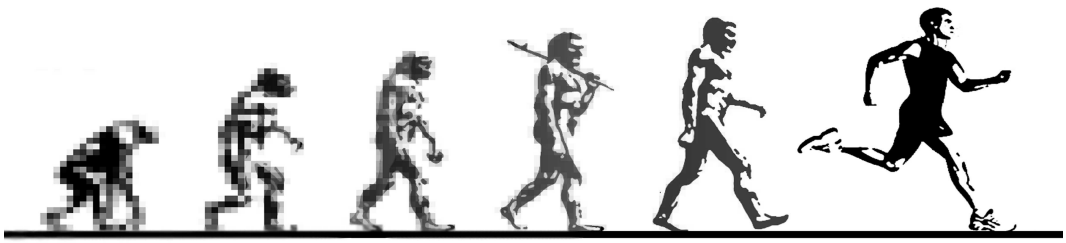
Multi Bit-Rate Video on Demand for P2P networks

Master's Thesis



Riccardo Petrocco

Multi Bit-Rate Video on Demand for P2P networks

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Riccardo Petrocco
born in Rome, Italy

Tribler Research Group
Department of Parallel and Distributed Systems
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.pds.ewi.tudelft.nl

Cover picture: A. Petrocco, increasing speed - increasing quality.

Multi Bit-Rate Video on Demand for P2P networks

Author: Riccardo Petrocco
Student id: et1339702
Email: r.petrocco@gmail.com

Abstract

The Internet has become in the last years more and more a means of conveyance for multimedia delivering. Many solutions have been proposed to gain high quality of service for video on demand in client-server environments, with adaptive algorithms that adjust the bit-rate of a video stream depending on the client's available bandwidth. Providing video on demand over decentralized peer-to-peer systems is an active research field. The variable bit-rate environment that characterizes peer-to-peer networks causes significant difficulties to ensure quality of service and playback continuity for video on demand applications.

This thesis addresses the challenge of serving high quality video on demand by designing and implementing a multi bit-rate video on demand architecture for peer-to-peer networks. We propose a switching scheme, an encoding methodology and a novel algorithm for multiple bit-rate video streaming over peer-to-peer networks. Identical video content is encoded into three different sets of streams with different average bit-rates. The novel multi bit-rate algorithm will switch dynamically between the three sets of streams depending on the available bandwidth. Through a series of experiments we present the effectiveness of this architecture in fluctuating bandwidth scenarios.

Thesis Committee:

Chair:	prof. Dr. Ir. H. J. Sips, Faculty EEMCS, TU Delft
University supervisor:	Dr. J. Pouwelse, Faculty EEMCS, TU Delft
Committee Member:	Dr. F. A. Kuipers, Faculty EEMCS, TU Delft

Preface

This document describes my MSc thesis research concerning Video on Demand in a variable bit-rate environment for the Tribler peer-to-peer network. The research was performed at the Parallel and Distributed Systems Group of the Faculty of Electrical Engineering, Mathematics, and Computer Science of Delft University of Technology.

I would like to thank all the members of the Tribler research group for the support and the nice working environment. I am also grateful to Dr. Ir. Johan Pouwelse for the support and continuous incentive to ”go further”, Dr. Ir. Jacco Taal and Arno Bakker for the support in designing the architecture, Dr. Jan David Mol for the support and the helpfull discussions about codecs and encoding formats. Special thanks go to Tamas Vinko and Victoria Perez for the continuous feedbacks through out the writing phase of the thesis. Furthermore I would like to thank prof. Dr. Ir. H. J. Sips for chairing the examination committee, and Dr. F. A. Kuipers for participating in the examination committee.

Riccardo Petrocco
Delft, the Netherlands
November 17, 2008

Contents

List of Figures

Chapter 1

Introduction

Peer-to-peer (P2P) has been, for years, a small field of research in computer architecture. The first networks with such an approach were studied back in the'60s for some of their properties considered already interesting at that time, such as the ability to operate in a completely decentralized way, the computational power associated with such an architecture and other characteristics.

Initially P2P networks were investigated only in scientific and academic environments, only later some important companies such as IBM and Sun understood the potential of this architecture. The architecture was widely developed just in the early years, figure 1.1, due to one of its strongest characteristics: the *file sharing*.

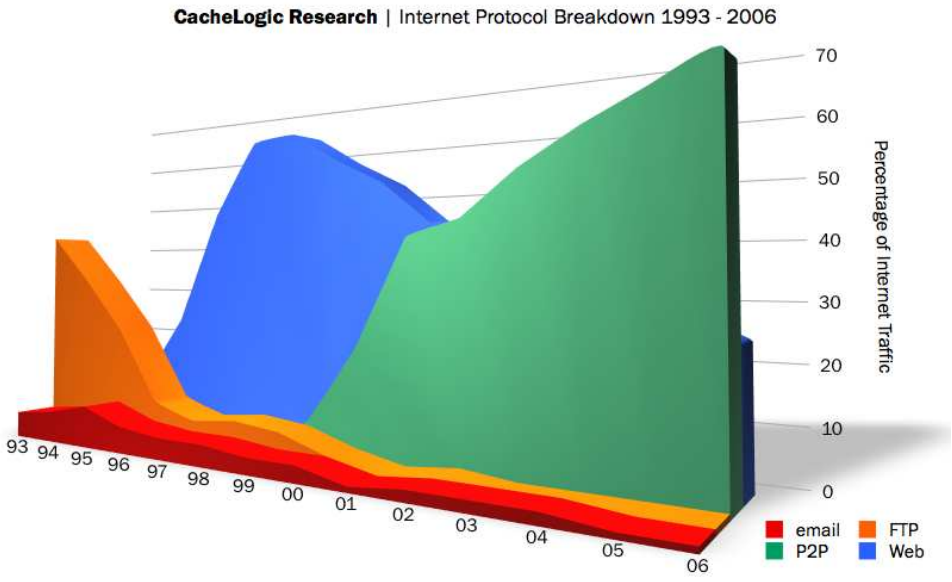


Figure 1.1: The increasing popularity of P2P systems

P2P networks have reached this success thanks to the large scale content distribution. Most of the P2P applications allow to download large amount of data in a fast way. Currently most of the P2P traffic is used to distribute video files, normally a user has to download the video file before he will be able to watch it, suffering from a long start-up time. Recently we have seen the emergence of many systems that integrate the P2P network with a client/server architecture, for delivering of multimedia content, reducing the server's workload [12][26][41][10][39].

The current solutions to provide VoD over P2P networks, we will see later the Swarmplayer, present constrains regarding the bandwidth availability. For the survival of the VoD functionality over a P2P network users could download only as much as they upload, restricting users with a low upload bandwidth such as ADSL users.

In this thesis we propose a novel solution for serving VoD over P2P networks, characterized by a variable bandwidth environment. Our Multi Bit-Rate, MBR, VoD solution switches between three different quality streams depending on the available bandwidth. It has shown to behave good in bandwidth-fluctuate environments, when a constant bandwidth is not guaranteed, and in stable environments.

In this introduction chapter we will first give an introduction on P2P networks in section 1.1. In section 1.2 we will present BitTorrent, and then Tribler as an enhancement of it in section 1.3. Finally in section 1.4 we will introduce Tribler's Video-on-Demand functionality and in section 1.5 the motivations for a MBR VoD implementation are discussed.

1.1 P2P Networks

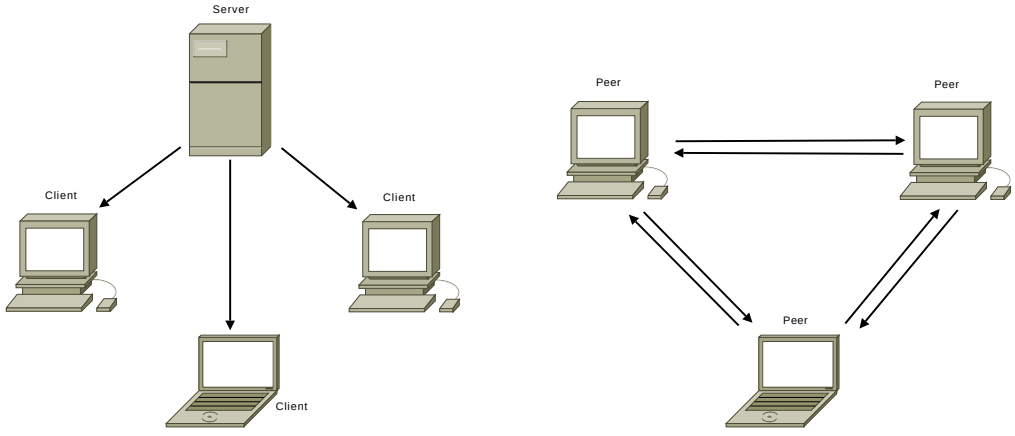
The common architecture designed for computer communication is generally a client-server architecture, figure 4.6a. The most famous system designed with a server-client approach is the world wide web. This scenario is characterized by a central server that handles the client's requests. This approach lacks of modularity, scalability and reliability. By the increasing of requests, servers need to adapt by increasing the available resources, such as available bandwidth and computational power, to satisfy the incoming requests. A server could be a point of failure and could stop handling the incoming requests, making the system unusable. On the other side P2P systems offer a solution to gain scalability and reliability in computer communications.

Typically a P2P network is a computer network or any kind of network that does not use a client-server approach, but an equivalent number of nodes (called peers) that serve both as client and as server to other nodes. This network model is the antithesis of a client-server architecture. Through this configuration any node is able to start or complete a transaction. The equivalent nodes may differ in the local configuration, the processing speed, bandwidth and variations in the amount of stored data. In general we define the term P2P network as two or more computers in which all computers occupy the same hierarchy. This modality is normally known with the term Working Group, against the networks where there is a central

domain.

In a P2P system peers communicate using symmetric protocols and they act both as server and as client by sending and handling requests. The P2P research has gone along over the years with the economical field of game theory. The reason for this is that a P2P system has to provide enough incentives for the peers to share the resources with each other. While on one hand P2P systems, in relation to client-server systems, are generally more reliable considering the possibility of a node failure on the other hand peers of a network are less reliable than servers in terms of tastiness and security.

The classic example of a P2P network is a network to share files (File sharing). Over the last years we have seen the emergence of an incredible amount of P2P file sharing networks often including servers for particular functionalities. Often P2P file sharing application use the support of servers, ending up in a mixed scenario where the P2P architecture and the client-server architecture coexist. P2P file sharing clients such as BitTorrent [20] or FastTrack, implemented in Kazaa [24], are based on P2P networks but still use some servers for locating files of connected peers. Other interesting P2P systems are the Gnutella [18] and the Kad Network [1] that implemented a completely serverless network. Anyway those networks still need to know before hand some peers of the network. As an example the popular program eMule [14] takes advantage of both: servers for file indexing and the Kad network for additional sources. With the increasing popularity of P2P file-sharing networks, systems such as the Kad Network demonstrate to react better than server based architecture in localizing peers with content. Clearly the respond time is higher, given by the distributed nature of the architecture. Innovative uses of the P2P technology include the deployment of real-time generated high data streams such as television programs or movies.



(a) Client-server (b) P2P

Figure 1.2: The centralized client-server architecture versus the distributed P2P architecture

[1] that actually implements the Kademlia P2P overlay protocol [23]

1.2 BitTorrent

We will now introduce the BitTorrent P2P system, designed by Bram Cohen in 2003, as the core of Trible's architecture. BitTorrent is both a protocol used for communications in a P2P fashion, and a client that uses the protocol for file sharing. It has been designed to allow a fast download of big files over a P2P network, limiting the bandwidth consumption. The protocol is based on an encryption algorithm, called Bencode, used for client/server and client/client communications.

Unlike traditional file sharing systems, the goal of BitTorrent is to create and provide an efficient system to distribute the same file to the largest number of available peers. This is a mechanism to automatically coordinate the work of a multitude of computers, obtaining the best possible common benefit. BitTorrent is a protocol that allows to distribute files of any type. To facilitate the transmission, the original document is split into many small fragments, called chunks, which then will be recomposed once at destination. The chunks have a fixed size, for verification a fingerprint [2] for every chunk is generated, using the SHA1 algorithm, and distributed along the peers.

Every file-sharing program designed on a P2P network need some share-ratio [3] enforcements to guarantee the survival of the system. The sharing-ratio enforcement is the set of rules that enforce peers to share their upload bandwidth with other peers of the system. In BitTorrent, the share-ration enforcement is guaranteed by the tit-for-tat [9] protocol, that tries to gain a high sharing-ratio between peers. Tit-for-tat is designed in a bidirectional way, a peer will be able to download from another peer if it is uploading some content to that one. This gives peers an incentive to be available and donate bandwidth to the BitTorrent network. Furthermore it gives a solution for the freeriding problem by stimulating cooperation [2].

The research of available content is done in a centralized way through web sites where *.torrent* files are located. The torrent file is a simple file, small, which can be published for example on a Web page. In order to take advantage of the system, it is therefore necessary, first of all, to download a file with the .torrent extension. This file acts as an index, with a description of all packages in which the original file was divided, including hash keys that ensure the integrity of the various pieces. The torrent file contains the address of a BitTorrent tracker. The tracker is used to discover the connection properties of the group of downloaders of this torrent. The total group of downloaders of a torrent is called the download swarm.

[2] The fingerprint in computer science is a string that identifies a given file. It is used to ensure the authenticity and security of files and also to quickly identify files distributed over a file-sharing network.

[3] ratio between the total amount of uploaded data and the total amount of downloaded data of a peer in the network.

1.3 Tribler

Tribler [38] is the name of a software designed and implemented since February 2006 in the Parallel and Distributed System group of the Faculty of Electrical Engineering, Mathematics and Computer Science of TU Delft. Initially only a small enhancement of the ABC client [1], it now integrates many functionalities that make this software unique.

Tribler differs from other popular BitTorrent clients such as Vuze [47] and uTorrent [45] due to some of its features. Tribler adds keyword search ability to the BitTorrent file download protocol using a gossip protocol. The software includes the ability to recommend content. After a dozen downloads the Tribler software can roughly estimate the download taste of the user and recommends content. This feature is based on collaborative filtering, also featured on websites such as Last.fm and Amazon.com. Another feature of Tribler is a limited form of social networking and donation of upload capacity. Tribler includes the ability to mark specific users as online friends. Such friends can be used to increase the download speed of files by using their upload capacity [5]. The last evolution of the software ingrates new functionalities to prevent free-riders and garantee fairness [33].

1.4 Swarmplayer

As the GUI of Tribler, the Swarmplayer is just an interface to Triber's API, see figure 1.3, that enhances Tribler with the VoD functionality. The Swarmplayer is responsible of handling the download and manage the video playback. It has been designed for the integration in web pages, in Tribler's GUI or as stand alone player. To obtain the VoD functionality some characteristics of Tribler, more precisely of the BitTorrent client, had to be modified. For this purpose a new algorithm, called Give-to-Get [34], has been designed and implemented to handle the download and upload policies.

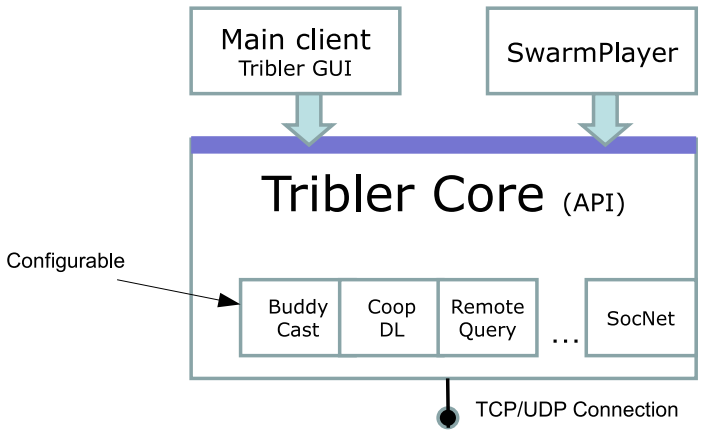


Figure 1.3: Tribler Core

Figure 1.4: The interface of the Swarmplayer

We implemented the novel VoD approach described in this thesis into the Swarmplayer. The high modularity of Tribler's architecture allowed to easily enhance the Swarmplayer with new functionalities in a transparent way for the existing VoD implementation.

1.5 Multi Bit-Rate Video on Demand

The poor quality and the typology of the current Internet accesses are a great limit for the spread of a VoD P2P technology. In our days Internet accesses are mostly ADSL and normally the available download bandwidth is much higher than the upload bandwidth. While it is prefered to have a higher download ratio when serving on the web, this is a limitation when considering P2P systems. Considering the VoD functionality discussed in the previous section a peer could only download as much as it uploads for the survival of the system. It is logical that, in a completely decentralized system, a peer can rely only on other peers of the system and therefore on their upload bandwidth.

Another limitation of the current VoD functionality is that it relies on the current download rate of a peer. While this is applicable in a client-server architecture, where servers provide a fixed bandwidth, it presents problems due to the uncertainty of the available bandwidth in P2P systems. In P2P systems the available bandwidth depends on the amount of connected peers and on their connections. If a peer drops out of the system, while we are

watching a video file in a VoD fashion, it can stall the player's playback because of the sudden drop of available bandwidth.

In this thesis we present design and implementation of a novel VoD functionality, called Multi Bit-Rate VoD, that aims to smartly react to sudden bandwidth variations. The Multi Bit-Rate (MBR) VoD functionality will switch between three different quality streams depending on the current available bandwidth. We designed a novel encoding methodology to create the three quality streams in a particular way. Furthermore we modified the multimedia player VLC to allow a flicker-free playback of our encoded streams.

1.6 Contributions

The contributions of this thesis are as follow:

- We study the current VoD solutions and their design decisions. In particular we analyse the state of the art of codecs and containers that could allow a MBR VoD functionality to be implemented.
- We propose a novel encoding methodology that creates a torrent with three different quality streams, aimed to serve our novel MBR VoD solution.
- We modified the demuxer module of the multimedia player VLC to allow a flicker free playback of chained Ogg streams [4].
- We present a novel algorithm that handles the download and playback of three proportional quality streams of a torrent file, depending on the available bandwidth.

1.7 Thesis outline

The remaining parts of the thesis are organized as follows. Chapter 2 will describe the problems related with the current P2P VoD solutions and with the approaches aimed to gain a VoD functionality in variable bandwidth environments. Chapter 3 will present the design of our novel MBR VoD architecture. In this chapter we will discuss the design decisions that guided us through the nine month thesis project. In Chapter 4 we will show how the design of Chapter 3 has been implemented by enhancing Tribler's core and the Swarmplayer. In Chapter 5 we present the result of our experiments that shows the behaviour of our novel MBR VoD functionality. Conclusions, discussions and recommendation for further investigations are presented in Chapter **??**.

[4] a stream created by concatenating different streams

Chapter 2

Problem description

First we consider the problems related with the Quality of Service, QOS, of video on demand for peer-to-peer networks.

In section 2.1 we will discuss how the network condition of the Internet is not reliable because the bandwidth and the load of Internet often change acutely. But the transport bitrate of media source is mostly constant. So it will affect the quality of Video-on-Demand such as delay and jitter. Further more the actual access bandwidths of many users, peers in our case, who attempt to watch the same media program are different. Some users with high access bandwidth can't get better video quality and some users with very low access bandwidth have not enough bandwidth to watch video. So it is necessary that media source can provide more than one bitrate to adapt to complex network condition.

In section 2.2 two different approaches to stream over a bandwidth-fluctuate network are discussed. The enormous attention that variable quality video-on-demand got over the last years brought to such encoding technologies as the *scalable video coding*, that would avoid handling big restrictions, explained in section 2.3

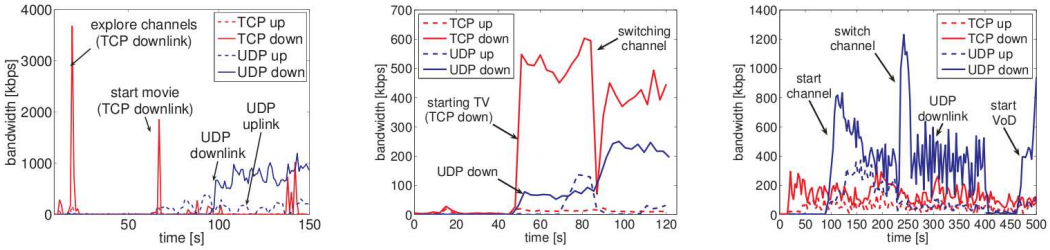
2.1 Heterogeneous Internet access

The Internet is constantly growing, and the connections speed with it. The nature of the environment is not reliable because bandwidth and load could change unexpectedly. Also the access bandwidths of the users, peers in our case, differ. Some peers might have an access bandwidth that is in order of Megabytes/s while other peers hundreds of Kilobytes/s. This diversification is the most important factor that brings the need in our days of a variable bandwidth environment where the Quality of Service is guaranteed. The current working solutions are majority server based [26] [41].

The simplest way is that media server prepares several media files for the same video file. The bitrates of these files are different from each other. The server redirect a client to a corresponding media file according to the client's selection on the bitrate.

The current services on the Internet that provide a Video on Demand (VoD) functionality

have been analysed to determine which were their design decisions concerning bandwidth allocation. Our analysis takes in consideration YouTube as the most popular VoD service but there are at least other 40 video-sharing websites [1]. We will show also "Joost", "Zattoo" and PPLive as the most popular video on demand distributed systems based on P2P technology. The results shown in figure 2.1 and figure 2.2 are taken from [21], where those popular video content delivery mechanism are studied. P2P technology has proved to be the future solution for VoD systems, offering a better reaction to a sudden increase in requests. On the other hand setting up the environment might take longer than in a server/client architecture when requests and content distribution over the P2P system is rather low [27].



(a) Joost (b) Zattoo (c) PPLive

Figure 2.1: Bandwidth consumption over time during P2PTV measurements

	TCP	UDP	up	down	TCP(up)	TCP(down)	UDP(up)	UDP(down)
Joost (short)	86.12	391.01	91.46	356.14	9.08	77.04	88.99	302.15
Joost Movie	9.85	546.75	69.24	487.08	3.02	6.82	66.32	480.48
Joost (Japan)	9.07	522.78	12.88	516.33	1.06	8.01	11.84	508.77
Zattoo (short)	285.15	104.27	28.68	359.51	11.44	273.71	19.06	86.83
Zattoo show	578.11	92.18	108.85	561.12	17.67	560.44	91.47	0.68
PPLive (short)	209.29	479.17	94.26	582.08	44.68	164.63	50.90	428.44
PPLive	117.95	586.80	196.43	502.29	42.22	75.75	155.82	430.95
PPLive (Japan)	159.73	547.79	196.69	509.95	42.15	117.59	154.58	392.73
YouTube	326.63	0.21	11.19	315.49	11.17	315.47	0.02	0.02

Figure 2.2: Measured bandwidth consumption in [Kbps] for different IPTV systems

As we can see from figure 2.3 YouTube is the VoD system that requires less bandwidth consumption. It needs at least a 315,47 Kbps as download bandwidth to be able to watch a video stream without stalling, see figure 2.2. This bandwidth consumptions are taken into account to design our quality streams in a proper way. We are going to see, chapter 3.1.1.2, how our lowest quality stream will need only a 256 kbps connection to be able of being watched fluently. Therefore all the users with a low speed connection will benefit from our architecture, without having to wait a certain time for the buffer pre-fill.

[1] `http://en.wikipedia.org/wiki/List_of_video_sharing_websites`

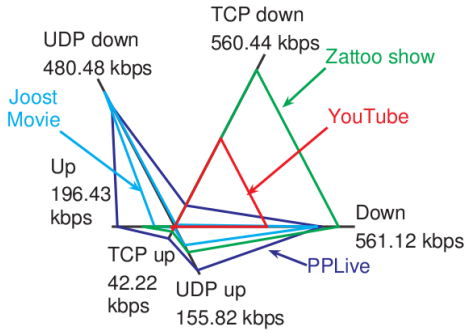


Figure 2.3: Spider plot of the bandwidth consumption of the five investigated IPTV applications

2.2 Streaming over bandwidth-fluctuate networks

Internet is considered a bandwidth-fluctuate network where heterogeneous connection speeds coexist and interact with each other. While the difference is not evident in a server-client environment, it is one of the biggest problems in a peer-to-peer or client-client environment. In a P2P environment the connection speed of a peer relies only on the connection speed of other peers, making the bandwidth capacity of a P2P system variable depending on the connected. It often happens that the upload bandwidth of a peer completely saturates the bandwidth needs of a second peer, while it will need a large set of peers to saturate it's bandwidth. This instability is the key point for a multi bit-rate implementation. Not only peers with low and medium connection speed benefit but also peers with a high speed connection that want to watch a video file held by peers with a slower connection.

Before presenting our final solution to gain a Multi Bit-Rate (MBR) video on demand functionality, we will analyse the state of the art for variable bit-rate encoding. The next subsection will present a possible approach based on scalable coding where an encoded video file is composed of a base layer and a certain number of extra information layers that improve the video quality. The following chapter will present our novel solution that, seeking for a codec agnostic approach (considering the limitations explained in section 2.3), encodes the video file into three different quality streams and switches at run-time between those depending on the available bandwidth.

2.2.1 Scalable video coding approach

One possible solution for streaming over bandwidth-fluctuate networks is the *scalable video coding* approach. Since 2003 when the Moving Pictures Experts Group (MPEG) made a proposal for scalable video coding (SVC) a lot of effort has been put in that direction. SVC will became the name given to an extension of the H.264/MPEG-4 AVC video compression

standard. The SVC has been developed to offer the possibility of encoding into an high quality bit stream split into different sub streams or layers that can be decoded independently. Such an approach will offer a simple solition to archive a variable bit-rate video on demand system where the base bitstream will be encoded to advantage low bit-rate clients and additional streams could be downloaded by faster peers. Even though the fact that this will be the best solution in terms of modularity, scalability etc., there is still no open-source implementation. The current free software encoding library corresponding to the H.264, that could be used to encode the original video, is the X.264. It is also used by the VideoLan player (VLC, discussed in section 3.2.2) that is the current video player used in the Tribler project for the Video on Demand implementation. The big problem concerning this implementation is that there is still no implementation of the X.264/SVC (scalable) extension and we will take this approach into consideration only once the open-source implementation will be available. Of course as soon as it is a stable technology to relay on it has to be taken in consideration as a valid alternative. Therefore in the next sections and for the rest of the research a Scalable Video Coding approach is not taken into consideration

2.2.2 Switching points, a novel approach for changing quality

The novel approach proposed in this thesis is a *switching points* technique. This technique aims to switch at run-time between different quality streams. Every stream stored in a different buffer, therefore to switch between different quality buffers. How to switch between buffers will be handled by a controller that will try to switch to an higher quality level as soon as there is enough saved buffer of the current quality, see figure 2.4.

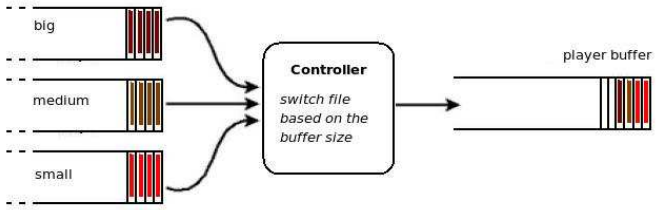


Figure 2.4: Multi bit-rate controller

The major problem deriving from this design is the alignment between quality buffers. If the buffers holds different qualities, than the informations held at a particular time will not be the same in time for other buffers. How to switch between buffers regards the ”switching point problematic” and will be addressed in section 3.2.

The switching point problem has different impacts at different levels of the architecture. The download engine, BitTorrent **??**, will be affected in the download policy. The new policy will have to download the needed quality based on the available bandwidth. This is easily archived in a client-server environment where the server sends a quality stream depending on the client feedback. This interaction between peers is not possible in our P2P

environment, therefore the ”client” peer will have to adapt regardless of other peers, in an automatic way.

The MBR Controller will have to fill in the right way the three quality buffers and than serve them to the video player. The algorithm has the responability of preventing stalling and determining when to attempts a quality switch.

The last but heaviest problem concerns the video playback, as we are going to see in the next chapter, where getting an alignment between the streams resulted in a complicate analysis of different container and codec formats, that will allow to manage streams in the desired way.

2.3 Video standard restrictions

As previously discussed, the main problem concerning the video playback is the quality buffers alignment. Thanks to the previous work [34] we know that downloading a torrent file in an ordered way, with only one video file, and streaming it to a multimedia player is possible to realize. This approach is quite useful considering only one stream. On the other side, considering such an approach for handling different streams involves some changes on the player side.

The major characteristic of the current video on demand functionality is a constant bit-trate encoding. Through the bit-rate estimation it is possible to easily predict when to start playing a video depending on the current download bit-rate. If the current download [2] bit-rate is higher than the playing bit-rate than it is possible to start watching the video directly, without having to wait for a buffer pre-fill. In other words if less time is needed to download the video than to watch it, than it is possible to watch it downloading the video pieces in an ordered way. Even if the download speed is lower than the needed time, through the constant bit-rate encoding it is possible to predict when it will be possible to start watching the video without future ”predicted” interruptions.

This entire approach is based on a bit-rate value that can be stored in the torrent header or automatically detected by analysing the video headers stored at the beginning of the stream. Anyway such an approach will not be valid in a variable bit-rate environment without restricting the codec choice. We need at least different bit-rates for different qualities to be more flexible when encoding the video file. On the other hand as we will see in the next section a constant bit-rate encoding would solve some alignment issues to switch buffers.

2.3.1 Frames structure

A movie is mostly composed of at least two streams: one for the audio and one for the video. Our concerns goes more for the video stream considering that most of the encoded information is carried here. Even though the audio has to be considered when encoding to

[2]The current download measurement is always an approximation of the average over a certain period of time

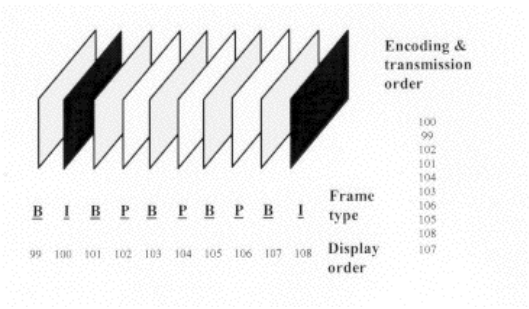


Figure 2.5: Group of pictures

different qualities and synchronised with the video, what plays the biggest rule for a quality difference is the video stream.

MPEG is the most common family of standards used for coding audio-visual information. Each MPEG-coded video stream consists of successive Group of Pictures, GOPs figure 2.5. From the MPEG pictures contained in it the visible frames are generated. A GOP can contain the following picture types [31][36]:

- I-picture or I-frame (intra coded picture) reference picture, corresponds to a fixed image and is independent of other picture types. Each GOP begins with this type of picture.
- P-picture or P-frame (predictive coded picture) contains motion-compensated difference information from the preceding I- or P-frame.
- B-picture or B-frame (bidirectionally predictive coded picture) contains difference information from the preceding and following I- or P-frame within a GOP.
- D-picture or D-frame (DC direct coded picture) serves the fast advance.

The I-frames contain the full image, they don't require any additional information to reconstruct the image. Therefore any errors in the streams are corrected by the next I-frame (an error in the I-frame propagates until the next I-frame). Errors in the P-frames could propagate until the next I-frame. B-frames do not propagate errors.

The more I-frames the MPEG stream has, the more it is editable. However, having more I-frames increases the stream size. In order to save bandwidth and disk space, videos prepared for Internet broadcast often have only one I-frame per GOP.

The most used are the I/B/P-frames and concerning our switching point problem when to switch between buffers we should always start with a new clean image provided by the right I-frame. If we would just jump in the middle of the buffer without taking in consideration the beginning of a group of picture we'll more probably end in the middle of a GOP reproducing P-frames and B-Frames that do not have the correct I-frame as reference.

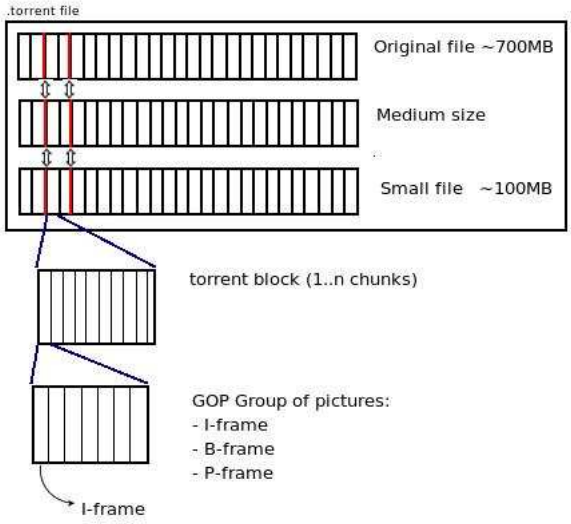


Figure 2.6: Switching point in concomitance with new I-frame

Considering our switching points problem, to avoid the error propagation caused by a missing I-frame, we should set our switching points in concomitance with the beginning of GOPs, see figure 2.6.

A constant bit-rate encoding will help this approach offering a way of localizing the I-frames in a video. Also some particular encoding configuration would allow to easily locate the I-frames in a stream but it will give a big limitation in the kind of codec to be used and in the flexibility of the encoding.

2.3.2 Codec Headers

The audio and video coded headers contain vital information for decoding such as frame rate and resolution that, if changed without re-initialising the decoders, will cause problems in our implementation. A big problem, when analysing the different approaches, is how to handle thos codec headers. Every video and audio stream will be decoded based on the information held in the header. Different container formats offer different header structure but normally the codec informations are stored at the beginning of the video file. Those information is used by video players, in particular by demuxers (see section 3.2.3) to initialize decoders that will handle the stream at run-time, figure 2.7 [3]. In most containers the codec headers are saved only at the beginning and some times at the end of the stream because during the video playback the streams information are not planed to change. In our situation we will have to change headers information at run-time depending on the available downloaded quality. One solution for the header localization would be to download the initial

[3]picture taken from: `http://cutebugs.net/files/multimedia/decoder.png`

bytes(ref, meaning) of every quality stream and store them to be used once is needed.

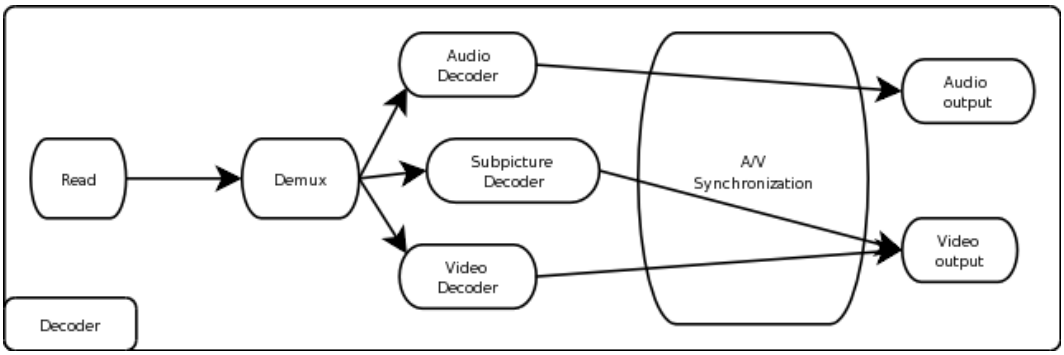


Figure 2.7: Demuxer and decodes architecture

Another problem that derives from such an approach is the *presentation time stamp* (PTS) and the *decoder time stamp* (DTP) synchronization that provides the decoders with information concerning ”when to decode” and ”when to present” every frame of a video stream [6]. That information is needed because often we will have a decoder order that differs from the presentation order.

The amount of frames will change with the quality, therefore even if the same I-frame appears in different buffers [4], it will never have the same values of PTS and DTS because the decoder will have already processed a different amount of frames. This problem could be solved be changing the PTS and DTS values at run-time, but will bring an extra amount of computational overhead to the system. Another aspect that has to be taken into account is that more ”hard coded” is the solution less modularity will be given to the architecture. The proposed solution, section 3.2.3.2, will not handle the video stream for modifications at run-time.

[4] happens only if specifically encoded

Chapter 3

Design

The architecture design was done by dividing the study in relation to the issues addressed. The first half of the time spent on the project was used to study a possible solution to the switching point policy. We obtained the simplest possible switching point solution using the characteristics of containers and codecs. This has led to a profound analysis of the multimedia player VLC [46] used in Tribler and after various architectural choices to a small backwards compatible modification to the software.

To obtain a P2P integration the quality streams will be encapsulated into a single torrent. The torrent will be therefore holding three different copies of the same video file encoded into different qualities.

At first sight it was clear that the only way to go was to exploit the possibility of assigning different priorities to different files of a torrent to be able to download only the interesting part by not loosing bandwidth downloading unnecessary files . As we are going to see later this feature is the heart of the download controller that deals with downloading the right chunks in the right order. The download controller has been analysed and implemented during the second half of the project.

3.1 Switching enabling mechanism

The problem of how to change the quality was the first to be addressed. Because of its big importance in the project, it has been considered from various points of view.

Let us analyse what advantages and disadvantages a codec agnostic solution will bring. To create the switching points depending on I-frames or on particular points in the stream also means to bind the architecture to the type of coding, by restricting future project developments or the modularity of the architecture. As seen in (ref section before), this solution greatly restricts the used encoding parameters. In addition a constant bit-rate should be applied or the three buffers should be analysed at run-time for synchronization, which would cost too much in terms of computational consumption.

To avoid restrictions on encoding parameters, the video player should be able to detect a change of quality in the stream (in other words a quality switch) and re-initialize the audio and the video decoders with the correct values. With the used multimedia player, VLC, we

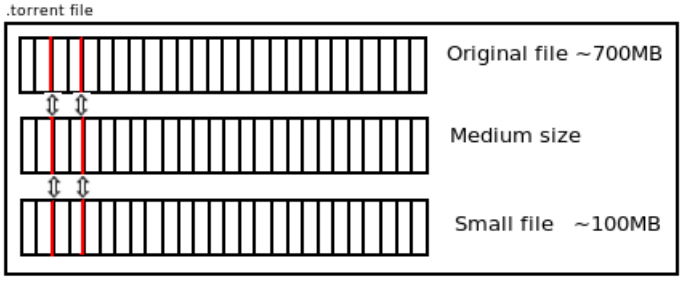


Figure 3.1: Switching points along the quality streams of an encoded MBR .torrent file

will see in the following sections how also the video and audio outputs are re-initialized with the right codec informations such as the frame size. This is usefull considering that different qualities can be encoded with different resolutions.

The final solution is a *time based* alignment, see figure 3.1. This approach will locate the switching points on a time scale. The original video will be encoded into many small files holding few seconds of the video/audio streams. Every file, called *Stripe*, is encoded in a different way, storing the codec headers needed by the player's decoder at the beginning. This approach makes every file able to be played independently. During our experiments we encoded a new stripe for every three seconds, offering the possibility of switching buffer every stripe, therefore every three seconds. Creating the switching points based on time was found to be the successful choice to have a codec independent approach. For our design we decided to use a particular codec and container, but a time based alignment between buffers could be used with every codec as long as the player allows it.

3.1.1 Multi Bit-Rate torrent

In this section we will explain how the torrent has to be created to accomplish the design. First we are going to explain how *Stripes* are saved in the torrent file. In section 3.1.1.2 the conditions that brought us to such a configuration, like the stripes size, are explained and the motivations are given.

3.1.1.1 Stripes

The original video is divided into a number of pieces, called *Stripes*, dependently on its duration:

$$\text{number of stripes} = \frac{\text{duration of the video file}}{\text{duration of each stripe}}$$

The name of every created stripe consists of a character who specifies the quality, in our case "l" for low quality, 'm' for medium quality and 'h' for high quality, and a value that will increase for each stripe. To specify that it is a torrent containing a video for this particular architecture, a text file called *multibitrate.info* that contains additional information is

also written in the torrent. Table 3.1 gives an overview of the files kept by the torrent created from a 21 seconds video file, with switching points every three seconds. This will create a torrent suitable only for use with Tribler, at a later time the user may decide whether to regain the video with the original quality by sequentially combining the stripes of the highest quality or not.

low quality	medium quality	high quality	other files
l0.ogg	m0.ogg	h0.ogg	multibitrate.info
l1.ogg	m1.ogg	h1.ogg	
l2.ogg	m2.ogg	h2.ogg	
l3.ogg	m3.ogg	h3.ogg	
l4.ogg	m4.ogg	h4.ogg	
l5.ogg	m5.ogg	h5.ogg	
l6.ogg	m6.ogg	h6.ogg	

Table 3.1: .torrent file containing multiple files

Every stripe will have a different size, variable bit-rate encoding, and naturally the lower the quality is the smaller the stripe is. The size difference between stripes is the fundamental condition to be able to download at different connection speeds. Table 3.2 gives an idea of what could be the size for the different qualities. In particular those are the average sizes of the final stripes created with our encoding algorithm 4.1.1 (see later).

Quality	low	medium	high
Size per Stripe	$\simeq 91kB$	$\simeq 202kB$	$\simeq 425kB$

Table 3.2: stripe sizes with configuration later discussed in section 4.1.1

3.1.1.2 Exploring Stripe size dependency

The values of table 3.2 have been chosen depending on the upload speed of ADSL connections (Asymmetric Digital Subscriber Line). As we discussed in the previous chapter, in a P2P system a peer could only download as much as it uploads. For our studies we took into consideration a connection with 256kbit/s as lower bandwidth bound and as upper bound a connection with 1,4Mbit/s as upload speed. The upper bound can be increased depending on the quality of the original video file, while we found that our lower bound, 256kbit/s as upload rate, is a reasonable value considering the current available ADSL connections.

Before going further explaining why we chose such values, let's first introduce a concept of which we will use extensively in the next chapters. While designing the architecture of the multi bit-rate Video on demand system, we had to always consider the possibility

of loosing bandwidth, due to a loss of connection or a peer connection closed in advance. Therefore a safe quality fall-back scenario has to be considered and, while downloading a higher quality, the lower quality will always be downloaded in background (chapter 3.3.3). In the following tables we will show how our stripes have been encoded to gain the best results for the most typical ADSL connections.

In table 3.3 the minimum speed requirements for every quality are shown. The low and medium quality values are typical upload limits for commercial ADSL connections.

Quality	low	medium	high
Minimum ADSL upload limit	$\geq 256 kbit/s$	$\geq 798 kbit/s$	$\geq 1,4 Mbit/s$

Table 3.3: Minimum ADSL upload speed limit for a continuous playback

Table 3.4 shows the reasons for choosing the stripe sizes. As we can notice from the table, the minimum required speed for continuous playback of one quality in a safe fall-back scenario is upper bounded by the upload limits of commercial ADSL.

- Low quality: $30,3 kByte/s < 31,25 kByte/s$
- Medium quality: $97,8 kByte/s < 98 kByte/s$
- High quality: $171,3 kByte/s < 175 kByte/s$

3.2 Solution for quality switching

In this section we are going to analyse what have been the crucial design decisions that brought us to gain a flicker free playback. The main addressed issue was to avoid the few milliseconds latency between stripes. This was made possible thanks to the correct choice of codecs and containers, allowing a simple management of the audio and video streams.

Further than choosing codecs and containers a small modification had to be done to the default Tribler multimedia player, VLC. This modification affects only a library used by the player, leaving the structure and behaviour of the same unchanged. In section 3.2.2 the modifications of the ogg demuxer library of VLC are explained in a detailed way.

3.2.1 Codecs & containers comparison

After the analysis of section 2.3 we will now describe the reasons that have caused us to use the ogg container. First of all:

the ogg container is a free software [25], therefore patent-free.

Quality	low	medium	high
Size per 3 second Stripe	$\simeq 91kB$	$\simeq 202kB$	$\simeq 425kB$
Minimum required speed for continuous playback of only one quality	$30,3kByte/s$	$67,5kByte/s$	$141KByte/s$
Minimum required speed for continuous playback of one quality in a safe fall-back scenario	low quality $\Rightarrow 30,3kB/s$ $30,3kByte/s$	low + medium $\Rightarrow 30,3kB/s + 67,5kB/s$ $\simeq 97,8kByte/s$	low + high $\Rightarrow 30,3kB/s + 141KB/s$ $\simeq 171,3kByte/s$
Bandwidth limit (kbit/s)	$\geq 256kbit/s$	$\geq 798kbit/s$	$\geq 1,4Mbit/s$
Bandwidth limit (kByte/s)	$31,25kByte/s$	$98kByte/s$	$175kByte/s$

Table 3.4: Stripe size dependency

Ogg specifications are publicly available. The libraries of reference for encoding and decoding are released under BSD license[1] . The official instruments to manage the containers are licensed under the GNU General Public License (GPL)[2].

The main reason that prompted us to OGG rather than Matroska [3] was one of its design characteristics:

> The internal structure of an Ogg file allows the binary concatenation of streams.

The resulting Ogg file perfectly complies with the specifications. The same specifications provide the so-called *chained streams*. As we know Ogg is just a format that specifies how the data should be sorted in the data stream. The audio or video encoded by a specific codec will be included in the Ogg container [37]. The container can contain streams encoded differently, for example, an audio/video file containing two streams encoded with different codecs.

As a form of containment, Ogg can integrate third-party codecs (like DivX, Dirac, XviD, MP3, etc.) but is usually used with the following codecs:

[1] further reference: `http://en.wikipedia.org/wiki/BSD_licenses`

[2] further reference: `http://en.wikipedia.org/wiki/GNU_General_Public_License`

[3] The Ogg container is not the only free software, also the Matroska container offers a good alternative. Matroska is an open standard project developed by the Matroska Development Team and licensed under GNU LGPL[4] designed as a compromise between the strong-copyleft GPL and permissive licenses such as the BSD license.

- Audio Codec
  - Lossy

    Speex : handles the compression of the human voice to low bit rate (8-32 kbit / s / channel)

    Vorbis : Manages moderate compression of audio generic (16-256 kbit / s / channel)
  - Lossless

    FLAC : Manages audio generic preserving all the information of the original signal

- Text Codecs

  Write : codec for the management of the text in subtitled movies

- Video Codec

  Tarkin : experimental codec that uses wavelet transforms 3D. Development is currently suspended, as the focal point of development is currently Theora

  Theora : video compression codec based on VP3 of On2

Of these, only FLAC is also commonly used without the ogg container. Out of this list of codecs we decided to use Vorbis for the audio stream and the Theora for the video stream. Both codecs are free and open-source, developed directly by the Xiph.org foundation [5]. We chose the Theora codec because it scales from postage stamp to HD resolution, and is considered particularly competitive at low bitrates. It is in the same class as MPEG-4/DiVX, and like the Vorbis audio codec it has lots of room for improvement as encoder technology develops. The Vorbis codec has been designed to completely replace all proprietary, patented audio formats [32]. Vorbis is an open source algorithm for lossy compressing digital audio-type, direct antagonist of other standards such as MP3, VQF, AAC. With the same quality, allows greater compression than the MP3 format, thanks to advanced psychoacoustic research. Among the qualities that are commonly attributed to Vorbis, referring especially to the inevitable comparison with the de facto standard MP3, we consider a greater extension and a cleaner sound at high frequencies (above 16 kHz), native support for multi-channel and a general better preservation of the spatiality of the sound of the original signal . One of the only defects, compared to other codecs like MP3, is the heaviness of the algorithm while decoding.

3.2.2 VLC media player

VLC media player [46] is a highly portable multimedia player, capable of supporting various audio and video formats (MPEG-1, MPEG-2, MPEG-4, DivX, mp3, ogg, ...), as well as DVDs, Video CDs and various streaming protocols. VideoLAN is a free open source

[5] reference: `http://xiph.org`

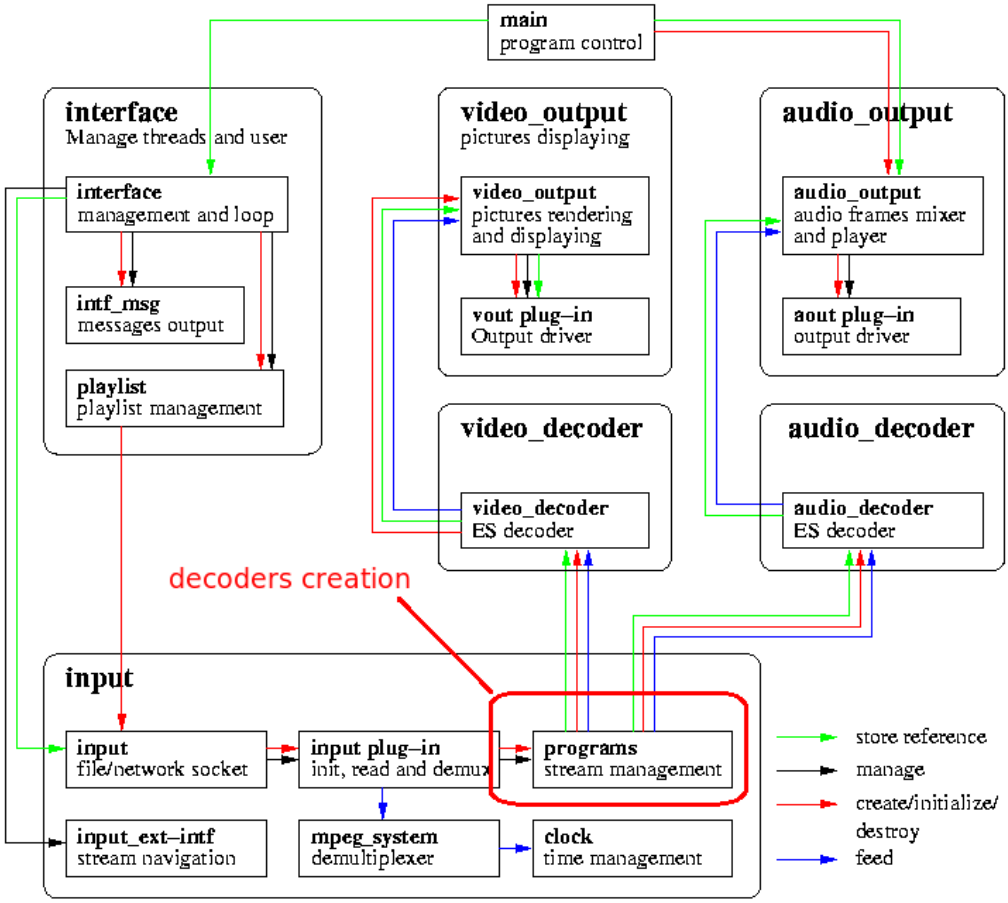


Figure 3.2: The VLC architecture

project released under the GNU General Public License and therefore can be modified and distributed within the Tribler software without any patent problems. The core of VLC media player, LibVLC, is built in a modular way, allowing an easy modification and integration of every feature of the program.

The core gives a framework to do the media processing, from input (files, network streams) to output (audio and/or video, on a screen or streamed on the network), going through various muxers, demuxers, decoders and filters. Even the interfaces are plugins for LibVLC. It is up to the developer to choose which module will be loaded. VLC is heavily multi-threaded and, as we can see from figure 3.2, taken from [46], is a layered system where layers are created in a hierarchical way.

The problem concerning the media player is the audio and video decoder threads recreation for every stripe encountered while reading an Ogg chained-stream. In figure 3.2 the thread responsible of the decoders creation is highlighted. The programs responsible of

the stream management are created and managed by the chosen demuxer. The demuxer is responsible of killing a thread when and ”end of stream” or ”end of page” occurs. The time needed by the audio and video decoders to decode the first bytes of a stream and send them to the audio and video outputs is what actually causes the $20 - 60ms$ latency between stripes. We will address a solution to this issue in the next section.

3.2.3 Alternatives for solving flickering problems

As we sad in the previous section the ogg demuxer is responsible for the decoders creation. First we will give an overview of what was the initial design to allow a flicker free playback, and then we will present our proposed solution.

3.2.3.1 Multiple decoders approach

The initial idea was to create multiple instances of the same decoder working in parallel and serving the same audio and video outputs. Figure 3.3 explains the concept in an easy way. The two decoder instances will alternately handle incoming Stripes serving the same outputs at the right time. In figure 3.4 the modifications that would have been done to achive this goal are shown. The demuxers modules would have to be modified to initialize two instances of the same decoder for every stream. Those instances will have the same reference of video and audio outputs.

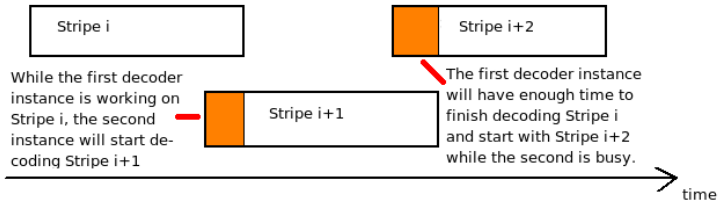
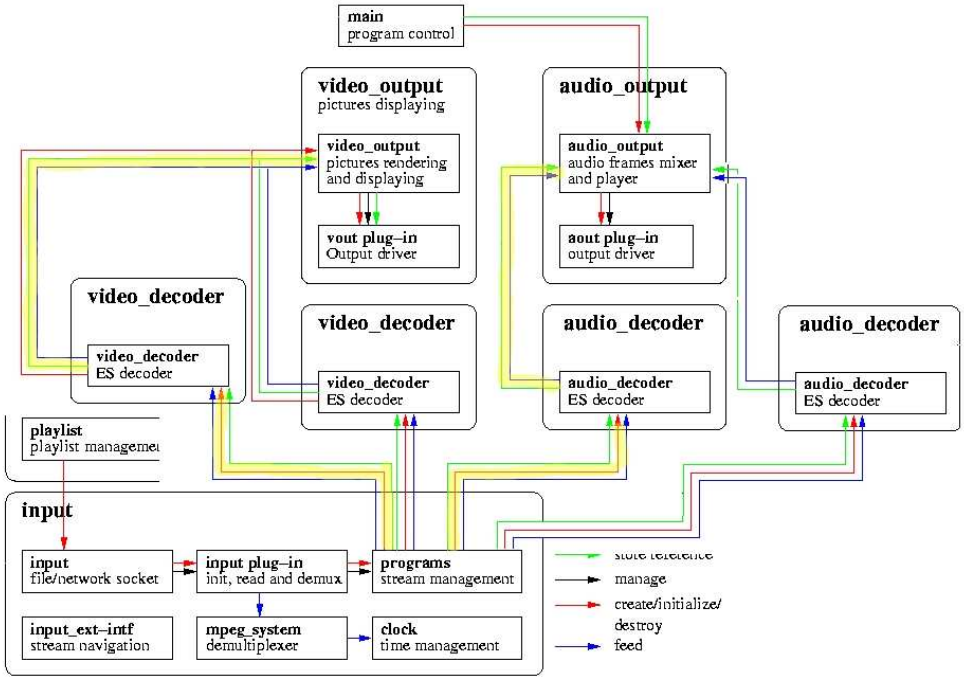


Figure 3.3: Codec header pre-reading

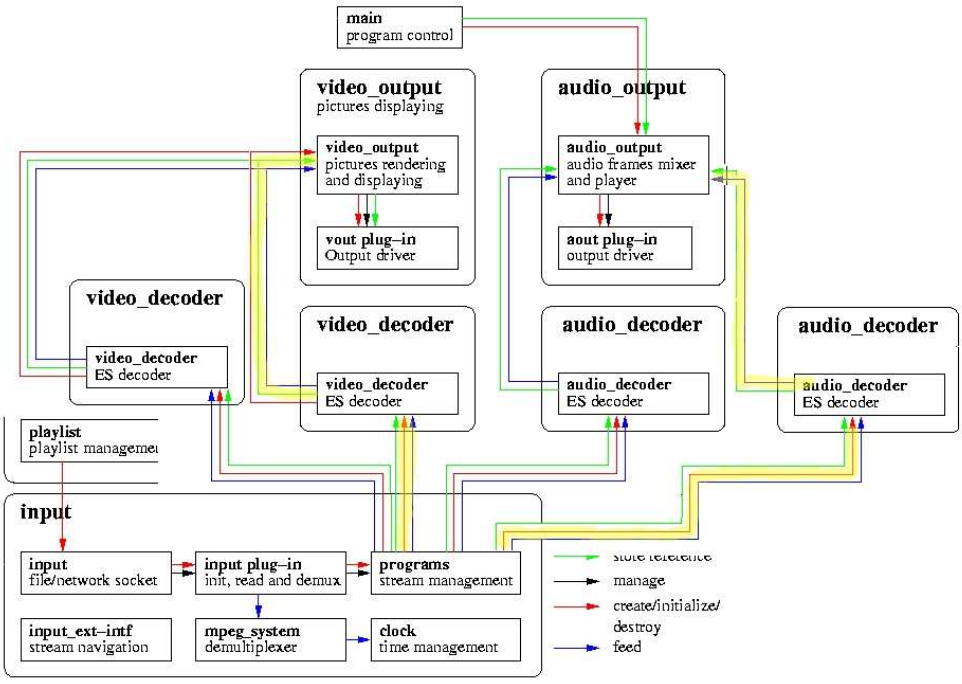
This solution would have brought multiple advantages like a codec independent architecture. On the other hand the many disadvantages that derive from this solution are discouraging such an approach:

- It would be a big modification to the behaviour of the application, moving away from the original architecture.
- All the used demuxers would have to be modified wisely.
- It would bring extra computational requirements for every video opened by VLC media player.

Therefore our final solution to gain a flicker free playback will only concern the ogg demuxer as we will see in the next section.



Control flow for Stripe i



Control flow for Stripe i+1

Figure 3.4: VLC control flow for a multiple decoders solution, chapter 3.2.3.1

3.2.3.2 The ogg demuxer approach

As we already explained in section 3.2.1 one of the main reasons that brought us to choose the Ogg container as our encoding target is his chained-streams functionality. When concatenating [6] Ogg streams, most of the players handle a chained stream as a single stream providing by default a flicker free playback. This was not the case for the VLC media player where the chained streams where handled like a sequence of different files. The only difference from the demuxer side was that once a new header is found the old decoder threads are terminated and new threads for the new audio and video streams are created. As explained in the previous section this brings the undesired playback block for few milliseconds.

To prevent the video from stalling, the Ogg demuxer has been modified to avoid the thread recreation while decoding a chained-stream of the same quality. As we can see from figure 3.2 the demuxer is responsible of creating, and therefore terminating, the audio and video decoders. Our modification to the ogg module concerns the *demuxing* process where a page is read and, if no stream is found, a signal is sent to the stream manager to terminate the decoders. Where on one side this functionality could be useful considering chained ogg streams encoded with different codecs, on the other side there is the need of a controller to check whether to spend time re-initializing the decoders or not. Our modification makes the input module responsible for terminating the demuxer at the right time, and therefore the decoders, once the *end of file* is reached.

Let us now consider our situation in a more detailed way. Our three buffers, holding the three different qualities, will be seen from VLC as three items in its playlist. The buffers will be filled up at run-time while downloading the video, once there is the need of a quality switch will we start filling the buffer referenced by the next item in the playlist. After few seconds the player will reach the end of the file for the current item and switch to the next item in the playlist, the next quality. This is the only time when re-initializing the input also terminates the decoders and initializes them again with the right values of the new quality. The proposed solution ensures that a ”hang” in the video will occur only with a playlist item or quality change, when there is the real need for decoders re-initialization. The stripe headers will still be processed by the ogg demuxer, but they will not be sent to the decoders for elaboration. The codec headers, interpreted by the demuxer as *wrong* data will be handled as garbage and skipped by the audio and video outputs.

The same modification has been applied to many different versions of VLC media player without encoring in any restyling of the ogg module. The versions that have been tested (with the modified module) go from the 0.8.6a, the oldest version used by Tribler in February 2008 [7], until the latest development version 0.9.3 that is planed to be used in future versions of the Tribler project. For all the tested versions [8], the ogg module’s behaviour and implementation did not change, except for few minor code enrichments, therefore the same

[6] to arrange into chained list

[7] The beginning of my project

[8] VLC media player versions: 0.8.6a, 0.8.6b, 0.8.6d, 0.8.6e, 0.8.6f, 0.8.6g, 0.8.6i, 0.9.0, 0.9.1, 0.9.3

patch [9] has been applied successfully.

3.3 BitTorrent enhancement

In this section we will analyse the design decisions and the algorithm that allows a multi bit-rate implementation. First we will see some details of how a generic torrent is handled by BitTorrent(ref), after we will explain our crucial algorithm that integrates with BitTorrent.

3.3.1 Downloading a torrent

One of the main reasons that brought the torrent protocol to be a widely used is its capacity of downloading big files in a fast way. This is given by the fact that the torrent protocol does not allow people to swap complete items, such as music tracks or entire TV series, but it breaks each piece of information into tiny fragments, called *chunks*. At the time of the torrent creation it is possible to determine a particular chunk size [10]. Every downloaded chuck will be controlled and verified by the Tribler core once downloaded.

The chunk boundaries are independent from the stripe boundaries. As we saw earlier a stripe has a variable size. The stripe size will affect more peers with a low speed connection, who are able to watch only the low quality stream.

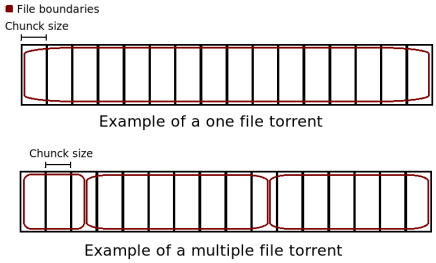


Figure 3.5: Chunk and file boundaries comparison

Considering that chunk boundaries do not coincide with stripe boundaries, we need to have stripe ordered in the right way into the torrent to optimize the download efficiency. In figure 3.6 we see how the download efficiency is related to the order of the stripe in a torrent file. If the stripes are not ordered, we will end up downloading unneeded chunks. While this is not a problem considering high quality stripes, this is a big issue with the low quality stripes. Experiments at chapter 5.1.3 have shown that when downloading the low quality stream, the size of the chunks is an important factor. Depending on the size of the chunks and on the size of the stripes, it is often found that one chunk can hold multiple stripes with a small amount of information (the credits of a movie is an example).

[9] A patch is a small piece of software designed to update or fix problems with a computer program or its supporting data.

[10] The size of a chunk can only be a power of 2, normally $\pm 2^{16}$

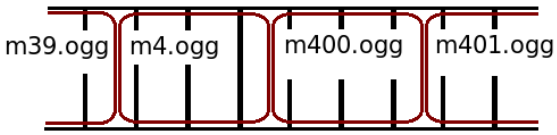


Figure 3.6: Normal order of enumerated files in a torrent

Figure 3.7: Final order needed for download efficiency

Therefore the order of the stripe in the torrent, see figure 3.7, is an important requirement for the design of the application. This issue has been solved in an easy way. Rather than taking care of how the files are referenced by the torrent, our encoding algorithm 4.1.1 will start naming stripes from a value that depends on the duration of the original video file.

$$\text{Initial value for Stripe naming} = \text{Number of stripes} + X,$$

where ”X” could be any value but for simplicity in our encoding algorithm it is rounded up to the closest power of 10 value.

3.3.2 Priority assignment

Every chunk can be downloaded with a different priority. Torrents are normally downloaded with a rarest first priority. This is implemented to offer an equal availability of the chunks over the Internet.

For every chuck four different priority values can be assigned:

- Low priority $\rightarrow L$
- Medium priority $\rightarrow M$
- High priority $\rightarrow H$
- Never download $\rightarrow redline$

This implementation will not download the torrent in an ordered way but will try to download the entire torrent as soon as possible. A different download implementation is already provided by the *Give-to-get* [34] algorithm.

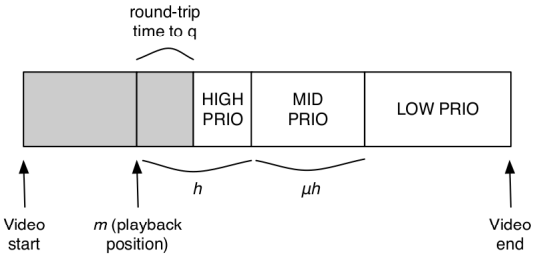


Figure 3.8: Give-to-get priority sets assignment

Figure 3.8 shows how the priorities are assigned depending on the playing position into the stream. Pieces that have an early deadline are taken first, therefore a high priority is assigned to them. Piece that are going to be played in the future will have a lower priority depending on the distance with the playback position. Except for the highest priority set, where chunks are downloaded in order, for the medium and low sets the Piece Picker[11] will download with a rarest first policy.

3.3.3 Dynamic priority assignment policy

Our algorithm will act more or less like the Give-to-get algorithm. The main difference is that we will have to switch between three buffers/streams and therefore we will have to take in consideration the priority assignments when switching buffer for changing the quality. Our algorithm will not directly manage the chunk priorities, like the Video on demand does, but it will set the priorities on the file. This is an indirect way of managing chunks priorities. The *file selector* [12] will than be responsible of setting the right priority to the right chunk.

Below is a clear explanation on how the priorities for the different qualities are set.

- Low qualities will be downloaded in a similar way to the give-to-get algorithm. The medium and the high quality will not be downloaded, figure 3.9.
- For medium qualities a safe fall-back scenario has to be taken into consideration. While downloading the medium quality also the low quality will be downloaded to provide this functionality, figure 3.10.
- The high quality download is a similar scenario to the medium quality. Therefore only the medium quality will not be downloaded, figure 3.11
- A different priority schema is applied when the algorithm assumes that there is enough bandwidth to increase quality. This means that we have enough downloaded buffer to

[11] The piece picker is responsible of downloading the chunks of a torrent
[12] The file selector allows to set a particular priority, fig 3.3.2, for every file in a torrent

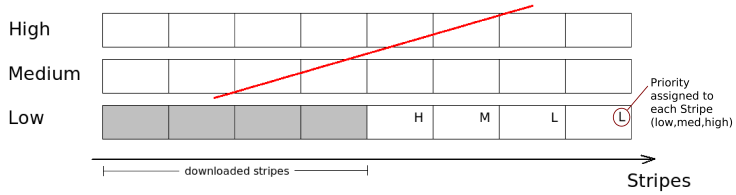


Figure 3.9: Low quality priority assignment

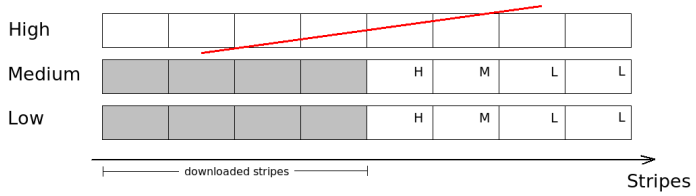


Figure 3.10: Medium quality priority assignment

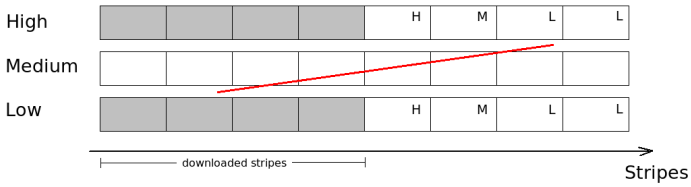


Figure 3.11: High quality priority assignment

safely start downloading the higher quality, figure 3.12. More information on when this happens and how the algorithm decides to apply this schema is explained in the further section 3.3.4.

- Once the end of the stream is reached and we stopped playing the video file, all the file priorities will be set to ”normal” or medium priority. The current implementation will then download them with a rares first policy, figure 3.13.

- The only other set of priorities will be set before starting the playback. At this time the algorithm as no idea of the available bandwidth. Therefore a particular optimistic set of priorities is applied, figure 3.14, taking in consideration the possibility of increasing the quality as soon as possible. For high speed connections we want to increase the quality every stripe, reaching the highest quality in X seconds. Where X is given

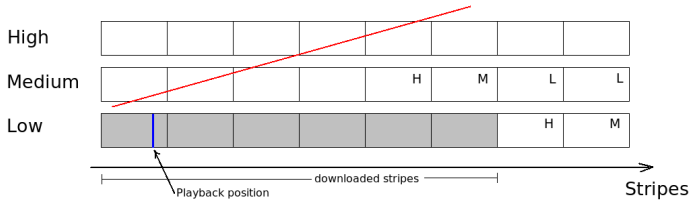


Figure 3.12: Quality improvement priority assignment

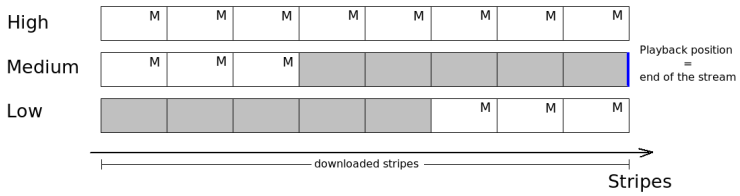


Figure 3.13: Final priority assignment

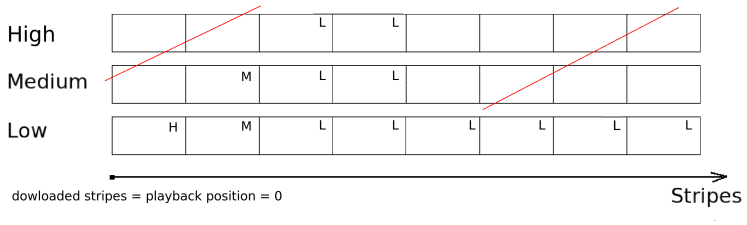


Figure 3.14: Initial priority assignment

by:

$X(\text{seconds to start playing the higher quality}) = \times\text{duration of a stripe}(\text{number of qualities} - 1$

For an easy understanding from now on we will talk about file priority and not anymore about chunk priorities.

3.3.4 The Multi Bit-Rate Algorithm

In this section our Multi Bit-rate algorithm will be presented. This will only define the design decisions for the algorithm. In chapter **??** we will analyse the algorithm more in depth. We will now analyse the different ”status” of the algorithm. Every status will determine a priority change as we saw in the previous chapter. With the current video on demand implementation there is only one major status.

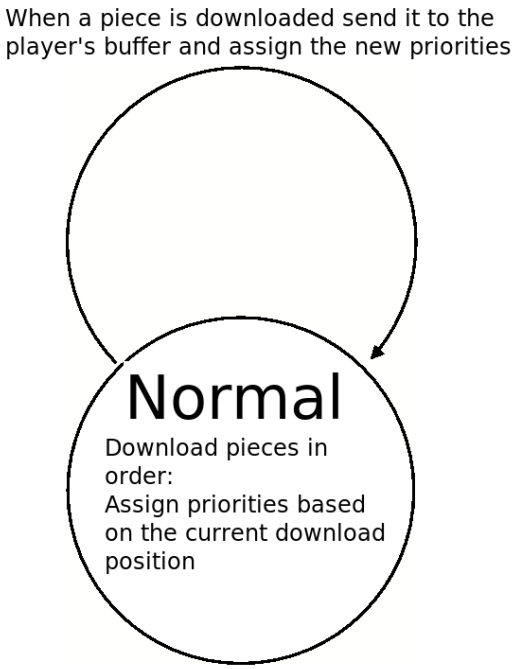


Figure 3.15: State diagram for the current VoD implementation

During the normal status the stream pieces are sequentially downloaded. Once a chuck has been downloaded it will be sent to the player and new priorities will be assigned.

3.3.4.1 State analysis

The MBR algorithm has different status that will determine witch sets of priorities will be assigned to the files.
We will now consider only the case of a *stable environment*, no bandwidth loose and no peer dropping of the system, in other words we do not consider a safe fall-back scenario. The algorithm does not send the stripe as soon as it is downloaded, but it waits until the player's buffer is too small and we have to take a decision otherwise the playback will stall. The only time a stripe is sent to the player's buffer, without having a short deadline, is when we actually increase the quality by switching buffer. To be more precise, the stripe sent to the player will not be added to the current buffer but will initialize a new buffer, that will then be added to the player's playlist.

In figure 3.16 "S" represents the reference to the current stripe that can be increased sequentially, i+1, or by quality, q+1 [13]. "X" & "Y" are values of the algorithm that can be

[13] example: i++$\Rightarrow$ *m*123.*ogg* $\rightarrow$ *m*124.*ogg*;q++$\Rightarrow$ *m*123.*ogg* $\rightarrow$ *h*123.*ogg*

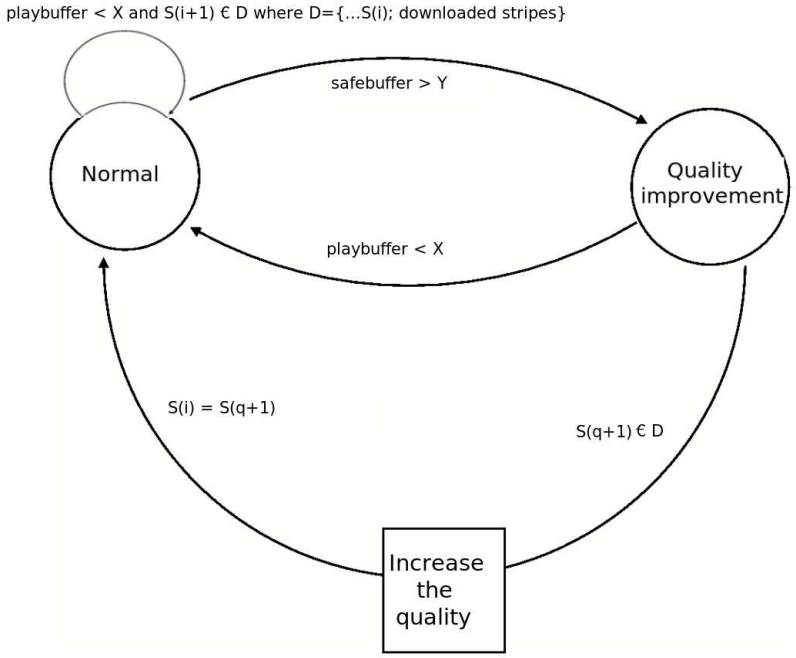


Figure 3.16: State diagram for the MBR implementation in a stable environment

modified to change its behaviour. ”D” is the set of already downloaded stripes.
As we see in figure 3.16 we switch between two status.

- In the *normal* state the algorithm will download stripes sequentially, as soon as a stripe is requested, because the player's buffer is to small, the algorithm will send it.
- The algorithm will switch to the *Quality improvement* state once there is enough safe buffer [14]. The algorithm will than switch back to the *Normal* state if the player's buffer is too small or it will update the current stripe to the relative stripe of the higher quality buffer, if a higher quality stripe has been downloaded.

Figure 3.17 shows the algorithm behaviour in a dynamic environment. For simplicity the graph does not show the previous discussed scenario, assuming the *Quality improvement* state previously explained. For a complete picture of the status diagram please refere to figure 3.18.
In this environment a new state appears.

- The algorithm will switch to the *safe fall-back* state once the player's buffer is too low and the current referenced stripe is still not downloaded. In this case we have to take a decision as soon as possible to serve the player with a new stripe. The algorithm

[14]The safe buffer is defined as the amount of downloaded stripes that still have not been sent to the player

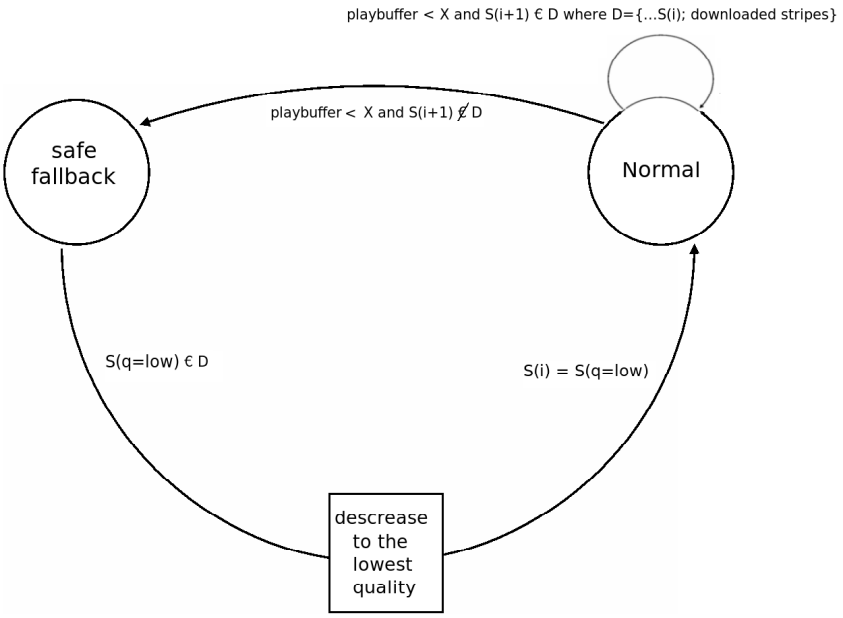


Figure 3.17: State diagram for the MBR implementation in a safe fallback scenario

checks if the stripe of lower quality relative to the current one has been downloaded. If it has been downloaded, than the algorithm will send it to the player's buffer and update the current stripe with an instance of it.

3.3.4.2 Signal analysis

Figure 3.18 shows how the two scenarios are interrelated. In this diagram we explain when the algorithm tells the player to switch to a new buffer, quality improvement or low quality fall-back *transaction*, but we assume the player is already reproducing video. We will now analyse when the algorithm will tell the player to start, pause or stop the playback.

Start The algorithm will tell the player to start the playback once the player's buffer has been filled with a certain amount of video. For our experiments we decided to start the playback once the first 6 seconds of video have been downloaded.

Pause We will pause the video once it is not possible to switch from a *safe fall-back* to a *normal* status. In other words once the a safe fall-back is not possible, when the relative stripe of lowest quality is not available. When this happens we reinitialize the playback position and we have to wait until the player's buffer is filled with the amount of data specified for the *start* condition.

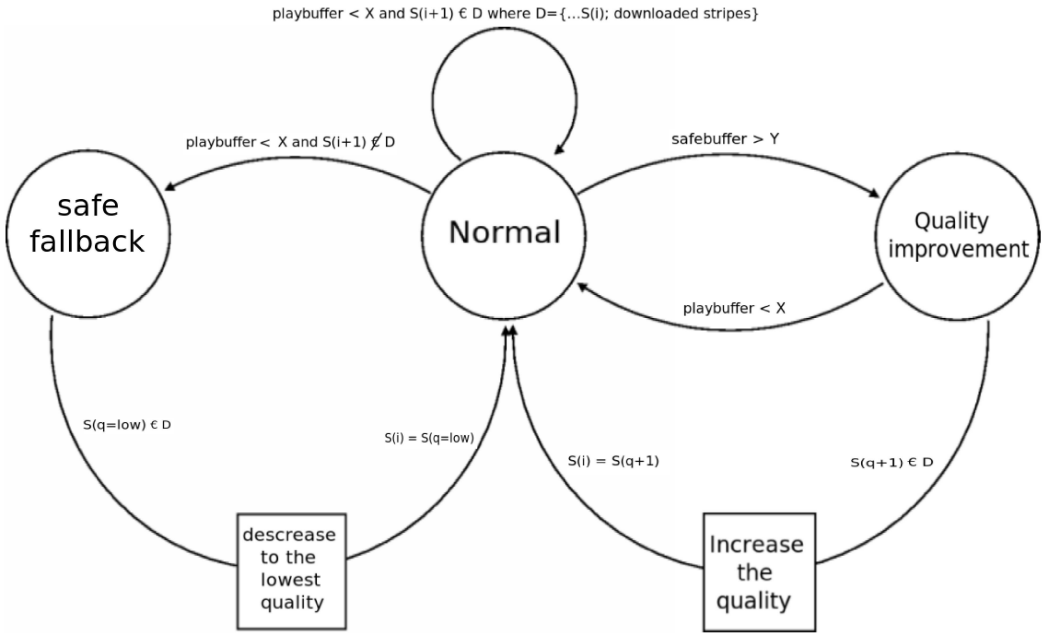


Figure 3.18: Final state diagram for the MBR implementation

Stop The playback will be stopped only when we reach the end of the stream. When this happens all the remaining stripes will be downloaded with a normal priority, with a rarest first policy.

In this final section we saw the state diagramm for our MBR implementation. We will use the final state diagram of figure 3.18, modifying it to figure 4.3, to explain the algorithm's implementation, section 4.2.2.

Chapter 4

Implementation

In this chapter we are going to present at a high level the implementation of our novel MBR design. The chapter is divided into two sections. The first section will talk about the ”side effects” of the proposed solution. Section 4.1.1 will explain how the encoding algorithm works and what is the quality difference between quality stripes. Section 4.1.2 will briefly explain how VLC’s ogg demuxer module has been modified to gain a flicker free playback. Section 4.2 will describe in a deeper way the behaviour of our novel algorithm proposed in chapter 3.3.4.

4.1 Encoding and Playing

As side effect of our design decisions we had to define a new encoding methodology, subsection 4.1.1, and a new flicker free playback solution, see subsection 4.1.2.

4.1.1 Encoding methodology

As previously discussed in section 3.1.1.1 we have some constrains about the stripe [1] sizes. Those constrains will put some limitations on the encoding procedure. Figure 4.1 gives a graphical overview on how the encoding algorithm creates a new torrent aimed to provide the three quality streams encoded into stripes.

The original video file is parsed by the encoder algorithm. For every three seconds three different quality Stripes are created: high, medium and low. We considered multiple tools to encode in a more efficient way the video file. After a short investigation we decided to use ”FFmpeg” as major encoding program.

FFmpeg [17] is a complete software suite to record, convert and play audio and video files. It is based on libavcodec library for encoding audio/video streams. FFmpeg is developed on Linux, but can be compiled and run on any major operating systems including Microsoft Windows, therefore quite useful considering the platform independent approach of Tribler. FFmpeg is also used by the current Video on Demand functionality to detect the

[1] A three second independently encoded video file

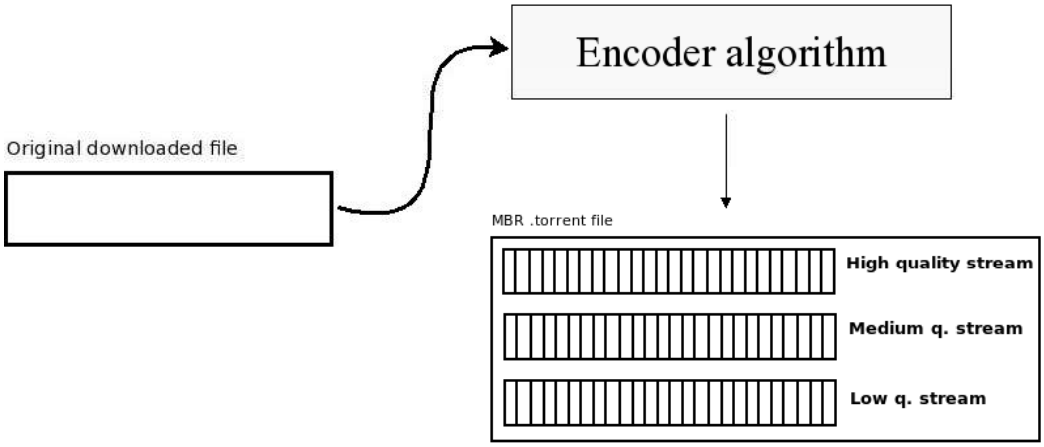


Figure 4.1: Encoding methodology

bit-rate of a video stream if it is not specified in the torrent metadata [2]. The theora library used by the standard ffmpeg installation is still a little bit ”buggy” and it is still not possible to fully appreciate the power of the theora codec. A better implementation is offered by the program ”FFmpeg2theora” [15] that allows to encode a video stream using the theora codec specifying a quality level for the frame encoding. The reasons for not using FFmpeg2theora to encode our Stripes is that the resulting ogg files can not be concatenated into a chained-stream. We need the chained-stream functionality offered by the ogg container format for a fluent audio/video playback, section 3.2.1. Therefore we will use ffmpeg to encode our Stripes, considering to modify the encoding parameters once the theora library, included in ffmpeg, is able to manage frame quality settings.

We are not trying to get an average bit-rate for the quality streams. We noticed that specifying the audio or video bit-rates did not have a consequence on the final Stripe size, considering the theora and vorbis codecs. All our concernes go to the Stripe size, section 3.1.1.1. On the other hand parameters such as the frame size and frame rate for the theora codec, brought us to the searched proportionality between Stripe qualities. Regarding the audio stream we noticed that encoding with Vorbis, the sampling rate and the quality adjustment had a bigger influence on the final size rather than modifying the bit-rate [3].

Table 4.1 shows the parameters used to encode the quality streams used during our ex-

[2] The .torrent metadata holds information about the location of trackers, file sizes and file hashed for detecting corrupted files. For the current Video on Demand functionality an additional field, for storing the bit-rate value of the video file, has been added to the metadata. This additional field is not needed for the multi bit-rate architecture, the algorithm does not relay on the bit-rate assuming that mostly the minimum available downloading bit-rate is higher or equal to the low quality stream bit-rate.

[3] probably because of the short duration of each Stripe

encoding parameters	**frame rate**	**frame size**	**audio sampling rate**	**audio quality [0-10]**
High quality Stripe	24	1024x576	48000	10
Medium quality Stripe $\simeq 3/4 \times$ high quality	18	768x432	24000	8
Low quality Stripe $\simeq 1/2 \times$ high quality	12	512x288	24000	5

Table 4.1: Encoding parameters used during the experiments

periments of chapter 5. Those have shown to be valid values from the final user experience. Even the percieved quality when watching the smallest stream is comparable to the quality experienced watching YouTube videos. Those values has been chosen depending on the target Stripe sizes of table 3.2 motivated in section 3.1.1.2.

The encoding parameters proportionality between quality Stripes clarifies the proportionality of the final stripe sizes. Considering only the video stream and the proportion between a low quality Stripe and a high quality one, we see how we halved the frame rate and the frame size. Therefore the final video stream size will be:

$$\text{low q. video stream} = \frac{1}{2}(\text{high q. frame rate}) \times \frac{1}{2}(\text{high q. frame size})$$

$$\Rightarrow \text{low q. video stream size} = \frac{1}{4}(\text{high q. video stream size})$$

The final low quality Stripe size will be 13-15% smaller that the quarter of the high quality encoded Stripe, given to the different encoding of the audio stream.

4.1.2 Flicker free playback solution

In this section we are going to discuss in a superficial way the modifications applied to gain a flicker-free playback. It is not of scientific interest to take a close look at the actual modification that has been applied to the VLC's Ogg demuxer module.

The major modification has been applied to the *Demux* function of the module. This function is responsible of reading and demuxing data packets received from the input module. This function runs in an infinite loop until an error or an *end of stream* occures. Once an *end of stream* occurs, in our case at the end of every Stripe, a signal to terminate the decoders is sent to the appropriate module. Our modification consists of keeping track of the id, or reference, of the audio and video streams. Once we reach the end of a Stripe, if the id of the demuxer is still the same [4] and the the audio and video stream of the next Stripe

[4] In case of a chained-stream, the Ogg demuxer will not be terminated but just re-initialized, holding the same id or reference

are encoded in the same way as the current one, we skip the demuxer re-initialization. This is done by sending the header packet to the decoders. The Decoders will detect the codec header and handle it as *garbage* information, skipping to the next data package that would be the first data package of the following Stripe.

This architecture has some limitations considering that not always the *Demux* function can detect correctly the typology of Streams and the quality change between Stripes. It was not into the scope of this document to provide a detail investigation about codec and container formats, but moreover to prove that a variable bit-rate Video on Demand is possible to realize using the existing BitTorrent protocol.

As last consideration regarding the quality Stream switch: When switching between qualities, in our case items of VLC's playlist, all the threads managed by the playlist will be re-initialized, figure 3.2. During this operation the decoders re-creation is the most time-consuming step. This can not be avoided considering that many parameters, such as the frame size, differ between qualities, ending up creating decoders and outputs in a different way [5].

4.2 Multiple Bit-Rate algorithm

In this section we are going to discuss the enhancements that have been applied to the BitTorrent download policy. Subsection 4.2.1 will give an high level view on the modifications applied to the Tribler Core. Subsection 4.2.2 will give a first introduction to the novel download policy. We will analyse how the sets of priorities presented in section 3.3.3 will be applied based on the "current state", defined by a set of variables managed by the algorithm.

4.2.1 Tribler Enhancement

We are going to briefly describe the crucial changes applied to the core of Tribler. The modularity of Tribler's Core allowed to apply the needed modifications in a transparent way for the current VoD implementations. Figure 4.2 describes the major classes used to manage a VoD streams. As the Tribler's GUI also the Swarmplayer is an external interface used to interact with Tribler's Core. The API of Tribler allow to manage every kind of supported download with few lines of code.

The Swarmplayer is actually nothing more than an interface that sets the right parameters in the *Download Config* class and initializes also the *Download* class in the proper way. The Swarmplayer will locate the video file [6], the bit-rate if specified and other useful informations from the torrent metadata file. This information together with the specific callback

[5] Eg. the video output has to be re-initialized for every new frame size, for every processed quality stream

[6] If more than a video file is detected in the .torrent file, the Swarmplayer will ask the user to select a specific one.

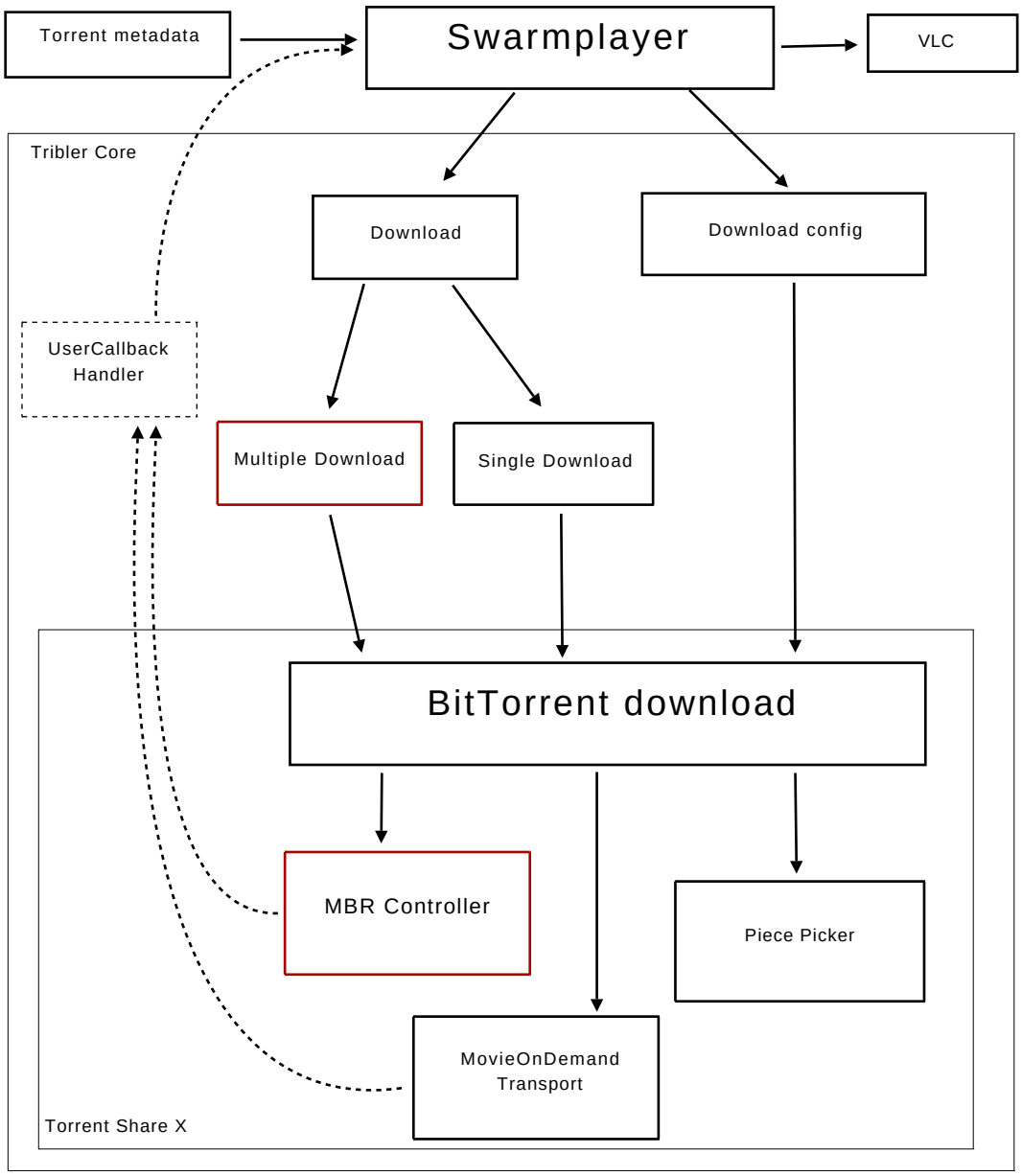


Figure 4.2: Partial architecture for the Video on Demand architecture in Tribler's Core

functions will be saved in the download configuration. Once the download configuration parameters have been set propertly, the Swarmplayer will initialize a *Download* object that indirectly initializes a *Single Download* object. Currently every kind of download, Video on Demand, live streaming or normal BitTorrent download, is handled by Tribler's Core as a *Single Download* object. The *Single Download* together with the *Download Configuration*

will be responsible of initializing with the right configuration and callbacks the *BitTorrent download* that will start the actual download process. It is not intention of this thesis to provide a clear understanding of the classes involved in the downloading process of a torrent file. Therefore in this chapter we will only focalize our attention on the major enhancements provided by the VoD and MBR implementation. For the VoD implementation the *BTdownload* will initialize a particular *Piece Picker* that will download the torrent pieces following the order of the Give2Get algorithm described in section 3.3.2. The other class that is shown in figure 4.2 is the MovieOnDemand Transport that will communicate with the Swarmplayer through the callback functions specified in the download configuration and a specific set of events. The events used by the VoD implementation are:

start once enough pieces to watch the video stream without interruption have been downloaded .

pause if we are running out of buffer and the downloading bit-rate is not the same as previously predicted.

resume follows the pause signal if the same conditions as for the start event occur.

The callback function located in the Swarmplayer will handle the video stream sending the right signal to the video player, VLC, depending on the given event.

The classes highlighted with red have been introduced for the novel MBR implementation. We introduced majorly two classes that will manage the torrent download in a different way than the current implementation. We decided to merge our implementation with the existing Swarmplayer following the same design decisions. In our case the Swarmplayer will detect if the provided .torrent file is a Multi bit-rate encoded file [7] and will initialize differently the *Download* and *Download config* objects.

We are not interested in the bit-rate of a specific video file but in keeping track of the piece range of every Stripe. A different set of information will be taken by the Swarmplayer from the torrent metadata and used to initialize wisely the download configuration. A different callback function is used to handle events, and a new set of events has been introduced. We added a **next** event that will add the next quality stream to player's playlist. Once the player reaches the end of the item it's playing it will automatically switch to the next item in the playlist, the next quality stream. The class responsible of managing events, downloading properly the Stripes and manage the quality streams is the *MBR Controller*. Already introduced in section 2.2.2, the *MBR Controller* will manage the execution of our novel MBR algorithm.

The MBR algorithm is performed and re-scheduled every half a second by the controller. Two time per second our algorithm analyses the download process, the video playback

[7] This is done by parsing the multibitrate.info file and performing a general check for some characteristics of the video files, or Stripes

status and reacts by sending events to the Swarmplayer, applying different priority sets and managing properly the quality streams.

4.2.2 Download policy

Before analysing directly the algorithm we found to be easier to give an introduction with the help of graphical examples. The download policy is handled by the MBR algorithm, a greedy algorithm that sequentially analyses the state of the download. As explained in the previous chapter the algorithm will be executed half a second, therefore also the status of the download will be periodically updated. From the algorithm design of section 3.3.4 we now propose a modified version of figure 3.18. In figure 4.3 we added a number for each state transaction, making easier to relate during the algorithm explanation.

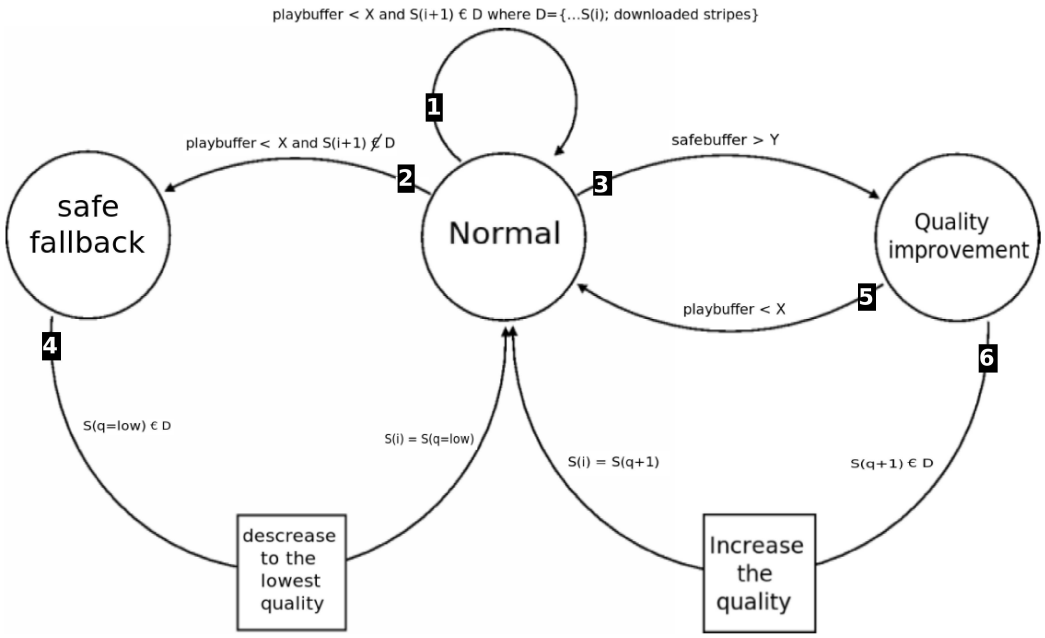


Figure 4.3: State diagram for the MBR algorithm

From figure 4.3 we can see how the algorithm's decisions mostly depend on the status of the playbuffer, transactions *1*, *2* and *5*. The first crossroad of the algorithm actually considers only the status of the playbuffer. The next two subsections will consider the two different paths.
Subsection 4.2.2.1 will discuss the haste choice scenario where we need to take a decision because the player's buffer is too low. On the other hand if we have enough player buffer we will have time to try to improve the quality of the stream. This scenario is discussed in subsection 4.2.2.2

4.2.2.1 Small buffer: Haste choice scenario

This scenario occurs if while we are playing a stream we potentially run out of buffer, figure 4.4, transactions *1*,*2*,*5* of figure 4.3. The algorithm holds a minimum buffer size value, ”X” in figure 4.3, that in the following examples will be of 3 seconds, the same duration as a stripe.

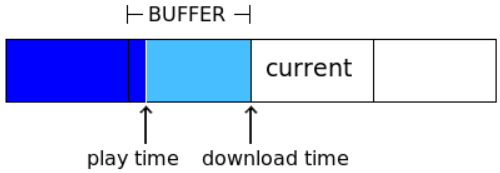
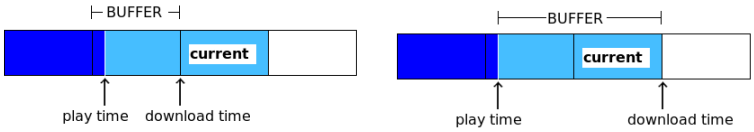


Figure 4.4: Haste choice scenario



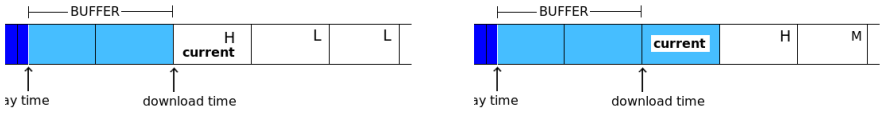
(a) Before (b) After

Figure 4.5: Add current Stripe to the player’s buffer

We have to consider that, differently from the current VoD implementation, we will not send a video piece, in our case Stripe, as soon as it is downloaded but we will wait until we *have* to. This particular implementation allows to consider until the last moment the possibility of switching to a higher quality stream.

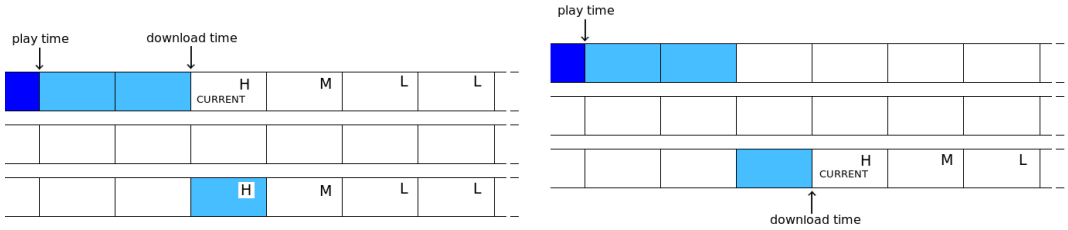
- If the current Stripe has been downloaded, figure 4.5a (transaction *1* of figure 4.3), add the current stripe to the player’s buffer and update the ”download time”, figure 4.5b
  - If we are in low quality and the next Stripe has not been downloaded, download it as soon as possible. Figure 4.6a
  - otherwise, if we are not in low quality or the next Stripe has already been downloaded, assign normal priorities. Figure 4.6b
- Otherwise if a safe-fallback is providen, figure 4.7a. It means we do not have the next Stripe for the same quality stream, therefore we will switch to the lowest quality stream, transactions *2* and than *4* of figure 4.3.

  and as for the previous case:
  - If next Stripe of the lowest quality has not been downloaded, download it as soon as possible. See figure 4.6a



(a) Low quality, haste choice (b) Switch to relaxed scenario

Figure 4.6: Crossroad of the haste choice scenario



(a) Before (b) After

Figure 4.7: Safe fallback scenario

– Otherwise assign normal priorities. As in figure 4.6b

- If we can not provide a safe fallback than we have to download the lowest quality Stripes as soon as possible, figure 4.8

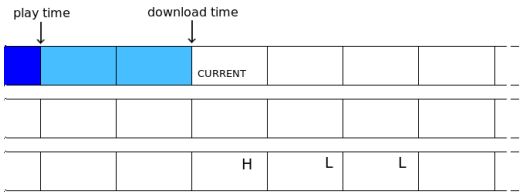


Figure 4.8: No safe fallback

It is clear that if we would not be able to download the low quality Stripe before the payer's buffer finished, we would have to pause the video playback. After the initial conditions for starting the playback are satisfied we can send a "resume" signal.

4.2.2.2 Enough buffer: Relaxed choice scenario

In this case scenario the minimum amount of payer buffer is satisfied, for our examples the player's buffer size will be longer than 3 seconds. In the relaxed choice scenario we count on other values to take our decisions. We will mostly rely on the so called "safebuffer", representing the amount of downloaded buffer in the future. In other words the safebuffer

will be the Stripes that we already downloaded and that we could add to the player's buffer. For simplicity, in the following examples, the value of the safebuffer will be of 3 seconds, the duration of a stripe, the "Y" value in figure 4.3.
The first crossroad of the algorithm at this point will by trying to improve the quality stream depending on the available safebuffer.

- Check if we can improve the quality stream, only if we are not currently playing the highest quality one(transaction *3* of figure 4.3). Figure 4.9 gives an example of the satisfaction of this condition.

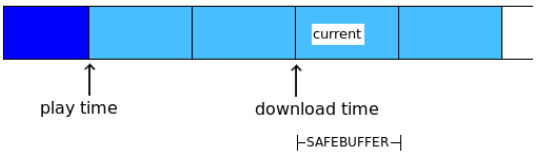


Figure 4.9: Relaxed choice scenario

  – If we already downloaded the higher quality Stripe, figure **??**. This is a condition that will be satisfied once the algorithm has gone through the next step. For a computational reason we need to provide this check before the algorithm performs the next step, but logically we will have a higher quality Stripe only if the algorithm performed the following step in one of its previous iterations.

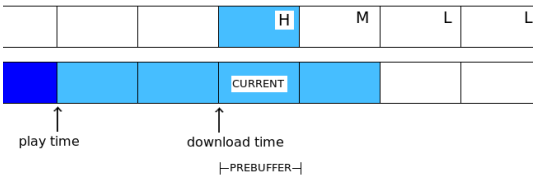


Figure 4.10: Relaxed choice scenario

  If we have the higher quality Stripe, send it to the player's buffer and update the algorithm status.

  – Try to set the priorities for the next quality stream in an optimistic way depending on the given *risk factor*. This is one of the crucial steps of the algorithm. To check the condition for a quality improvement we need to have enough downloaded buffer for the current quality stream. In our implementation it means we need enough saved safebuffer, and to check it we introduced a so-called "risk factor". The risk factor is a value that multiplied by the safe buffer duration determines the condition for trying the quality switch.

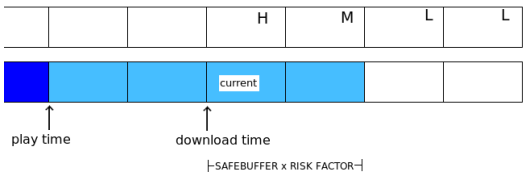


Figure 4.11: Optimistic quality improvement

As we see from figure 4.11 we will set the priorities for the higher quality Stripes. This means that we will start downloading the higher quality stream, hoping that one of the future iterations will end up in the previous step, improving the quality stream. This is defined as transaction *6* of figure 4.3.

– If we did not start the playback just fill the player's buffer

- Otherwise if we completed the download of the current Stripe and we did not start the playback, just fill the player's buffer

- If none of the previous conditions has been satisfied, continue without doing anything.

4.2.3 Algorithm analysis

In this section we are going to discuss the pseudo-code of our novel MBR algorithm. After the design analysis of section 3.3.4 and the theoretical analysis of the previous sections we can now easily understand the algorithm's code.

From line 1 to 10 are the variables used by the algorithm to keep track of the status of the download and to take decisions.

The first set, from line 2 to 6, represents values that determines the algorithm's behaviour. We will see in the experiments chapter, chapter 5, how changing the value of those parameters has a big impact on the algorithm's behaviour and correctness. The lower is the *bufferTime*, the earlier we will have to take a decision of witch Stripe to send to the player's buffer, haste choice scenario, subsection 4.2.2.1. The *priorityDepth* and *forceLowDepth* variables are only used as parameters for calling the setPriorities function. Those variables will determine how in depth the algorithm will assign priorites. For example during our experiments we decided to assign priorities to the next 10 Stripes. Of course the function will take into consideration the architecture design. Therefore it will assign priorities to the medium and low quality stream when downloading the medium quality, to high and low quality stream when downloading the high quality, and only to the low quality after a fallback or for low bit-rate connections. The *forceLowDepth* variable is used when, during the downloading of the low quality stream, the next low quality Stripe is not available. In our experiments we found useful to assign priorities only to the next 2 or 3 following Stripes.

Another important variable that determines the stability of the algorithm is the *riskFactor*. For low values the algorithm will often try to improve quality stream (lines 49-51) while for high values a longer *safeBuffer* is needed. The second set of variables (from line 7 to 10) represents the variables used by the algorithm to keep track of the current status. The *current* variable is a reference to the *following* Stripe. As we already explained, we will not send the *current* or *following* Stripe as soon as it is downloaded, considering the possibility of changing the *current* reference for an higher quality Stripe until we are not running out of buffer (transaction *5* of figure 4.3). In the proposed pseudo-code we consider the possibility of changing the *current* reference to the next Stripe in the same quality stream, *current*+1 (eg. lines 22 and 33), or to the next Stripe in a different quality stream, modifying the value of *current*.quality (lines 26, 37 and 45).

Line 11-12 represents the first check for the end of the stream. If we reached the end of the stream we will start downloading all the remaining Stripes to increase the pieces availability (figure 3.13). After this initial check we have the major crossroad of the algorithm: from line 13 to 39 the haste choice scenario, while from line 40 to 68 the relaxed choice scenario.

The three bullets of subsection 4.2.2.1 correspond to the three conditions at lines 15, 24 and 35. All the time a Stripe is sent to the player's buffer, eg. line 16, we will update the reference to the *current* Stripe, eg. line 22, and increase the *downloadTime* by the duration of a Stripe, in our case three seconds, eg. line 23. A new function, called safeFallback, appears on line 24. This function will only check if the relative Stripe of the lowest quality stream has been downloaded or not. The result of this check will be the necessary condition to switch to the lowest quality stream in case there is not enough bandwidth to stay on the current quality.

During the relaxed choice scenario, from line 40 to line 68, the algorithm will try to increase the quality stream. Because the algorithm will never download an unnecessary quality stream, except for the low quality one, for the switch we need to perform two steps. The fist step consist of assigning priorities to the higher quality stream. This happens if the condition of line 49 is satisfied, in other words is the amount of already downloaded Stripes of the current quality, called *safeBuffer*, is larger that the $preBuffer \times riskFactor$. After assigning priorities to the higher quality stream the algorithm will start downloading it, and if during one of the following iterations the higher quality Stripe, relative to the *current* one, has been successfully downloaded, line 44, the algorithm will switch to the higher quality stream.

MBR Algorithm

```
1: //Parameters used to change the algorithm's behaviour
2: priorityDepth        //how in depth we assign priorities
3: forceLowDepth        //how in depth we assign priorities for "force low"
4: bufferTime           //minimum duration of the player's buffer
5: preBuffer            //minimum duration of the already downloaded Stripes
6: riskFactor           //determines the condition for a quality change

   //Parameters used by the algorithm to keep track of the downloading status
7: current              //the current processed Stripe
8: playTime             //playback position
9: downloadTime         //the duration of the already downloaded Stripes: the player's buffer
10: safeBuffer          //duration of the already downloaded Stripes of the current quality

11: if end of stream then
12:     setPriorities()
13: else if (playTime ≥ 0)&&(downloadTime − playTime ≤ bufferTime) then
14:     //we need to take a decision, buffer too small
15:     if current is complete then
16:         downloadBuffer.add(current)
17:         if (current.quality == low)&&(current + 1is not complete) then
18:             setPriorities(forceLowDepth)
19:         else
20:             setPriorities(priorityDepth)
21:         end if
22:         current ← current+1
23:         downloadTime ← downloadTime+ length(Stripe)
24:     else if safeFallback() then
25:         //check the presence of the low quality piece
26:         current.quality ← low
27:         downloadBuffer.add(current)
28:         if current + 1 is not complete then
29:             setPriorities(forceLowDepth)
30:         else
31:             setPriorities(priorityDepth)
32:         end if
33:         current ← current+1
34:         downloadTime ← downloadTime+ length(Stripe)
35:     else
36:         //we need to download the low quality Stripe as soon a possible
37:         current.quality ← low
38:         setPriorities(forceLowDepth)
39:     end if
```

```
40: else
41:    //Normal iteration, we have enough player's buffer
42:    //check for improving quality
43:    if (current.quality ≠ high)&&(safeBuffer ≥ preBuffer) then
44:       if current.quality(next) is complete then
45:          current.quality ← current.quality(next)
46:          downloadBuffer.add(current)
47:          current ← current+1
48:          downloadTime ← downloadTime+ length(Stripe)
49:       else if (safeBuffer ≥ preBuffer × riskFactor)&&(playTime) > 0 then
50:          //try to set priorities on an optimistic way
51:          setPriorities(increase quality)
52:       else if playTime == 0 then
53:          //fill the player's buffer
54:          downloadBuffer.add( current )
55:          setPriorities(priorityDepth)
56:          current ← current+1
57:          downloadTime ← downloadTime+ length(Stripe)
58:       end if
59:    else if (currentis complete)&&(playTime == 0) then
60:       //fill the player's buffer
61:       downloadBuffer.add( current )
62:       setPriorities(priorityDepth)
63:       current ← current+1
64:       downloadTime ← downloadTime+ length(Stripe)
65:    else
66:       wait to complete the current piece or to switch to a different quality
67:    end if
68: end if
```

Chapter 5

Experiments

In this chapter we present the experiments performed with our novel algorithm. We will demonstrate how changing the algorithm's parameters, such as the risk factor or the size of the buffers, will change the algorithm's behaviour advantaging certain scenarios.
The values used for graphs are taken from a log file updated every iteration of the algorithm. This gives an approximation of around half a second, as the algorithm is re-scheduled two times per second.

From our experiments we run the network on a local machine by setting up a tracker that already hold the downloaded torrent. The Swarmplayer will connect to the tracker through the localhost connection and start downloading our torrent file. We used a functionality offered by the Tribler Core to set the bandwidth limits, simulating different scenarious.

Figure 5.1 shows how the results are going to be presented. On the axis we have a time line where the numbers represents the seconds of the player's playback.

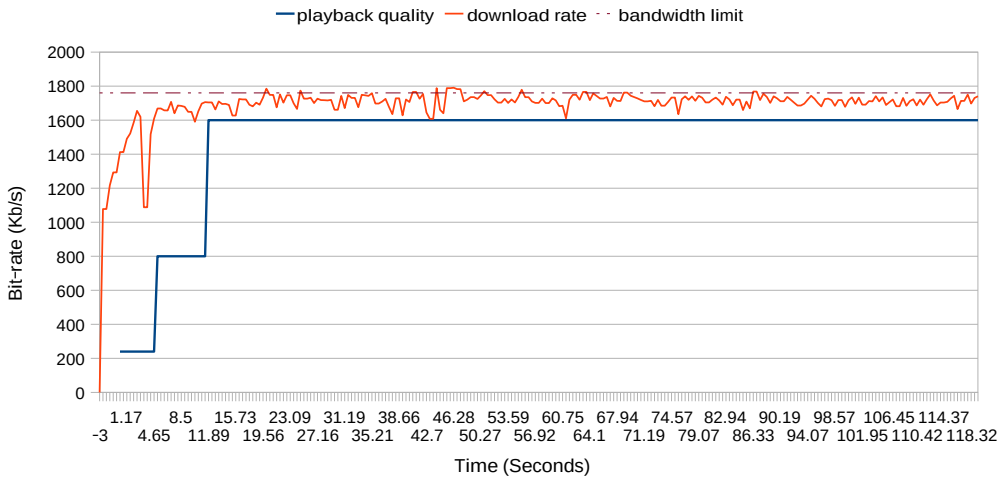


Figure 5.1: High quality, stable environment

The progress of the blue curve represents the player's playback time. The low, medium and high quality streams are respectively located at 240 Kbit/s, 800 kbit/s and 1.6 MBit/s. This is only a theoretical approximation, based on the stream's average bit-rate, to facilitate reading the graphs. As we know from the design chapter stripes are encoded with a variable bit-rate and the download process only depends on the stripe size, not on its bit-rate.

The set of date used to draw the graphs is initialized with the first iteration of the algorithm. We can therefore see the start-up time, from the initialization of the download until the start of the playback, as the time until the red curve starts. The normal **start-up time** is around the 3-5 seconds, for high speed connections (see figure 5.2 and figure 5.6), until a maximum of 10 seconds for connections slower than 280 Kbit/s (see figure 5.7).

The red curve represents the downloading rate over time, while the dashed burgundy curve represents the downloading rate limit used for our experiments. As the reader might notice, some times the downloading rate exceeds the rate limit. This is caused by the fact that the download rate is calculated over the amount of data downloaded on the rage of half second.

Section 5.1 will show the results of running the Swarmplayer in a stable environment, no peers disconnecting from the system and a constant downloading bit-rate. Section 5.2 will show the results in a variable bandwidth. We tested the system by changing at run-time the available bandwidth, observing the reactions of the algorithm to a sudden bandwidth drop or bandwidth increase.

5.1 Stable environment experiments

In this section we will show the results of executing the MBR algorithm in stable environments. For high bit-rates we limited the bandwidth to 1.750-1.9 Mbit/s, for medium bit-rates we limited it to 950 Kbit/s while for low bit-rates to 280 Kbit/s [1]

5.1.1 High bit-rates

With high bit-rates the algorithms behaves in the correct way. Figure 5.1 shows how the quality increases as soon as enough *safebuffer* has been downloaded. As we previously saw in section 3.3.4 and section 4.2 the condition that determines a quality improvement is given by the *prebuffer* multiplied to the *risk factor*. For all our examples we used a prebuffer of three seconds and we see the reaction of modifying the risk factor from the different time of quality switch between figure 5.1, figure 5.2 and figure 5.3.

For the experiment of figure 5.1 we used the value 2 for the risk factor, while for figures 5.2 and 5.3 we increased the risk factor to values higher than 3 and observed the behaviour.

[1] We where actually able to reduce the bandwidth limitation to 226 Kbit/s by reducing the torrent chunk size, see subsection 5.1.3.

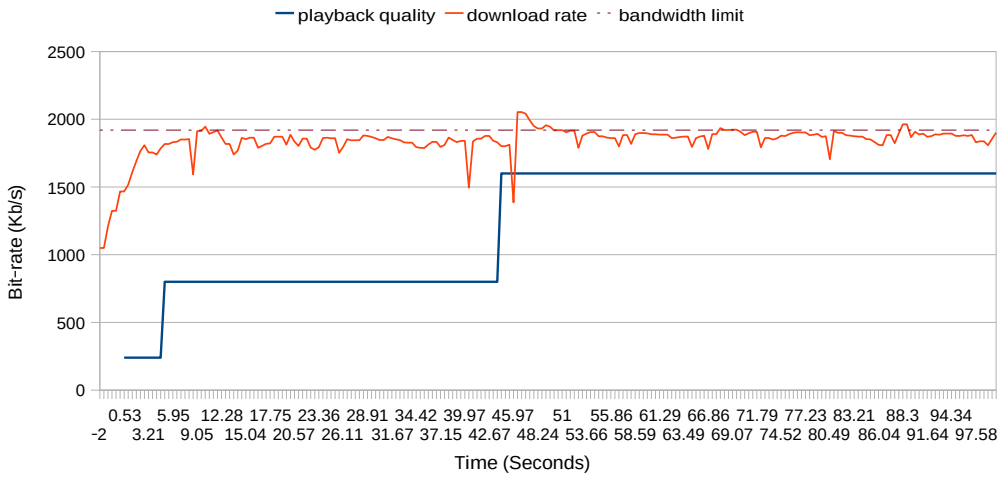


Figure 5.2: Stable environment; risk factor = 3

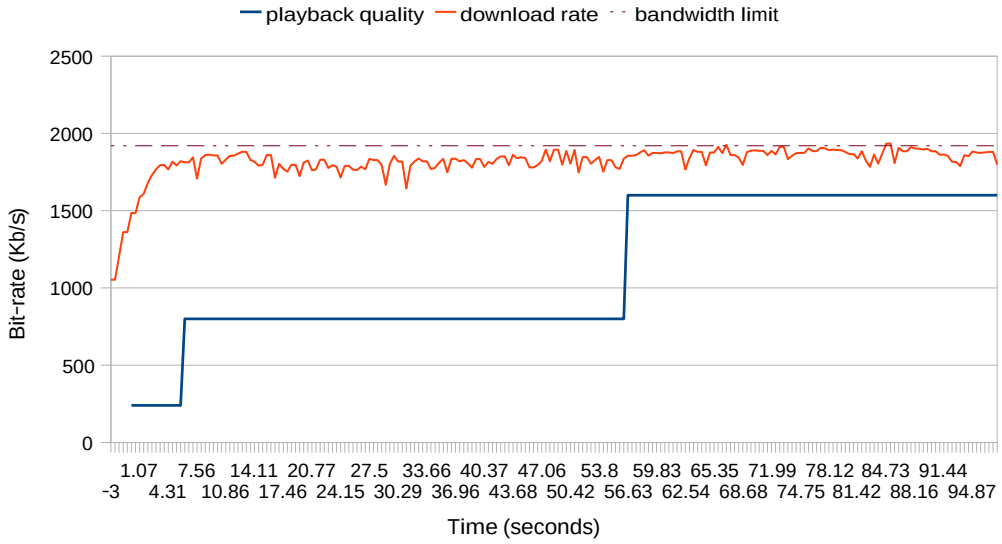


Figure 5.3: Stable environment; risk factor = 3.5

By increasing the risk factor we increase the time needed to reach the highest quality stream.

Thus with a low risk factor and abundant bandwidth our algorithm quickly shifts to the highest quality stream. We will see in the next sections that the risk factor is not only related to the quality switch time.

5.1.2 Medium bit-rates

Figure 5.4 shows the algorithm's behaviour with a bandwidth limit of 940 Kbit/s. With medium bit-rates we need to take in consideration different aspect. We noticed that this scenario is the most unstable. When downloading one of the intermediate quality streams at a certain point the algorithm will try increase the quality.

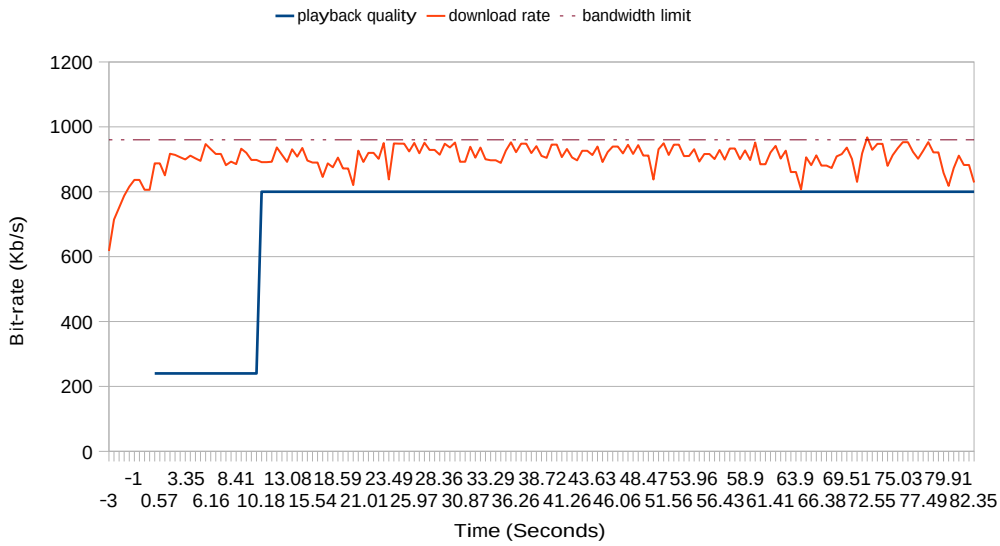


Figure 5.4: Medium quality, stable scenario

In figure 5.5 and figure 5.6 is clearly visible how the algorithm switches to the lowest stream 110 seconds of video playback. This behaviour is caused by a quality improvement attempt, the spare bandwidth limits imposed buring the experiments and the larger size of the following Stripes of the medium quality.
If the available bandwidth is just enough to download the current stream, the result of trying to increase the quality could differ from what aspected. By concentrating the available bandwidth on the higher quality stream we could loose important time do download the current one. Once the quality increase attempt fails, if the following Stripes of the intermediate quality are holding a big set of data, like those elected from action scenes, than the available bandwidth could not be enough to avoid a quality fallback.

The difference between the two figures is given by a different assignment of the risk factor. Figure 5.5 shows a more stable reaction cased by a higher value for the risk factor. The first fallback of figure 5.6 explains an earlier attempt for a quality switch.

We now see how it this scenario an higher value for the risk factor is prefered. This contrasts with the lower value prefered in a high bandwidth scenario.

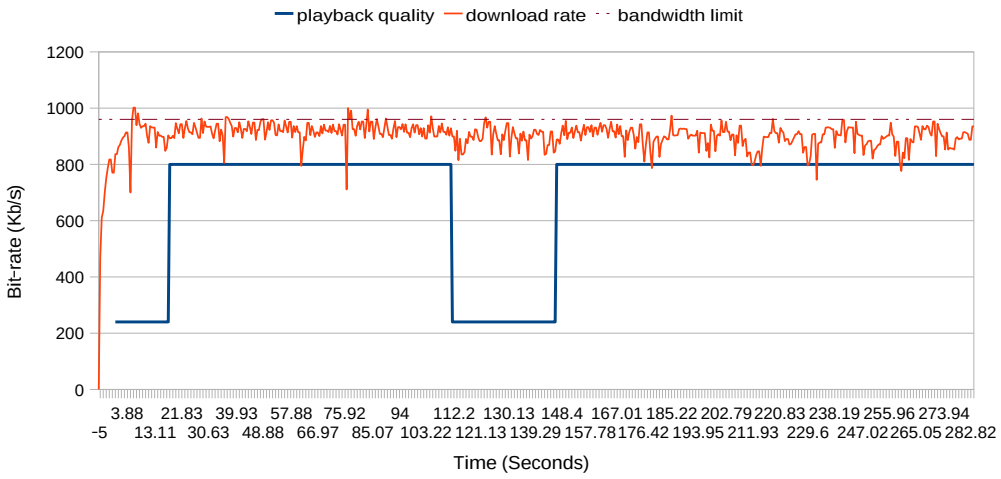


Figure 5.5: Stable scenario, risk factor = 3

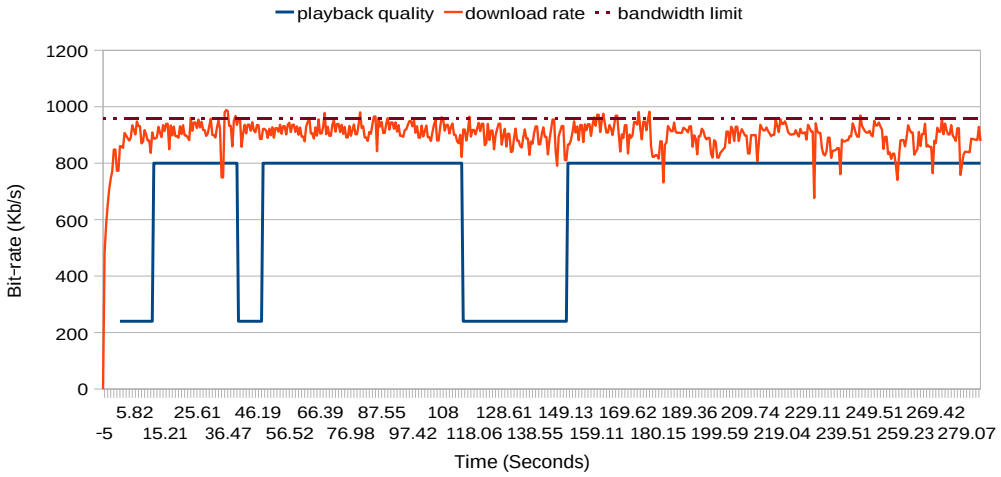


Figure 5.6: Stable scenario, risk factor = 2

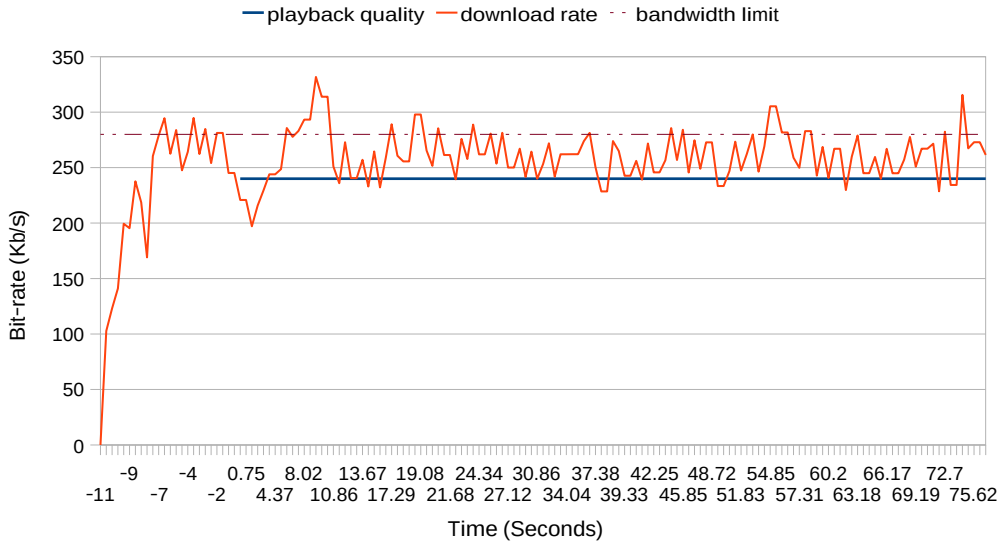


Figure 5.7: Low bandwidth limit

5.1.3 Low bit-rate

Figure 5.7 is an example of the algorithm's behaviour in a low bandwidth scenario. The algorithm offers a good stability, given by the fact that differently than in a medium bandwidth scenario trying to increase the quality stream is not dangerous. As we saw in section 3.3.3, to provide the safe fall-back we will always download the low quality stream together with an higher one. Therefore the same priority set will be applied to the low quality stream even when trying to improve quality.

Surprisingly performing the same experiments with torrent files created differently, torrent files with a smaller chunk size perform better in low bit-rate scenarios. This will probably be caused by the internal implementation of the file selector of BitTorrent.

5.2 Agility Experiments

This chapter regards the algorithms behaviour in a variable bit-rate environment. This is a quite frequent situation caused by the heterogeneous nature of the Internet accesses, see chapter 2.1. It can occur because a seeder, with a fast upload connection, disconnects from the system or because our available bandwidth is shared between different applications or downloads.

Whatever is the condition that causes the decreasing of the available bandwidth the algorithm performed quite good buring our experiments, as we can see from figure 5.10 and figure 5.11.

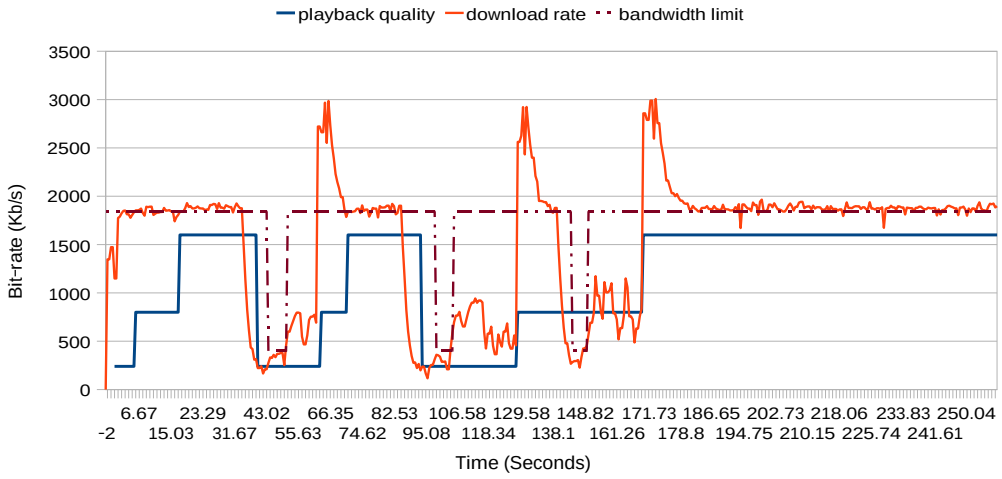


Figure 5.8: 3 spikes down, risk factor = 2

As we repeatedly saw, to provide a safe fallback and the algorithm will always download the lowest quality stream together with the medium or high quality streams. Therefore the algorithm will switch back to the lowest quality stream if the available bandwidth decreases and then trying to increase it if there is enough bandwidth.

In the experiments of figure 5.8 and figure 5.9 we simulated a three spike down scenario. We suddenly reduced three times the bandwidth limit, from 1.87 Mb/s to 400 Kb/s, for six seconds. It is clearly visible how increasing the risk factor encreases the stability of the algorithm as previously saw.

In the experiment of figure 5.10 we reduce the bandwidth limit from 1.87 Mb/s to 400 Kb/s to see how the algorithm reacts to a sudden bandwidth drop. After few seconds we increase the bandwidth limit first to 1 Mb/s and than back to 1.8 Mb/s to observe the time needed by the algorithm to resume the high quality stream.
In the experiments of figure 5.11 we reduced the bandwidth from 1.87 Mb/s to 1 Mb/s observing the time needed by the algorithm to switch to the medium quality stream. During a second experiment, in the same scenario as the experiment of figure 5.11, we increased the risk factor increasing the stability of the algorithm. We can see from figure 5.12 how increasing the risk factor from 2 to 3, increases the time needed for a quality switch.

Running multiple experiments in the same scenario we noticed that the algorithm performs always differently. This is given by the dynamic nature of the environment. It depends on the tracker status and downloading process that is unpredictable.

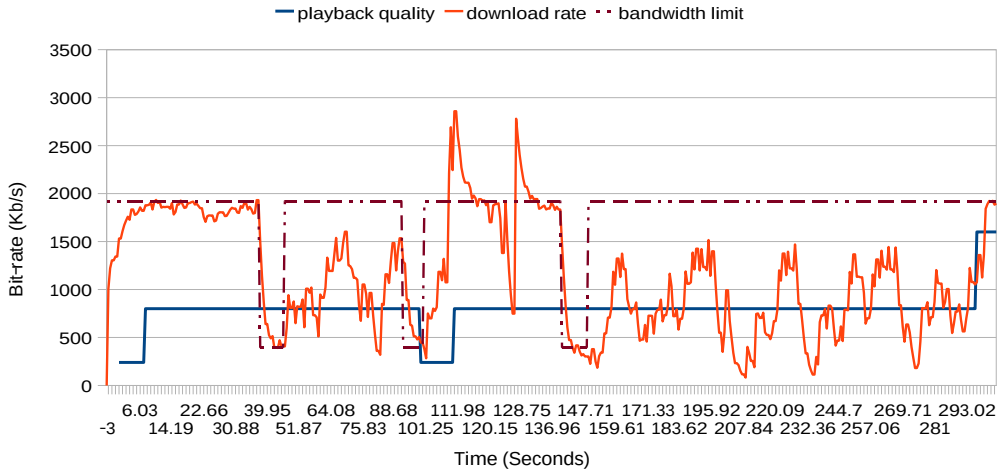


Figure 5.9: 3 spikes down, risk factor = 3

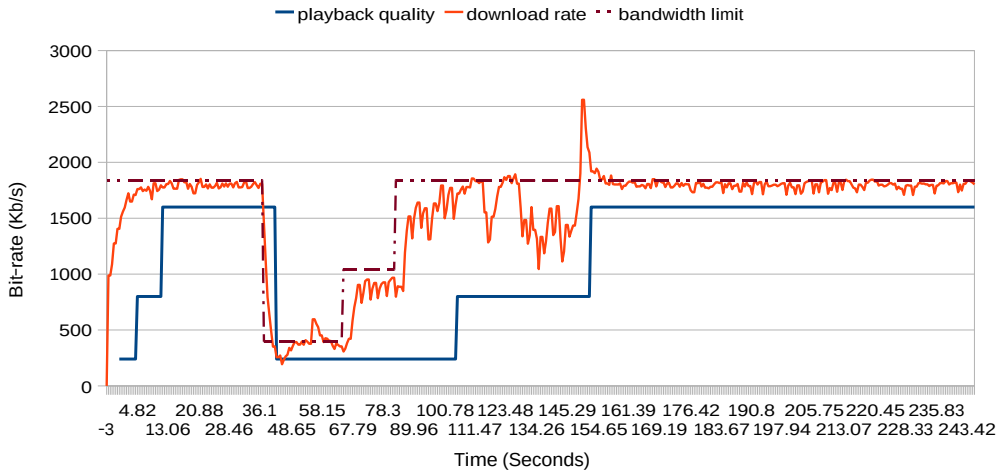


Figure 5.10: Variable quality, bandwidth fallback and resume

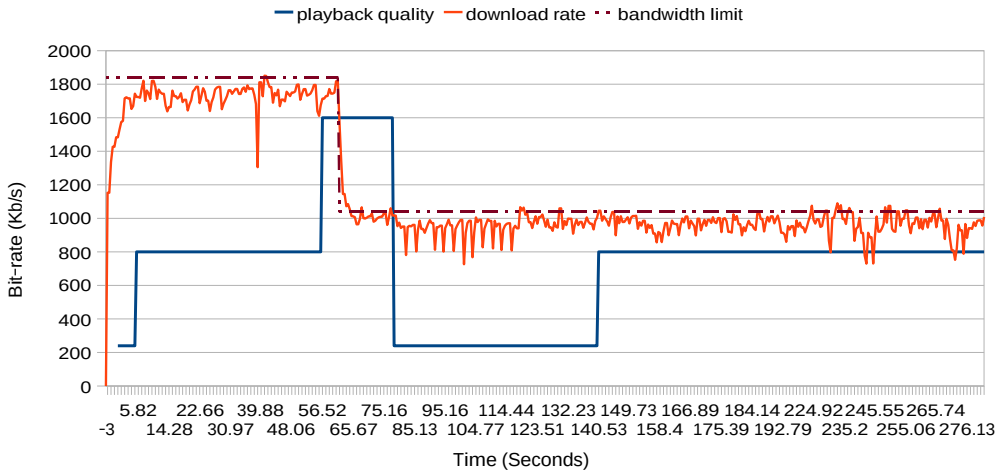


Figure 5.11: Variable quality, bandwidth drop

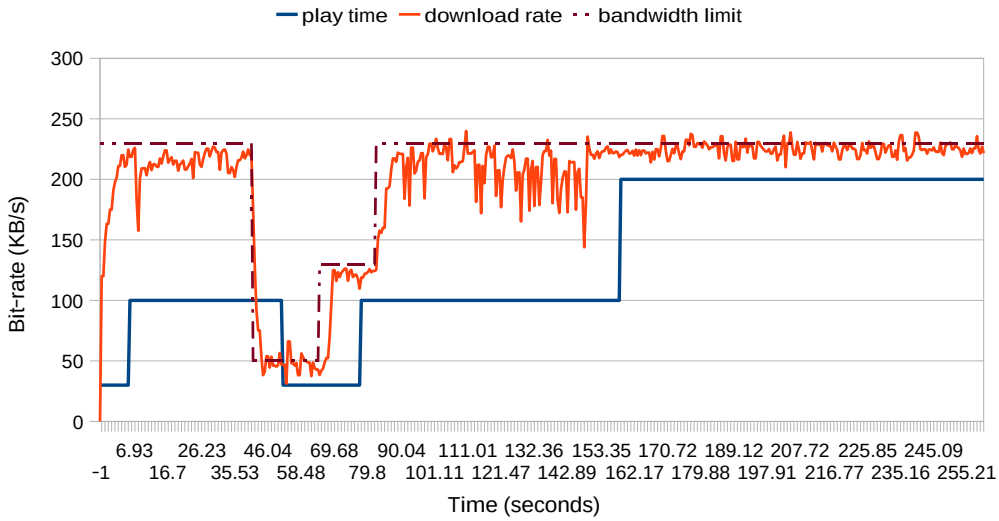


Figure 5.12: Variable quality, variable bandwidth

Chapter 6

Conclusions and Future Work

In this chapter we give a summary of the project's contributions. After the overview of our conclusions in section 6.1, we will reflect on some characteristics of our architecture in section 6.2. Finally, some ideas for future research will be discussed in section 6.3.

6.1 Conclusions

Video-on-Demand has grown exponentially over the last years. Faster Internet accesses and more powerfull technologies are the causes of this phenomenon. The current techniques that implement a VoD functionality all rely on a constant bit-rate encoding. This splits the audience in two major categories: users that have enough bandwidth to watch a video file on streaming and users who have to wait a certain time to watch the video file continuously. This situation gets worse if we consider delivering VoD over P2P networks. The problem is the heterogeneous Internet accesses that complicate a real time multimedia delivery.

Theoretically, for the survival of a P2P system, peers could download only as much as they upload and, considering that the current Internet accesses are mostly ADSL, this makes the introduction of a VoD functionality harder. In order to solve this problem, we have designed and implemented a novel algorithm that, depending on the available bandwidth, switches between three different quality streams. We also introduced a novel encoding methodology to create a torrent file holding one single movie encoded into three quality streams of proportional size. Furthermore we modified the internal behaviour of the multimedia player VLC to allow a flicker free playback, needed for a high quality experience of the user.

Our novel architecture allows every user to watch a video stream continuously, taking advantage of their bandwidth. This particular implementation allows every user with at least a 250 Kbit/s bandwidth connection to watch a video file on streaming over a P2P network. Our algorithm will increase the quality stream as soon as enough bandwidth is available, until the highest quality stream, who's average bit-rate is around 1.5 Mbit/s. Our architecture has been implemented into the Tribler project, introducing also new functionalities such as

an ordered download process and VLC's playlist management.

6.2 Discussion/Reflection

VoD solutions are a rather hot concept in our times. We propose the first open-source implementation of a variable bit-rate solution for VoD. During the design of the architecture some ideas came up:

- Considering the current state of P2P networks, our architecture could be revolutionary if it would be widely used in the network community. Currently almost every peer shares a large amount of video files to gain a high upload ratio. If a peer would like to watch a movie, fist it has to be downloaded on the local machine and opened with a multimedia player. Our idea of a future evolution would be a system where every peer holds not more than 2-3 movies encoded with our methodology, to hold a good upload ratio. If the peer wants to watch a video file they can just start watching it through our novel architecture without having to wait for the download to be finished. This scenario would aim at a more homogeneous distribution of multimedia content over peers of a file sharing network.
- We belive that our implementation could achieve optimal results in a mixed environment. We have a mixed environment when a client-server architecture coexists with a P2P network, figure 6.1. Considering that the majority of the current VoD systems are client-server based, this implementation would drastically reduce server's workload.

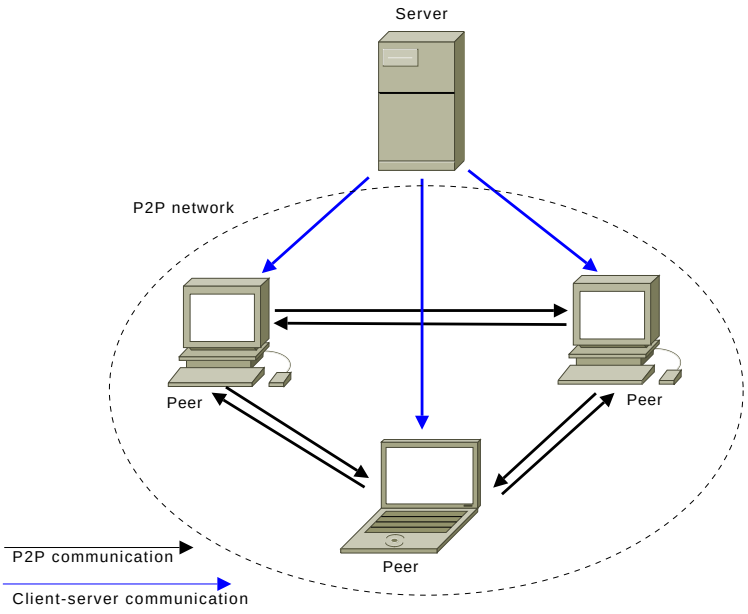


Figure 6.1: Mixed scenario, client-server arch. + P2P network

- Our solution does not need a central server to coordinate peers or syncronize transmissions. The architecture has been designed to work in a completly decentralized P2P environment.
- Unlike current VoD implementations our architecture does not rely on a constant bit-rate encoding. Furthermore we do not think in terms of bit-rate but only in terms of available bandwidth. We assume a peer has at least enough bandwidth to watch the lowest quality stream.
- It is not into the scope of this thesis to consider the pieces availability for the proposed MBR architecture. Anyway we think the algorithm performs well as any peer in the network has interest in downloading the lowest quality stream, to provide a safe fall-back scenario. Therefore even if peers are downloading different qualities they will allways share the low quality stream. This allows bidirectional communication between peers, while normally the VoD architecture provides only unidirectional communications.

6.3 Future work

During our research we have concluded that the following fields should be further investigated:

- The encoding methodology has to be investigated. Currently the used codecs are under development and still in an early stage. This has put some limitation in the encoding procedure, forcing to encode the streams using generic parameters. Once the codec libraries will be completly merged with the encoding applications, we should be able to encode the streams without changing parameters such as frame size or frame rate. By only changing internal parameters of the decoders we could avoid re-initializing them, getting rid of the 40-60 ms latency needed during a quality switch.
- The algorithm has to be simulated in a real world scenario. Observing the algorithm's behaviour when interacting with a real P2P network on the Internet
- The necessity of a dynamic risk factor has to be investigated. Depending on the codec's investigation results, if a flicker-free quality switch can not be implemented, the necessity for an adaptive risk factor could be investigated. On the other hand a static risk factor is prefered if we can gain an unnoticeable quality switch, having the algorithm switching between qualities in a transparent way for the final user.

As future work we take into consideration the direct integration of a chunk priority assignment, without using BitTorrent's fileselector to assign individual priorities. During our experiments we noticed some incongruences with the fileselector when downloading at low bit-rates, see section 5.1.3. Another utility that will be soon implemented is a progress bar for the swarmplayer to convey the video progress.

Bibliography

[1] ABC: Another BitTorrent Client. http://sf.net/projects/pingpong-abc

[2] N. Andrade, M. Mowbray, A. Lima, G. Wagner, and M. Ripeanu: Influences on cooperation in bittorrent communities. *In P2PECON '05: Proc. of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems, 2005. ACM Press.*

[3] P. Baccichet, T. Schierl, T. Wiegand, B. Girod: Low-delay peer-to-peer streaming using scalable video coding *Packet Video 2007, 12-13 Nov. 2007*

[4] B.Birney: Intelligent Streaming. http://www.microsoft.com/windows/windowsmedia/howto/articles/intstreaming.aspx *May 2003*

[5] A. Bakker, P. Garbacki, J. Pouwelse: Cooperative Download Extension, Version 1 https://www.tribler.org/attachment/wiki/CooperativeDownload/CooperativeDownload-20060227.pdf *February 27, 2006*

[6] Al Bovik: Handbook of Image & Video Processing. *Elsevierr Academic Press, 2005*

[7] C. Y. Chan and Jack Y. B. Lee: On Transmission Scheduling in a Server-less Video-on-Demand System *Springer-Verlag Berlin Heidelberg, 2003*

[8] B. Cohen: Bittorent protocol. http://www.bittorrent.org/beps/bep_0003.html

[9] B. Cohen: Incentives build robustness in bittorrent. *In Proc. of the 1st Workshop on Economics of Peer-to-Peer Systems, 2003.*

[10] CoolStreaming broadcast TV http://www.coolstreaming.us/

[11] G.J.Conklin, G.S.Greenbaum, K.O.Lillevold, A.F.Lippman and Y.A.Reznikm: Video Coding for Streaming Media Delivery on the Internet *IEEE Transactions on Circuits and Systems for Video Technology, Vol.11, No. 3, Mar. 2001*

[12] Philippe de Cuetos, Keith W. Ross: Adaptive rate control for streaming stored fine-grained scalable video *ACM Portal, NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video, May 2002*

[13] W. Dapeng, Y.T. Hou, Z. Wenwu, Z. Ya-Qin, J.M. Peha: Streaming video over the Internet: approaches and directions *Circuits and Systems for Video Technology, IEEE Transactions on Volume: 11 Issue: 3 Mar 2001*

[14] Emule file sharing application: `http://sourceforge.net/projects/emule/`

[15] FFmpeg2theora: A simple converter to create Ogg Theora files. `http://v2v.cc/~j/ffmpeg2theora/`

[16] FFmpeg documentation. `http://ffmpeg.mplayerhq.hu/ffmpeg-doc.html`

[17] FFmpeg: audio/video editing program. `http://ffmpeg.mplayerhq.hu/`

[18] Gnutella protol documentation `http://gnet-specs.gnufu.net/`

[19] Mei Guo, Yan Lu, , Feng Wu, D. Zhao, and Wen Gao: WynerZiv Switching Scheme for Multiple Bit-Rate Video Streaming *IEEE Tansactions on Circuits and Systems for Video Technology, Vol.11, No. 5, May 2008*

[20] D. Harrison, B. Cohen: BitTorrent `http://www.bittorrent.org/index.html`

[21] T. Hossfeld, K. Leibnitz: A qualitative measurement survey of popular Internet-based IPTV systems *Communications and Electronics, 2008. ICCE 2008. Second International Conference on 4-6 June 2008 Page(s):156 - 161*

[22] C.Huang, P.A.Chou, A.Klemets. Optimal Control Of Multiple Bit Rates For Streaming Media Picture *Coding Symposium, San Francisco, CA, Dec. 2004.*

[23] Kademlia specifications: `http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html`

[24] Kazaa P2P file sharing application `http://www.kazaa.com`

[25] Seth Kenlon. Video codecs and the free world: Volume 2008 ,In *Linux Journal, Volume 2008, Issue 166 (February 2008), Article No. 10*

[26] S. Kim, C. Kim, Y. Cho: An effective resource management for variable bit rate video-on-demand server *EUROMICRO 97. 'New Frontiers of Information Technology'. Short Contributions., Proceedings of the 23rd Euromicro Conference; 1-4 Sept. 1997 Page(s):74 - 79*

[27] K. Leibnitz, T. Hofeld, N. Wakamiya, and M. Murata: Peer-to-peer vs. client/server: Reliability and efficiency of a content distribution service *in Proc. of ITC-20, (Ottawa, Canada), June 2007.*

[28] Weiping Li: Overview of fine granularity scalability in MPEG-4 video standard *Circuits and Systems for Video Technology, IEEE Transactions on Volume: 11 Issue: 3 Mar 2001*

[29] C. Loeser, P. Altenbernd, M. Ditze, W. Mueller: Distributed video on demand services on peer to peer basis *Proceedings of the First International Workshop on Real-Time, 2002*

[30] Chris Loeser, Franz Rammig: GRUSEL: A Self Optimizing, Bandwidth Aware Video on Demand P2P Application *IEEE Computer Society. May 2004*

[31] J.L. Mitchel, W.B. Pennebaker, C.E. Fogg, G.J. LeGall. MPEG video compression standard. *Champman & Hall, 1996*

[32] Jack Moffitt: Ogg VorbisOpen, Free AudioSet Your Media Free. *Linux Journal, Volume 2001 , Issue 81es, Art. No. 9, 2001*

[33] J. J. D. Mol, J. A. Pouwelse, D. H. J. Epema, H. J. Sips: Free-Riding, Fairness, and Firewalls in P2P File-Sharing *IEEE Computer Society, P2P '08: Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing, Sept. 2008*

[34] J.J.D. Mol, J.A. Pouwelse, M. Meulpolder, D.H.J. Epema and H.J. Sips: Give-to-Get: Free-riding-resilient Video-on-Demand in P2P Systems. *in P2P Systems, Proc. of SPIE, Multimedia Computing and Networking Conference (MMCN), 2008*

[35] MPEG systems overview, `http://www.mpeg.org/MPEG/mpeg-systems-resources-and-software/mpeg-systems-overview.html`

[36] Multimedia Wikipedia. `http://wiki.multimedia.cx`

[37] S. Pfeiffer: RFC3533: The Ogg Encapsulation Format Version 0, *RFC Editor United States, 2003*

[38] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, H. J. Sips: TRIBLER: a social-based peer-to-peer system *John Wiley and Sons Ltd., Concurrency and Computation: Practice & Experience, Volume 20 Issue 2, Feb. 2008*

[39] PPlive internet TV `http://www.pplive.com`

[40] R. Rejaie, A. Ortega: PALS: peer-to-peer adaptive layered streaming *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video, June 2003*

[41] J. Rexford, D. Towsley: Smoothing variable-bit-rate video in an internetwork *Networking, IEEE/ACM Transactions on Volume 7, Issue 2, April 1999 Page(s):202 - 215*

[42] S. Sen, D. Towsley, Z. Zhi-Li , J.K Dey: Optimal multicast smoothing of streaming video over the Internet *S*elected Areas in Communications, IEEE Journal on Volume: 20 Issue: 7 Sep 2002

[43] Y. Shen, Z. Liu, S.S. Panwar, K.W. Ross, Y. Wang: Streaming Layered Encoded Video Using Peers *Multimedia and Expo, 2005. ICME 2005. IEEE International, 2005*

[44] Xiaoyan Sun, Feng Wu, Shipeng Li, Wen Gao, Ya-Qin Zhang: Seamless switching of scalable video bitstreams for efficient streaming *Multimedia, IEEE Transactions on Volume: 6 Issue: 2 Page(s): 291- 303 April 2004*

[45] uTorrent P2P client `http://www.utorrent.com`

[46] VideoLAN, VLC multimedia player, `http://www.videolan.org/`

[47] Vuze P2P programm `http://www.vuze.com/`