start
The.Green.Mile.1999.DVDRip....	23 Jul 08	Movies	2	4	1.36 GB	Show	Start
Full DVD 9: THE GREEN MILE ...	10 Sep 05	Movies	2	10	7.54 GB	Show	Start
The Green Mile 1999 DvdRip e...	16 Feb 08	Movies	2	4	1.37 GB	Show	Start
The Green Mile 1999 DVDRIP ...	27 Aug 08	Movies	2	0	1.36 GB	Show	Start
The Green Mile {DVD RIP, iPo...	03 Apr 08	Movies	1	1	1.14 GB	Show	Start

Figure 35: BitThief 0.2.0: Subscription search window shows the results of the subscribed torrents.

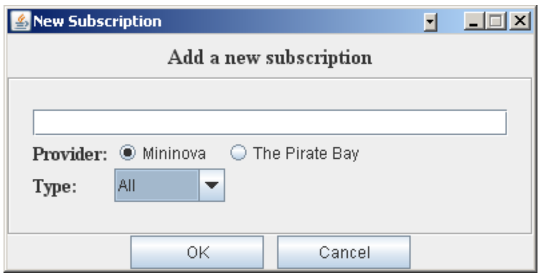
Manage subscriptions" menu from the main window. This window is shown in Figure 36. This window has a menu-bar *File* to calls the *New*, *Edit* or *Cancel* function. To give a quick overview for each entry a tool-tip is implemented which present the type and the provider for each subscription.



1. Scrubs S07E11
2. The Green Mile

Figure 36: BitThief 0.2.0: Subscription service window shows all subscribed torrents.

To add a new subscription manually the window is shown from Figure 37. In that dialog the user has to enters the name, select the provider and to chose the torrent type. Further the user can adds new TV shows manually, then he has to defines the season and the episode. If no episode (begin at) selects but the season counter is selects, then the user want to subscribe to the whole series. In that case, that the user subscribes to no TV shows the TV show indicator in the *Subscription.properties* file is null, which marks this field as invalid.

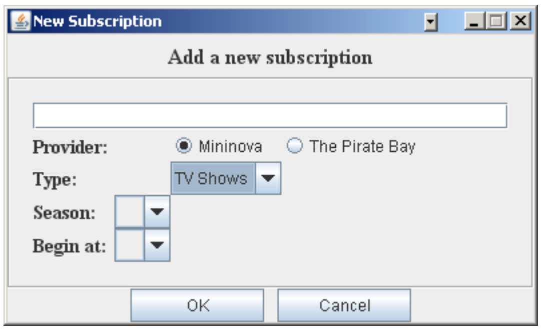

New Subscription

Add a new subscription

Provider: [✓ Mininova] The Pirate Bay

Type: All

(a) Normal display for all torrent types.


New Subscription

Add a new subscription

Provider: [✓ Mininova] The Pirate Bay

Type: TV Shows

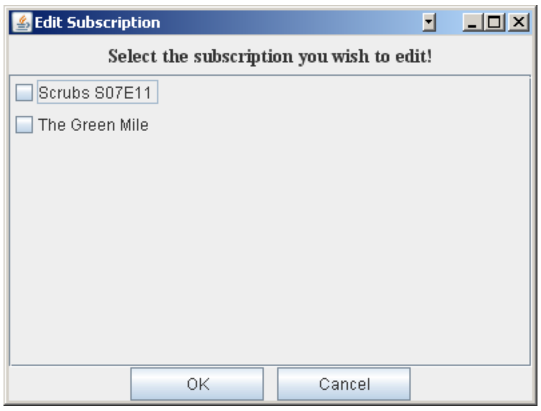
Season:

Begin at:

(b) Display for TV shows.

Figure 37: BitThief 0.2.0: Window to add a subscription manually.

The window to change the properties of a subscription is split into two views. At first the user sees a selection dialog where he has to selects the wanted torrent and than in the next step the edit dialog is shown. This window is equal to the new subscription window. Both views are shown in Figure 38. The edit window is a little bit more

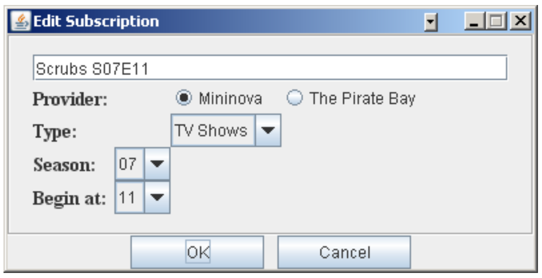

Edit Subscription

Select the subscription you wish to edit!

☐ Scrubs S07E11

☐ The Green Mile

(a) Window to select a subscription.


Edit Subscription

Scrubs S07E11

Provider: [✓ Mininova] The Pirate Bay

Type: TV Shows

Season: 07

Begin at: 11

(b) Edit dialog for a subscription.

Figure 38: BitThief 0.2.0: Window to edit a subscription.

complex as the *New* window, because it has to sets all current values and further in a special case where the subscription is a TV show it has to recognizes the series-episode pattern. If the old subscription was a TV show and the new one is it not, then it has to remove the season-episode indicator from the name. In the other case, it was for example a movie and now the user want the TV show of it, then it must adds the season-episode indicator to the name. The edit window needs a special replace function. This function does not really replace the subscription, but it removes at first the old entry and then it saves the new one. It is required to guarantee that the subscription file will not corrupt, because by replacing a string in a file which length is unequal to the length before it could happen that by overwriting the old entry it is overwriting the following entry.

If the user called the *Cancel* function the window in Figure 39 is shown with all subscribed torrents. There the user can selects simultaneously one or more subscriptions for canceling.

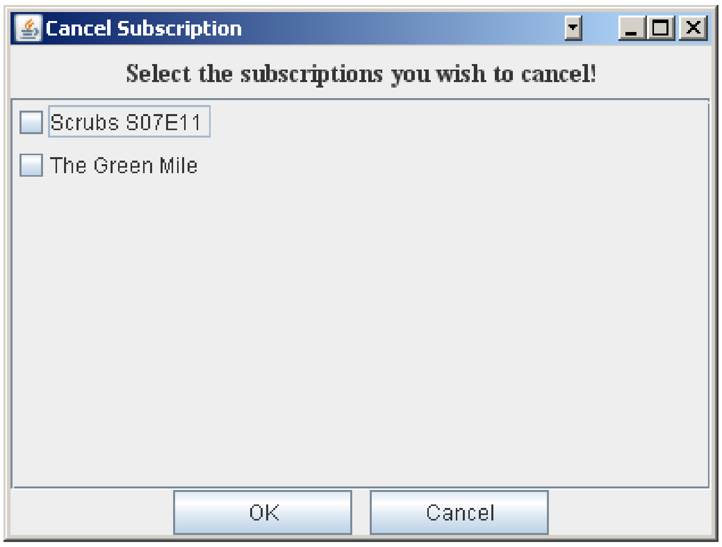


Figure 39: BitThief 0.2.0: Subscription window to cancel subscribed torrents.

The option to subscribe to a TV show automatically is only implement in the search window (Chapter 8). If the user downloads a TV show torrent it will be tested if it is subscribeable. Then the user has the choice to subscribe to this torrent or not. If not, the subscription function will be skipped else a new entry will be added in the subscription file. After this step, the download process starts normally.

The automatic subscription for a specific TV show is only for one season. The reason is it is very tricky or even very hard to predict the last episode of the current season. (Of course it is possible to ask the user: How many episodes will be produced? But many users do not know this and sometimes it changes during the production and publishing.

9.5. Testing

The testing of the subscription functions was split into different aspects. At first I have checked the recognition of subscribeable torrents. For that I have searched for different TV shows to test all known season pattern. In doing so I have detected that the separator, which separates the pattern from the name, can be diverse. The most used pattern is the space character but also the dot or minus. This behavior decreases the number of search results so that it is useful to remove and/or replace this pattern by a standard character. The most efficient character is the space character, thus I replaced all separators by a space. The reason is the space character will be connected to the rest of the word in a search request in contrast to other separators where the provider interprets this as one word and searched for this. In this test phase I checked also, if all not subscribeable torrents will be not recognizes by the automatic subscription function.
In the next step I checked the service function with special attention of the cancel function. Because by canceling a subscription the entry will be removed in the subscription file, and that means the function manipulates the file pointer in both directions and this is a big error source. In this test no failure was detected.
In the next scenario I tested the replace function, for it I have downloaded an offered torrent which was subscribed to and then I have controlled the subscription file if the following episode is added and the old one removed. This function manipulates also the file pointer and thus it is an error source.
In the last case I checked the function which detects if a subscription exists. For this test I have examined two different scenarios. The first scenario was to detect a subscription which was subscribed before. And in the second scenario I checked the case when a user subscribed to more episodes of one season. In that case the function should detect one of it when both subscriptions have reached the same episode counter and then it should remove one entry. Also this test was failure-free and the functions works well.

10. Failure Analysis

The version of *BitThief*, during my thesis, throws some exceptions with are not caused by programmers directly. The first failure group, failure in the heap space, has the result that the program crashes. To find and fix the failures was important for publishing the next version, because we have to deliver a stable version. The biggest problem was the heap space. The client allocates too much memory and therefore the program crashes. To find the cause for it is useful to use a profiler software, which is monitoring the runtime environment during runtime. I used the *jConsole* from Java. The analysis of this failure is described in the next Section (10.1). The second failure group were caused by the charts. These failures are thrown sporadically from the *JFreeChart*[17] library and no analysis tools exist. The procedure of analyzing manually and fixing the problem is described in the Section 10.2.

10.1. Memory failure

When monitoring the client, I have used a popular torrent from the web. A popular torrent can guarantee that there are enough seeders for this torrent. I chose the current version of Ubuntu (Hardy Heron 8.04) as an ISO image CD version with 733 MB and the DVD version with 3.7 GB. After time of monitoring the CD torrent had around 1300 seeders and the DVD torrent had circa 250 seeders.
In Figure 40 the first record of the heap space allocation is shown. At this time the client had up to 255 open connections. The graph shows that the program used more than 220 MB of heap space. The problem is that Java start the virtuell machine with a default heap space size, usually this value is 69 MB on Windows (65 MB for the program and 4 MB for the Java VM) and it depends on the OS.
In the second step I have used the DVD version of Ubuntu. This file is much larger and is supposed to show the relation between torrent size and open connections. At the time of measurement 250 seeders were available and the program opened 125 connections to other BitTorrent clients. This graph is shown in Figure 41 and it can be seen that the allocated heap space is half for size.
Both graphs show that there are no dependencies on the torrent size. But the availability from the torrent is different. The problem is the client opened too many connections. For every open connection the Java VM reserved extra heap space.

[17]JFreeChart is a free (LGPL) chart library for the Java(tm) platform, http://www.jfree.org/ .

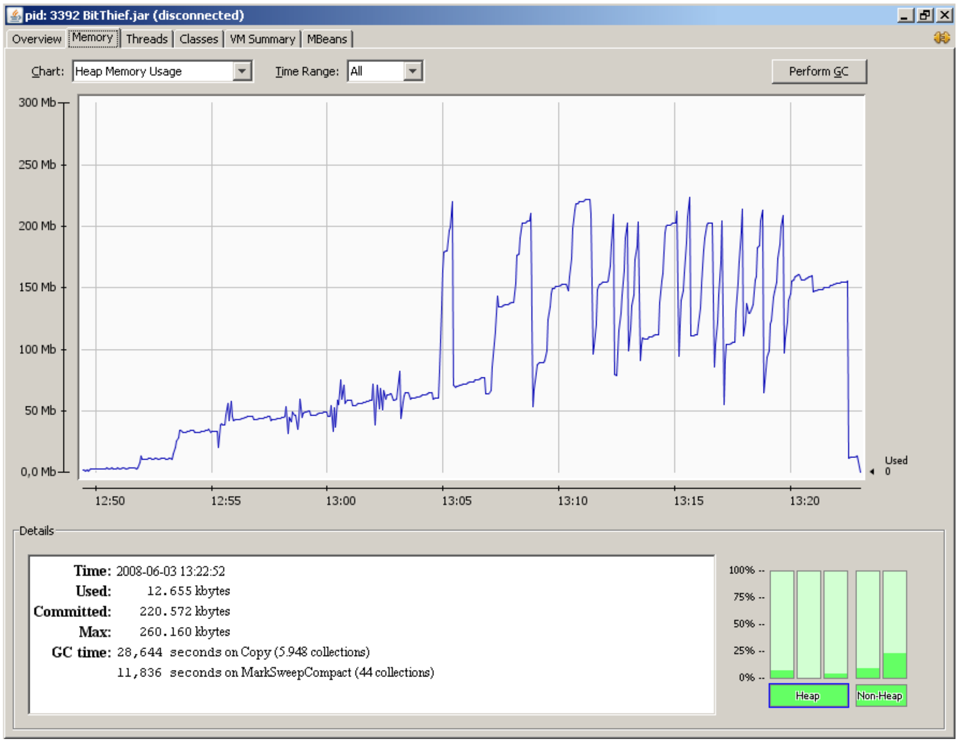


Chart: Heap Memory Usage
Time Range: All

Time: 2008-06-03 13:22:52
Used: 12.655 kbytes
Committed: 220.572 kbytes
Max: 260.160 kbytes
GC time: 28,644 seconds on Copy (5.948 collections)
11,836 seconds on MarkSweepCompact (44 collections)

Figure 40: Allocated heap space with 225 open connections and a 733 MB torrent download.

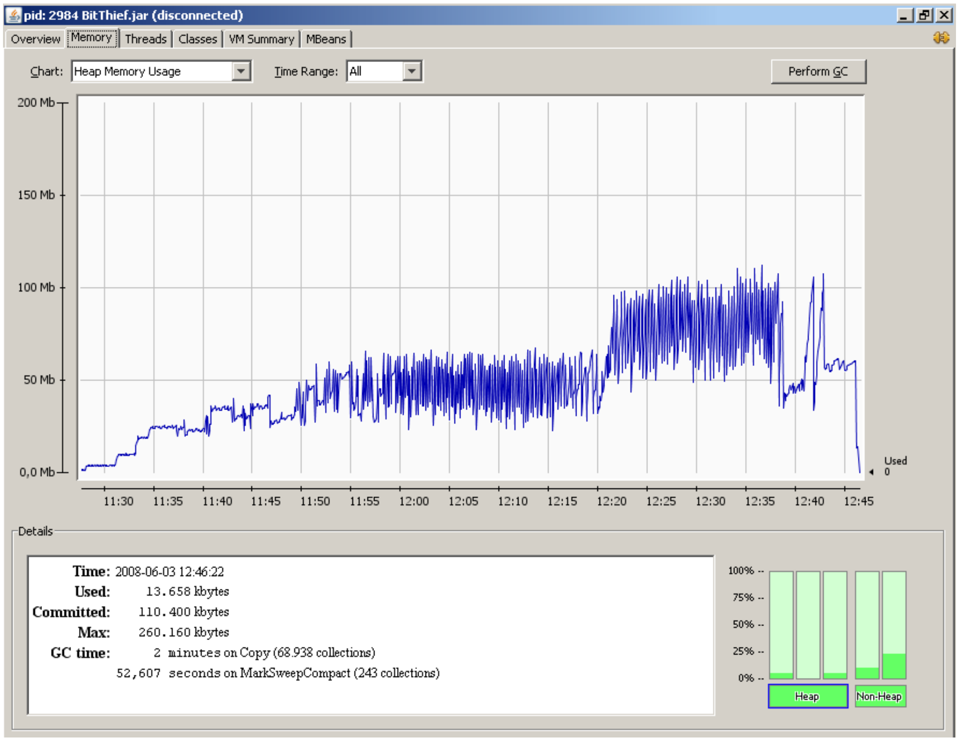


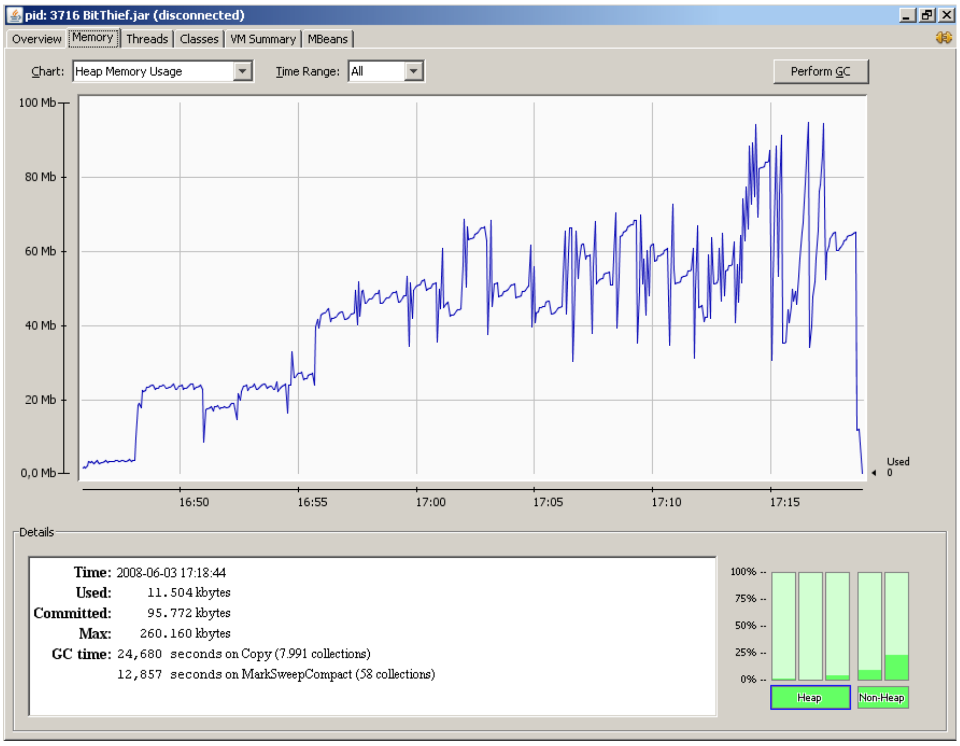
Chart: Heap Memory Usage
Time Range: All

Time: 2008-06-03 12:46:22
Used: 13.658 kbytes
Committed: 110.400 kbytes
Max: 260.160 kbytes
GC time: 2 minutes on Copy (68.938 collections)
52,607 seconds on MarkSweepCompact (243 collections)

Figure 41: Allocated heap space with 125 open connections and a 3.7 GB torrent download.

10.1.1. Result

To show that too many connections need too much memory, I tested it and set a limit for open connections. The result is shown in Figure 42, the connections was limited to at most 65 open connections.



```
      Time: 2008-06-03 17:18:44
      Used:      11.504 kbytes
 Committed:      95.772 kbytes
       Max:     260.160 kbytes
   GC time: 24,680  seconds on Copy (7.991 collections)
            12,857  seconds on MarkSweepCompact (58 collections)
```

Figure 42: Allocated heap space with maximal 65 open connections and a 733 MB torrent download.

In the graph it shows that the space is half in comparison to the first run. To guarantee a run without space failure the client was restricted to maximum of 45 open connections per default. To retain the advantage that *BitThief* could open much more connections as other BitTorrent clients the number of the maximum connections can be changed by the user. The aim of the restricted connection by default is that the client can run with the default heap space size from the Java VM when downloading one torrent. If the user wants more connections he has to guarantee enough heap space size. Further, the cause is detected and the developers of *BitThief* have to modify the client in a way such that the network part uses less memory. That can be realized in different ways. First the client must unload all currently unused classes. Second use a network library which used not so much heap space as the Java tool. And the client must swap out the received data in special files.

10.2. Chart failure

The *BitThief* client used 10 different charts for analyzing the download performance. For painting the charts the client used the *JFreeChart* library. The client version which is current in the developing state throws some errors, which are appear by painting the charts, sporadically. To find the cause I have performed some scenarios by hand because there are not existing tools for analyzing. A debugger was not helpful because the effort of implementation the source code in our project suit was too much. The failures which were thrown by the library is an "index out of bounds exception" by Java arrays and are originated during developing without changing the relevante parts (i.e. changes in the chart functions).
In the first phase I had try to catch the failure in our client. But this was not succeeds because the exception came not through to the client. Thats denote that this failure are not caused by a programmer from the BitThief project. In the next step I had isolate the functions in the library source code, where got thrown the exception and tried several ways to avoid these failures. I had localized two classes from the lib, the *TimeSeries* and *TimeSeriesCollection*. In this classes three different functions exist where the failures come from (*getDataItem* from the TimeSeries class and *getXValue* and *getY* from the TimeSeriesCollection class). In all three functions the failure is thrown when recalling a data item from a list. How can it happen that a recall calls a value out of bounds? One reason could be that a thread, who set the value, lost its time slice. To prove this theory I set a sleep time for 10 ms in the failure case and then the function tried again for the recall. After that change the failures were removed.
The explanation is on the one hand the theory which I explained before. On the other hand, why the failure did not appear before is that the source code of *BitThief* increased in size and thereby the run time increased as well.

11. Conclusion

At the end of my thesis the useability is increased enormously so that *BitThief* can match with other popular *BitTorrent* clients. The client has an update function what is fundamental for the scientific work, because it guaranteed that most users works with the newest version and so the scientific analysis are in a representative context.
Now the client consist of new unique features. e.g the subscription service, the torrent search function including torrent ranking and the choice to select single files of a multi torrent. This features are important to appealing more users and therefore to increase the *BitThief* community. Furthermore with a growing user community would appear a higher correctness of the statistical analysis and hence to get a more correct conclusion.

References

[1] Bram Cohen. Incentives Build Robustness in BitTorrent. *http://www2.sims.berkeley.edu/research/conferences/p2pecon/papers/s4-cohen.pdf*, 2003.

[2] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. *http://www.sigcomm.org/HotNets-V/locher06free.pdf*, 2006.

[3] Bram Cohen. The BitTorrent Protocol Specification. *http://www.bittorrent.org/beps/bep_0003.html*, February 2008.

[4] Bittorrent specification. *http://wiki.theory.org/BitTorrentSpecification*.

[5] Ali Ghodsi. Distributed k-ary system: Algorithms for distributed hash tables. *http://www.sics.se/~ali/thesis/dks.pdf*, 2006.

[6] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system. *http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf*, 2002.

[7] Patrick Moor. Free riding in bittorrent and countermeasures. Master's thesis, Swiss Federal Institute of Technology Zurich, Summer 2006.

[8] Thomas Locher, Stefan Schmid, and Roger Wattenhofer. Rescuing tit-for-tat with source coding. *http://dcg.ethz.ch/publications/p2p07.pdf*, 2007.

[9] David Harrison and Bram Cohen. Fast extension. *http://www.bittorrent.org/beps/bep_0006.html*, 2008.

A. Appendix

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file represent the individual search
        options for different websites-->
<begin>
<name>Mininova</name>
        <url>http://www.mininova.org/search/</url>
        <separator> </separator>
        <option>All</option>
                <value>0</value>
        <option>Featured</option>
                <value>10</value>
        <option>Anime</option>
                <value>1</value>
        <option>Books</option>
                <value>2</value>
        <option>Games</option>
                <value>3</value>
        <option>Movies</option>
                <value>4</value>
        <option>Music</option>
                <value>5</value>
        <option>Pictures</option>
                <value>6</value>
        <option>Software</option>
                <value>7</value>
        <option>TV Shows</option>
                <value>8</value>
        <option>Other</option>
                <value>9</value>
        <sort>Date</sort>
                <value>added</value>
        <sort>Name</sort>
                <value>name</value>
        <sort>Size</sort>
                <value>size</value>
        <sort>Seeds</sort>
                <value>seeds</value>
        <sort>Leechers</sort>
                <value>leech</value>
        <sort>Type</sort>
                <value>cat</value>
<name>The Pirate Bay</name>
        <url>http://thepiratebay.org/search/</url>
        <separator>,</separator>
        <option>All</option>
                <value>0</value>
        <option>Music</option>
                <value>100</value>
        <option>Movies</option>
                <value>200</value>
        <option>TV Shows</option>
                <value>205</value>
        <option>Software</option>
                <value>300</value>
        <option>Games</option>
                <value>400</value>
        <option>Other</option>
                <value>600</value>
        <option>Books</option>
                <value>601</value>
        <sort>Name</sort>
                <value>1</value>
        <sort>Date</sort>
                <value>3</value>
        <sort>Seeds</sort>
                <value>7</value>
        <sort>Leechers</sort>
                <value>9</value>
        <sort>Type</sort>
                <value>14</value>
<end>
```

Listing 7: Content *searchOptions.setting* file

B. Appendix

Appendix B contains the project CD. All project related stuff is available on CD.

Declaration of Authorship

I certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other University.

Reading, 28/09/2008

Ben Hennig