On the HTTP segment streaming potentials and performance improvements.

by

Tomas Kupka

Doctoral Dissertation submitted
to the Faculty of Mathematics and Natural Sciences
at the University of Oslo
in partial fulfilment of the requirements for
the degree Philosophiae Doctor

February 2013

© Tomas Kupka, 2013

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1360

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika publishing merely in connection with the thesis defence. Kindly direct all inquiries regarding the thesis to the copyright holder or the unit which grants the doctorate.

Abstract

Video streaming has gone a long way from its early years in the 90's. Today, the prevailing technique to stream live and video on demand (VoD) content is adaptive HTTP segment streaming as used by the solutions from for example Apple, Microsoft, and Adobe. The reasons are its simple deployment and management. The HTTP infrastructure, including HTTP proxies, caches and in general Content Delivery Networks (CDNs), is already deployed. Furthermore, HTTP is the de facto standard protocol of the Internet and is therefore allowed to pass through most firewalls and Network Address Translation (NAT) devices. The goal of this thesis is to investigate the possible uses of adaptive HTTP segment streaming beyond the classical linear streaming and to look at ways to make HTTP servers dealing with HTTP segment streaming traffic more efficient.

In addition to the deployment and management benefits, the segmentation of video opens new application possibilities. In this thesis, we investigate those first. For example, we demonstrate on the fly creation of custom video playlists containing only content relevant to a user query. Using user surveys, we show, that it not only saves time to automatically get playlists created from relevant video excerpts, but the user experience increases significantly as well.

However, already the basic capabilities of HTTP segment streaming, i.e., streaming of live and on demand video, are very popular and are creating a huge amount of network traffic. Our analysis of logs provided by a Norwegian streaming provider Comoyo indicates that a substantial amount of the traffic data must be served from places other than the origin server. Since a substantial part of the traffic comes from places other than the origin server, it is important that effective and efficient use of resources not only takes place on the origin server, but also on other, possibly HTTP segment streaming unaware servers.

The HTTP segment streaming unaware servers handle segment streaming data as any other type of web data (HTML pages, images, CSS files, javascript files etc.). It is important to look at how the effectiveness of data delivery from this kind of servers can be improved, because there might be potentially many "off the shelf" servers serving video segments (be it a homemade solution or an HTTP streaming-unaware CDN server). In general, there are three possible places to improve the situation: on the server, in the network and on the client. To improve the situation in the network between the server and the client is generally impossible for a streaming provider. Improving things on the server is possible, but difficult because the serving server might be out of the control of the streaming provider. Best chances are to improve things on the client. Therefore, the major part of this thesis deals with the proposal and evaluation of different modifications to the client-side and only some light modifications to the server-side. In particular, the thesis looks at two types of bottlenecks that can occur. The thesis shows how to deal with a client-side bottleneck using multiple links. In this context, we propose and evaluate a scheduler for partial segment requests. After that, we discuss different techniques on how to deal with a server-side bottleneck with for example different

modifications on the transport layer (TCP congestion control variant, TCP Congestion Window [1] (CWND) limitation) and the application layer (the encoding of segments, segment request strategy).

The driving force behind many of these modifications is the *on-off* traffic pattern that HTTP segment streaming traffic exhibits. The *on-off* traffic leads in many cases of live streaming to request synchronization as explained in this thesis. The synchronization in turn leads to increased packet loss and hence to a downgrade of throughput, which exhibits itself by decreased segment bitrate, i.e., lower quality of experience. We find that distributing client requests over time by means of a different client request strategy yields good results in terms of quality and the number of clients a server can handle. Other modifications like the limiting of the CWND or using a different congestion control algorithm can also help in many cases.

All in all, this thesis explores the potentials of adaptive HTTP segment streaming beyond the linear video streaming and it explores the possibilities to increase the performance of HTTP segment streaming servers.

Acknowledgements

First of all, I would like to thank my supervisors Prof. Pål Halvorsen and Prof. Carsten Griwodz for reviewing this thesis and for interesting discussions. I would also like to thank all my great colleagues at the Simula Research Laboratory for providing a superb working environment.

However, most of all, I would like to thank my family, David, Zuzka, my parents, my grandparents and last but not least my perfect girlfriend Kerstin for always being there for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A video is nothing more than a series of static images shown quickly one after another in order to create an illusion of continuity. Yet, it took almost 30 years after the Internet was born in 1969 for the commercial video streaming applications to break through (RealPlayer [5], ActiveMovie [6], QuickTime 4 [7]). The responsible factors are the large computing and bandwidth requirements associated with the video encoding and transmission. To stream (transfer) a video, each video image must be transmitted to the remote site up to a precisely defined time to be ready in time for playout. Furthermore, because each image requires a rather big amount of bytes to be transfered and because a lot of images is required per second to create the illusion of continuity, the bandwidth requirement was just too high for the early days of the Internet, i.e., even for low resolution videos. Compression can be used to reduce the number of bytes required per image, but, in general, the computational complexity of compression grows with the number of bytes that can be saved, and therefore, powerful computers are needed. It was first in 1997 that both the computational and bandwidth requirements could be fulfilled, and successful commercial Internet video streaming started to boom with players from for example RealNetworks [5], Microsoft [8] and Apple [7][1].

Today, video streaming is one of, if not the most popular service of the Internet. Only YouTube alone delivers more than four billion hours of video globally every month [10]. Furthermore, many major (sports) events like the European soccer leagues, NBA basketball and NFL football are streamed *live* with only a few seconds delay. Other examples include the 2010 Winter Olympics, 2010 FIFA World Cup and NFL Super Bowl, which were successfully streamed to millions of concurrent users over the Internet, supporting wide range of devices ranging from mobile phones to HD displays. The list of video streaming providers is growing and includes companies like HBO [11], Viasat [12], TV 2 Sumo [13], NRK [14]. But there exist also pure Internet streaming companies like Netflix [15] and Comoyo [16]. Thus, the amount of Internet video traffic has been steadily growing and is predicted to grow even more. For example, Cisco is predicting the video traffic to quadruple by 2016 [2]. A substantial part of the video traffic is going to be VoD over 7 minutes and live traffic as shown in Figure 1.1. Moreover, with a share of more than 50% of the total Internet traffic, video traffic is without doubt one of the most important candidates when it comes to the optimization.

[1] Before the successful commercial era there were tools like vic [9] and a few streaming commercial companies that did not survive.

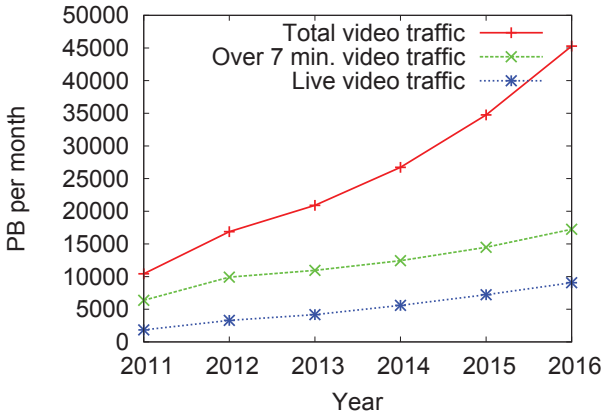


Figure 1.1: Cisco Visual Networking Index [2]: Internet video traffic forecast.

1.1 HTTP Segment Streaming

As the amount of video traffic has grown, the video streaming methods has been evolving. Historically, User Datagram Protocol [17] (UDP) was the protocol of choice for delivering live videos, and reliable transport protocols like Transport Control Protocol [18] (TCP) were used to deliver non-live VoD videos. This has changed and delivering live and on-demand videos over the Hypertext Transfer Protocol [19] (HTTP) on top of TCP has become very popular, e.g., used by Comoyo [16], Netflix [15], NRK [14]. The main reasons are to be found in the Internet Service Provider (ISP) and company network policies rather than in the technical aspects. TCP is Network Address Translation (NAT) friendly, and additionally, the standard HTTP port 80 is allowed by most firewalls. From the technical perspective, it is the possibility of reusing the already deployed HTTP infrastructure that makes streaming over HTTP so attractive.

To provide the client with the possibility to adapt to varying network resources in HTTP segment streaming, a video is usually split into segments, and each of these segments is made available in different bitrates. Thus, the client can adapt to changing network conditions or resource availability by simply requesting video segments in bitrates that fit the current network conditions. Figure 1.2 captures the process. We see that a live stream is first captured and sent to an encoder. The encoder encodes the stream into multiple content-identical streams with different bitrates. These streams are then split into segments by a segmenter. The segments are ready for playout after segmentation and can be distributed via a CDN for example. We call this type of streaming *(adaptive) HTTP segment streaming*. It is also known as HTTP dynamic streaming [20], HTTP live streaming (HLS) [21], Smooth Streaming [22] and as MPEG Dynamic Adaptive Streaming over HTTP (MPEG-DASH) [23]. MPEG-DASH has recently been ratified for international standard by ISO/IEC's joint committee known as MPEG.

Even though the HTTP segment streaming is based on HTTP, the temporal dependence between video segments makes the network traffic different from a regular HTTP web traffic.

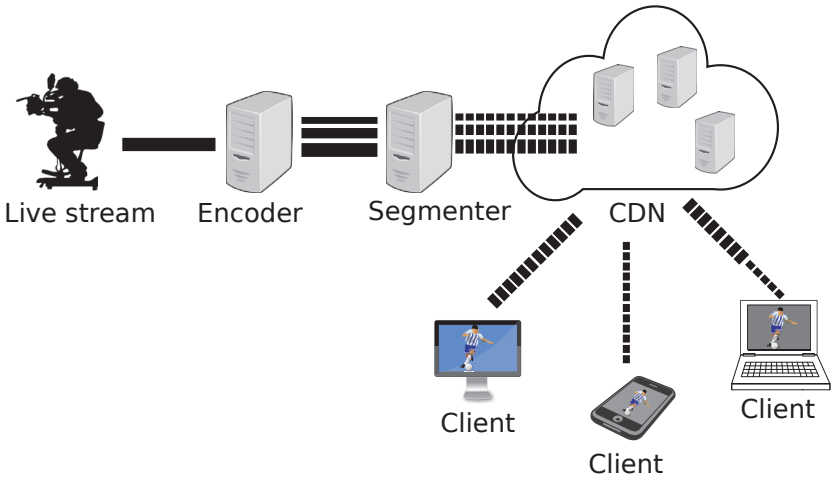


Figure 1.2: Adaptive HTTP segment streaming architecture

The server traffic generated by HTTP segment streaming clients is influenced by the temporal dependencies between segments and the segment availability in case of live streaming. This results into an *on-off* traffic pattern, i.e., traffic where sending of data is interrupted by idle periods. It is particularly different from the traffic generated by web clients on a web server. The understanding and optimization of the server traffic is of importance to streaming providers so that they can optimize their server and client software with respect to for example cost per client and quality per client.

1.2 Problem Description and Statement

The problem with the classical video streaming services like for example YouTube [10] is that the response to a search query is a list of complete videos. However, the user search query might match only a specific part of a video, yet the returned videos must be watched in their full length to manually find the moments of interest. However, the client driven segment streaming opens great opportunities to enhance user experience beyond basic progressive streaming as we know it from YouTube. In this thesis, we explore the possibilities that HTTP segment streaming brings in terms of playlist creation based on different videos.

We further look at the performance of delivery of segments as this is a very important aspect of a service as popular as the HTTP segment streaming. An HTTP segment streaming client downloads the segments of a stream one after another. The client chooses the bitrate of the segments according to the available bandwidth so that the time it takes to download a segment is shorter or equal to the actual segment duration (the playout time of a segment). The download time must be shorter or equal than the segment duration, because otherwise the client buffer would eventually become empty and pauses would occur in the playout. Since there is only a handful number of bitrates to choose from, the download time is usually shorter (not equal) than the segment duration. Therefore, it takes less time to download

a segment than it takes to playout a segment, i.e., the download speed is higher than the playout speed. The client buffer first hides this inequality by queueing every segment that is downloaded. However, after some time, it gets full and the download of the next segment needs to be postponed until a segment is consumed by the playout. This leads to an *on-off* download pattern as shown in Figure 1.3. This pattern is substantially different from a bulk data transfer. Furthermore, the regularity of the *on-off* pattern (especially in the case of live streaming where the segments become available in periodic intervals) differentiates this pattern from the web traffic pattern (download of HTML pages, pictures, CSS, javascript etc.).

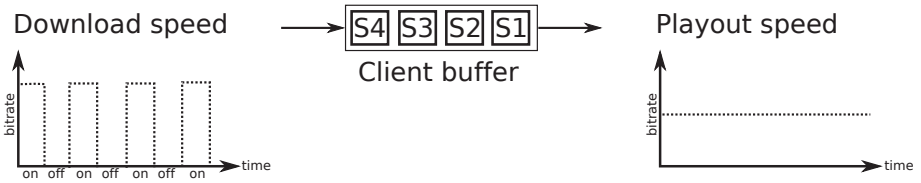


Figure 1.3: When the download speed is higher than the playout speed the client buffer becomes full eventually and the *on-off* traffic pattern arises.

When downloading a web page, a client requests first the HTML code of the page and then requests different web objects that are linked from that page like the javascript files, images, style sheets etc. It has been shown [24] that this type of web traffic for web browsing can be modeled fairly well by an *on-off* source. However, the distribution of the length of the on and off periods is different from the distribution of the on and off periods in case of HTTP segment streaming. The length of these periods is governed by the playout speed, which is constant, and the segment availability in case of live streaming. This results in almost constant on and off periods unless bandwidth or bitrate changes (the playout speed is usually fixed to a specific number of frames per second). Therefore, other traffic patterns are formed.

In case of live streaming, all clients have the incentive to be as "live" as possible, requesting a new segment as soon as it becomes available. Even though, in general, the client requests take different time to get to the server due to for example different RTTs, the incentive to be as live as possible leads to a nearly perfect request synchronization. This synchronization of the responses does not happen to web traffic. The challenge is to find ways to optimize the HTTP segment streaming *on-off* traffic. This requires looking at both the transport and the application layer.

Problem Scope and Limitations In the scope of this thesis, we first look at the application enhancements, which are possible with segmented streaming. In particular, we practically demonstrate the possibility of playlist creation based on segments from different original streams.

However, the main problem we explore in this thesis is the interplay of live segment streaming clients that share a common bottleneck. We do not restrict ourselves to single client

behaviour, but look at the system composed of clients and a server as a whole. We explore primarily how client-side modifications and very light, non-invasive server-side modifications both on the transport and the application layer can increase the number of clients that a server is able to handle with respect to quality, deadline misses and liveness.

We are aware of the advantages of segmented streaming, especially its infrastructure reuse, and therefore, we intentionally limit the scope of the modifications, to client and light server modifications, making most of our investigations relevant also to general CDN infrastructures where major server modifications might not be possible. For example, we do not investigate scenarios where the server keeps state about every client in order to be able to schedule segment delivery. We also do not assume any communication possibilities between the clients like in a P2P environment. This makes for a perfectly distributed problem, since the clients can not coordinate their actions with each other directly nor via a server.

We explore the following key areas in this thesis:

1. *The potential of HTTP segment streaming beyond classical streaming.* Here, we explore the potential application enhancements that are possible with HTTP segment streaming. We are specifically interested in the possibilities of combining segments from different videos into a seamless, smoothly playable playlists.

2. *Leverage multiple network interfaces on a client.* Today, many, especially mobile devices, have multiple network interfaces available, e.g., HSDPA [25] and WLAN. We therefore explore the possibilities to utilize all the available interfaces in order to achieve better user experience, i.e., video quality and liveness.

3. *Improving the performance of segment delivery from a congested server.* Because the delivery of the segments plays a major role by the scalability of HTTP segment streaming, we are interested in the way segments are delivered. We consider the *on-off* traffic pattern of HTTP segment streaming traffic and search for modifications on both the client- and the server-side to increase the performance of an HTTP segment streaming server.

1.3 Research Method

There are several ways to perform research in the area of computer science. In this respect, the Association for Computing Machinery (ACM) describes in [26] three major paradigms, or cultural styles, by which computer scientists approach their work:

1. The *theory paradigm* is rooted in mathematics. Objects of study are specified first. Then, a hypothesis about relationships between the objects is formed, and finally, the hypothesis is proven logically.

2. The *abstraction paradigm* is rooted in the experimental scientific method. Here, a hypothesis is formed first. A model and predictions based on the model are then made. As the third step, experiments are designed, data is collected and finally analyzed.

3. The *design paradigm* is rooted in engineering. Here, the requirements and specifications are stated first. Then, a system is designed and actually implemented. The last step is a system test to see if the stated requirements and specifications were satisfied.

This thesis follows mainly the abstraction and design paradigm. The research on video playlist composition (see Chapter 2) is based on the *design paradigm*. Our goal is to see the potential of HTTP segment streaming beyond linear playout and enrich and improve the user experience of video search with the features of HTTP segment streaming. We analyze existing services (like VG-Live [27] and TV2 Sumo [13]) providing video clips of small events in soccer games and talk to people from Microsoft to collect their requirements and specifications. Based on these, a design and a subsequent implementations matching the two different application domains are created. Besides technical testing, we create user surveys to see if the system actually significantly improves the situation or if we have to return to the design board. Moreover, our multilink solution approach to leverage a potential client-side link bottleneck (see Chapter 4) also follows the *design paradigm*. We want the client to make use of multiple interfaces at the same time, therefore we design and implement a solution that is able to do so. Through system testing in emulated, as well as in real networks, we propose a segment download scheduler.

Our approach to deal with the server-side bottleneck (see Chapter 5) falls rather into the *abstraction paradigm* than into the design paradigm, because it is practically hard (in real networks) to evaluate proposals that affect thousands of users. Our hypothesis is that certain client and server-side modifications will help to improve the effectiveness of an HTTP segment streaming server when the network bottleneck is at the server potentially serving thousands of clients simultaneously. Based on our hypothesis, we formulate a simulation and emulation model to verify our hypothesis. Unfortunately, we have no choice but to skip step three (the real world experiment) due to practical reasons, i.e., the main reason being that we are not able to get (in time) a streaming provider to evaluate our proposals in real networks with thousands of real clients.

1.4 Main Contributions

This thesis explores the potential of HTTP segment streaming beyond basic video streaming as well as proposes improvements to existing HTTP segment streaming techniques and settings. The main contributions to the challenges stated in Section 1.2 are summarized here:

1. *Proof of concept application enhancement implementing playlists based on segments from different videos.* Two prototypes from two different application domains (sports and education) were implemented to demonstrate the feasibility of segment combinations not only from the same video, but also across different videos. This means that the results to a search query no longer need to be complete videos, but can also be video segments from possibly different original videos. Furthermore, we measured the improvement in terms of user experience with user studies and found out that in both

cases a significant improvement was achieved. Users liked the added functionality and also thought it might be useful on a day-to-day basis.

2. *Increasing user experience with a multilink segment download scheduler.* We proposed and tested a method to divide video segments into smaller subsegments that can be retrieved in parallel over multiple interfaces. The proposed scheduler aggregates the throughput of multiple interfaces enabling the client to retrieve higher quality segments. We saw that our scheduler works not only in managed environment, i.e., emulated test networks, but also for real HSDPA [25] and WLAN aggregation. The result was a higher quality video stream compared to a single link scenario using either of the available interfaces.

3. *Proposed system improvements to HTTP segment streaming.* Our analysis of a real video streaming provider's log data shows that a lot of video segments must be served from sources other than the origin server. Since these sources are usually out of the control of the streaming provider, no major modifications can be done on the server-side of these, e.g., on HTTP proxy caches. Based on this knowledge, we proposed different possible modifications to both the server, but mainly to the client-side to optimize segment delivery from an HTTP server. Our exploration focused on transport layer modifications (TCP congestion control variant, CWND limitation) and application layer modifications (segment duration, segment request strategy). The benefits and drawbacks were evaluated by simulation and in some cases emulation. The global effects were evaluated in terms of increased client count per server, quality of downloaded segments, deadline misses and liveness (see Section 4.2). Additionally, an easy and fully distributed method for request de-synchronization was proposed. The main finding was that client requests synchronization should be prevented. If client request are distributed, the server performance increases, and the server can serve more clients with higher quality segments.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2, *"Beyond Basic Video Streaming with HTTP Segment Streaming"*, provides a detailed overview of HTTP segment streaming, especially the encoding of segments. It also discusses the work on video playlists and its application to different domains.

Chapter 3, *"Analysis of a Real-World HTTP Segment Streaming Case"*, shows the analysis of 24-hours logs provided by a Norwegian streaming provider Comoyo. It explains the data behind the indications of significant data delivery from sources other than the origin server.

Chapter 4, *"Improving the HTTP segment streaming with multilink"*, discusses the use of multiple links to deal with a client-side bottleneck. It is based on the observation that modern devices usually posses more than one network interface.

Chapter 5, ***"Enhancing the server performance",*** begins with the explanation of the parts of TCP relevant to HTTP segment streaming. It then proposes and evaluates different techniques and modifications to improve the effectiveness of a single server.

Chapter 6, ***"Conclusion",*** summarizes previous chapters and the work of this thesis and sheds light on perspective future work.

Chapter 2

Beyond Basic Video Streaming with HTTP Segment Streaming

The transmission of video has been around for many years. The first TV receivers became commercially available in the late 1920s, and since then the technology has boomed. The television was initially based on analog technology and used broadcasting, i.e., one-to-all communication, to reach its audience. However, since the 1920s, there has been a lot of progress not only in the area of video transmission, but also in the consumer needs and wishes.

The analog transmission for commercial, especially entertainment use, is being abandoned. In addition to moving from analog TV to digital TV, we see that video content and other related real-time content (subtitles, overlaid text, links, etc.) is being moved more and more to the Internet. In general, we call the delivery of video and other related real-time content over the Internet *media streaming*. Media streaming has raised many new challenges, not only due to the high processing requirements, but mainly due to the unreliability of the Internet links, making it hard to deliver video content in time.

There has been a lot of efforts (e.g., [28, 29]) to introduce Quality of Service (QoS) guarantees to the Internet. However, they have not been so far widely deployed and remain out of reach of a normal Internet user. Thus, for the regular user, the majority of Internet links and subsequently connections is still unpredictable in terms of available bandwidth (capacity), delay and loss, resulting in a best-effort service. These three metrics are not only unpredictable, but also vary over time for the same connection. The Internet media streaming must take the unpredictability into account in order to improve user's quality of experience.

2.1 The (not so) Historical Background

In the beginning of the Internet media streaming era, the usual way to deliver video over the Internet links was to use proprietary, (non-open) protocols like the Microsoft Media Server (MMS) protocol [30], Real Player's Progressive Networks (PNM/PNA) protocol, and Adobe's Real Time Messaging Protocol (RTMP) [31]. These were then largely replaced by the open standard Real-time Transport Protocol [32] (RTP)/Real Time Streaming Protocol [33] (RTSP) protocol suite. The RTSP protocol's responsibility is to negotiate a media streaming session parameters as well as to control the media streaming session (e.g., start,

pause and stop). The RTP protocol carries the video data and is usually encapsulated in a datagram protocol, which in practice means it is carried by UDP (the specification does not say it has to be UDP and it could as well be Datagram Congestion Control Protocol [34] (DCCP), Stream Control Transmission Protocol [35] (SCTP) or something else). In addition to RTP/RTSP, a protocol called RTP Control Protocol [32] (RTCP) is used to get feedback from the video receiver about its perception of the connection and the media player's state. Because the protocols have the full control over packet retransmission, they can be for example used for applications where packet loss is preferred over increased delay caused for example by TCP's congestion control, like video or audio conferencing. The protocol suite is very flexible, and the usage scenarios range from pure audio conferences to multicast multiparty low delay video sessions.

However, the flexibility of packet level control brings also disadvantages in practical implementations. Systems like these become quickly very complex because they have to deal with flow and congestion control, packet loss and out-of-order packets themselves. One other disadvantage of RTP/RTSP streaming is that the server has to keep track of the state of every streaming session. It needs to know whether the stream is paused or not, which ports to send different media tracks to, which RTP streams belong to which session etc. This implies that specialized RTP/RTSP servers must be used, which might be costly. Additionally, UDP traffic is often not permitted by default firewall and NAT settings, which makes RTP/RTSP deployment a rather complicated and challenging task. For example, a VG Nett (one of the largest online newspapers in Norway) log analysis showed that only 34% of UDP streaming attempts were successful [36], and the rest was served by TCP over MMS and HTTP progressive download.

On the other hand, the deployment and scaling of HTTP servers is easy and well understood. The default firewall and NAT settings pose no problems in most cases. Therefore, it is a very straightforward approach to try to deliver video over HTTP. The video file is uploaded to a regular HTTP web server no differently than any other static web object (web page, picture, etc.). The client then accesses the HTTP web server and downloads the media file. The difference to traditional static web object download is that the web server can limit (pace) the download speed of the client. In the perfect case, the reduced download speed perfectly matches the playback speed (in praxis the download should always be by a safe margin ahead of the playback to accommodate for jitter created by TCP and the Internet link [37]). Having the download only slightly ahead of the playback reduces the number of unwatched yet downloaded video content in terms of bytes if the user decides to stop the playback before the end of the video. In this way, the video provider saves link capacity that can be used by other users. This type of paced download is called progressive download and is very popular for VoD.

One of the main technical differences between progressive download and RTP/RTSP is the use of TCP under HTTP. TCP is not optimized for real-time traffic per se. Clients need to either buffer more data to compensate for jitter caused by TCP's congestion control or have a connection speed substantially faster than the playback rate [37]. However, even so, the practical benefits of HTTP deployment and scalability seem to overweight the advantages of

RTP/RTSP, which is confirmed by the large scale deployment of HTTP/TCP based solutions[1].

2.2 Video Segmentation

One possible way to adapt to available bitrate is to facilitate encoding techniques that were designed with scalability in mind. Examples of such encoding techniques are Multiple Description Coding [38] (MDC), Scalable Video Coding [39] (SVC) and Scalable MPEG [40] (SPEG). MDC uses a technique of fragmenting a single stream into a set of substreams. Any arbitrary subset of the substreams can be used to decode and watch the content but the more substreams are decoded, the better the quality of the video. However, the high fault tolerance is payed with a big overhead [41]. SVC is similar to MDC. It uses layered encoding where each layer N can only be decoded if layer N-1 is also decoded. A simple example of SPEG or priority progress streaming [40] extends the currently available compression formats with priority dropping. Priority dropping is a technique where the server starts dropping less important data first when informed that the client buffer is about to underrun. This saves bandwidth and gives a chance to the client buffer to recover. In contrast to random dropping of data, priority dropping provides smoother degradation of quality.

However, the problem with most of these advanced encodings is that they are not supported on commercial devices. This disqualifies them from the use in practical implementations. Therefore, other "bitrate adaptation techniques", like the progressive download, based on traditional codes like H.264 [42] are used in practice. However, the problem with progressive download (like provided by, e.g., YouTube, Dailymotion and Metacafe) is that the quality of the media must be chosen (for example manually by the user) in the beginning of the download. Particularly, the quality can not be changed later on when the playout is in progress. This is a major drawback if the connection properties change during the playout. In other words, an improvement of the connection speed leads to lower video quality as would be possible and the worsening of the connection speed leads to playout hiccups and annoying rebuffering periods.

To fix this problem, it should be possible to adapt the bitrate of a video stream to the current network conditions while streaming. This is where video segmenting helps (perhaps first mentioned in a patent filed in 1999 [43]). A live stream (or a prerecorded stream) is encoded multiple times, each time with a different bitrate and thus different video quality. The encoding is done in such a way that each of the streams can be chopped into smaller self contained pieces, called segments, that can be played out on their own, i.e., a segment has no dependencies on other segments. The client downloads the video segment by segment. It chooses the bitrate of each segment so that it fits the currently observed connection properties (see Figure 2.7(a)). Figure 1.2 illustrates the process of distributing a video via adaptive HTTP segment streaming. In this figure, the encoder machine receives a live stream and produces 3 content-identical streams. Each of the 3 streams has a different bitrate. The segmenter splits each stream into segments that are distributed for example via standard CDN

[1] For interactive media streaming sessions, like conferencing, people are now talking about RTCWeb (`http://tools.ietf.org/wg/rtcweb/`), which again may use RTP over UDP, but this is beyond the scope of this thesis.

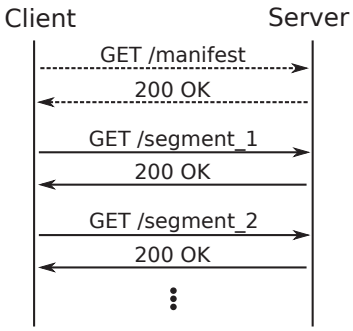


Figure 2.1: HTTP segment streaming timeline

infrastructure. A client can find out about which segments are available by downloading a file called a manifest file[2]. The manifest file contains the location of all available segments and is updated every time a new segment becomes available.

Additionally to the location of segments, the manifest file can also contain some metadata about the segments like the encoding parameters. After the manifest file is received, the client has enough information about where to find each segment and how to decode it as illustrated in Figure 2.1.

2.2.1 Commercial Systems

The adaptive HTTP segment streaming described in the previous section is implemented by many commercial vendors and here we present three examples from major companies. We describe only their main functionality here, for details please see the references.

Smooth Streaming

Smooth Streaming [22] is a streaming technology by Microsoft. The file format used is based on ISO/IEC 14496-12 ISO Base Media File Format [44]. The reason for choosing ISO/IEC 14496-12 ISO Base Media File Format is that it natively supports fragmentation. Actually, there are two formats, the disk file format and the wire file format. A video is recorded as a single file and stored to the disk using the file format shown in Figure 2.2. The *ftyp* box specifies that this is a media file [44] so that different applications can quickly get this information. The *moov* box includes file-level metadata that describes the file in general, e.g., how many media tracks there are. The media data itself is in the fragment boxes. They include metadata on the fragment level and the actual media data (*mdat*). The *mfra* box contains an fragment index with the video playout time each fragment contains and is consulted if random access into the file is needed. When a client (player) requests a video

[2]The concept of a manifest file works similarly to a BitTorrent tracker that keeps a list of peers that have a specific file.

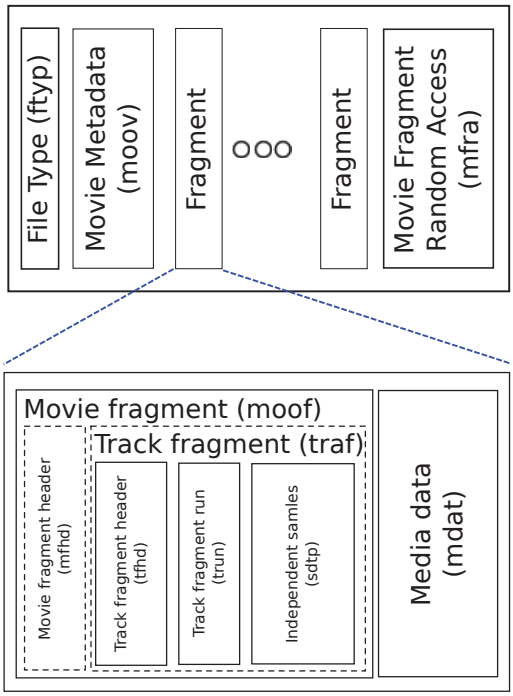


Figure 2.2: Smooth Streaming file format

segment (called fragment in [44]), the server seeks to the appropriate fragment and sends it to the client.

There are two file extensions that are used for the file format as described above, namely *.ismv* and *.isma*. The *.ismv* file contains video and optionally also audio. The *.isma* file contains only audio. In addition to the file containing the media data, there are two more files on the server. The *.ism* file describes the relationship between the media tracks, bitrates and files on the disk (there is a separate *.ismv* file for each bitrate). This file is only used by the server. The *.ismc* file is for the client. It describes the available streams (codec, bitrates, resolutions, fragments, etc.). It is the first file that the client requests to get information about the stream. Both *.ism* and *.ismc* files are based on XML.

The clients use a special URL structure to request a fragment. For example, a client uses the following URL to request a SOCCER stream fragment that begins 123456789 time units from the content start[3] in a 1,000,000 bit/s (1 Mbit/s) bitrate from comoyo.com:

```
http://comoyo.com/SOCCER.ism/QualityLevels(1000000)/
Fragments(video=123456789)
```

The Internet Information Services (IIS) server [45] with an installed Live Smooth Stream-

[3]The exact unit size is specified in the *.ismc* file, but is usually 100ns.

ing extension[4] knows the URL structure and parses out the information. It then looks up the corresponding video stream (in the *.ism* file) and extracts from it the requested fragment. The fragment is then sent to the client. Note that the fragments are cacheable since the request URLs of the same fragment (in the same bitrate) from two different clients look exactly the same, and as such, can be cached and delivered by the cache without consulting the origin server.

HTTP Dynamic Streaming

HTTP dynamic streaming [20] is a technology by Adobe Systems. The basis of this technology is similar to Microsoft's Smooth Streaming. The file format is based also on the ISO/IEC 14496-12 ISO Base Media File Format. However, there are a few differences. There exists three types of files (assumed the content is not protected by DRM). The *.f4m* file contains the manifest file. The manifest is XML based and includes information like the information needed to bootstrap, e.g., server serving the video content, (run) tables containing the mapping from a time point to the corresponding segment (see later) and fragment. The *.f4f* file is a file that contains the actual media data. The difference to Smooth Streaming is that there might be multiple of these files, called segments[5], for the same stream. The *.f4f* contains fragments as specified by [44] with a few Adobe extensions. For example, each fragment contains metadata that was originally encapsulated in the *moov* box that is originally present only once per [44] video file. The *.f4x* file is an Adobe extension that lists the fragment offsets that are needed to locate a fragment within a stream by the web server.

Since not all fragments of a media content are necessarily contained in just one *.f4f* file (segment) the URL addressing scheme is different from Smooth Streaming's addressing scheme. To request fragment 10 from segment 12 in HD quality from the SOCCER stream, the player issues the following request:

```
http://comoyo.com/SOCCER/1080pSeg12-Frag10
```

On the server-side, similarly to Smooth Streaming, a web server extension is used to parse the client's request and based on the obtained information extract and send the corresponding fragment from a *.f4f* file back to the client. For further details please refer to [20].

HTTP Live Streaming

HTTP live streaming [21] (HLS) is a technology by Apple. In contrast to Smooth Streaming and HTTP dynamic streaming, Apple does not use ISO/IEC 14496-12 ISO Base Media File Format [44] as the basis for its file format. HLS encodes each segment as a sequence of MPEG-2 Transport Stream [4] packets. We detail the encoding in the next section. Apart from the encoding container format, the main difference is that each segment is stored as a separate file on the server. For example, a 3600 second (1 hour) movie will consist of 360 segments, i.e., 360 files, if each segment contains 10 seconds of video (the recommended

[4] `http://www.iis.net/downloads/microsoft/smooth-streaming`

[5] Note that the term segment in this thesis means a fragment in HTTP dynamic streaming terminology.

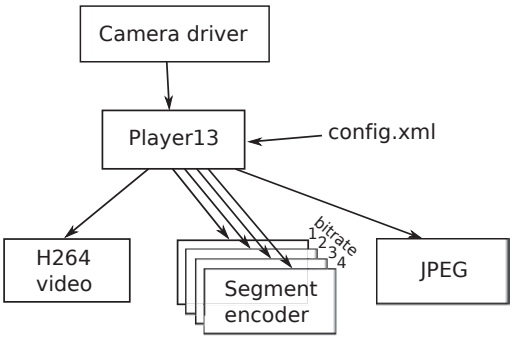


Figure 2.3: Architecture of Player13 encoder.

value)[6]. The advantage of having each segment as a separate file is that the server does not need to extract the segments from a continuous file and therefore no extensions are required on the web server. The drawback is that the distribution network, e.g., a CDN, needs to handle thousands of small files.

The manifest file is implemented as an extension to the M3U playlist format [46]. The playlists are text based and simply list the segments that are available. For all extensions and their meaning please refer to [21].

2.2.2 Segment Encoding

The video encoding process for adaptive HTTP segment streaming must take into account that the video will be segmented. The segments must be self contained so that they can be played out independently. The exact level of self containment depends very much on the technique used. For example, some segmentation techniques [22, 47] require the client to first download a meta file describing the decoding parameters common to all segments in the stream. Others [21] include this information in each segment.

In the scope of this thesis, we implemented a real-time segment encoder that is used to encode live video from cameras installed at a soccer stadium in Tromsø[7][8]. The encoder's architecture is sketched in Figure 2.3. The *Player13* [9] component receives raw frames from a network or an USB camera and distributes them to the other modules. The possible modules include a raw H.264 encoder, a JPEG encoder that stores each frame separately (further

[6]Smooth Streaming encodes each video bitrate as 1 file and for HTTP dynamic streaming the number of files is configurable.

[7]At the time of writing the encoder, we were not able to find a reliable encoder that would fulfill our two requirements. Firstly, it had to produce segments playable by both VLC and Apple devices. Secondly, it had to produce segments that are so self contained that segments from different live streams can be mixed.

[8]The video capturing is a first step in a video processing pipeline [48, 49].

[9]The soccer stadium project's ultimate goal is to automate football analytics, i.e., help the coach analyze the game. Since there are 11 players on the field and the audience is considered to be the 12th player, we decided that the 13th player contributing to the teams success is our software, hence the name.

Segment		TS	Program Association Table
		TS	Program Map Table
	Frame	TS	Adaptation field PES packet Data
		TS	Data
		... More data only TS packets ...	
		TS	Adaptation field Data
	Frame	...	

Table 2.1: Transport stream layout of an adaptive HTTP segment streaming segment (TS is the Transport Stream Packet [4]).

used by 3D reconstruction software) and an HTTP segment encoder. The configuration file *config.xml* supplied to *Player13* specifies the desired number and type of bitrates that should be produced. For each bitrate, a new HTTP segment encoder thread is spawned (Figure 2.3 shows four HTTP segment encoders with bitrates 1, 2, 3 and 4 Mbit/s). When the streaming is started, the *Player13* component distributes the received frames to each module. It also provides the wall clock time to all HTTP segment encoder modules. The clock synchronization is important for the production of combinable segments as we explain later in this section.

The HTTP segment encoder encodes the received frame with the H.264 codec [42] (specifically libx264 [50] with the *zero latency* settings). The codec is set to produce an intra-coded frame (I-Frame) every fps[10] $* \, segmentDuration$ frames. This ensures that every segment always starts with an I-Frame, which gives a random access point in the start of each segment. The Network Abstraction Layer [42] (NAL) units produced for each frame are wrapped into a H.222 [4] stream as shown in Table 2.1 (the NAL units are shown as data).

The Transport Stream packet (TS) is always 188 bytes long and starts with a 32 bit header. The header specifies the content of the packet and carries a continuity counter that is incremented for each packet. We placed in the beginning of each segment a Program Association Table [4] (PAT) that associates a program number to a Program Map Table [4] (PMT). PMT in turn specifies which elementary (audio/video) streams are associated with a program. We

[10]Frames per second (fps)

placed PMT always in the next packet after PAT (note that PMT and PAT are always padded and use a full TS packet). Having these tables in the beginning of the segment simplifies the parsing for the decoder since it does not need to search through the segment or get the tables from a different place. The third TS packet includes an adaptation field that specifies the Program Clock Reference [4] (PCR) value. The PCR value is the same for the same frame across different HTTP segment encoders (they use the same wall clock provided by *Player13*). This is important since a player later incorporates the frame into the playout buffer based on this value. Following the adaptation field is the Program Elementary Stream [4] (PES) packet specifying the Presentation Time Stamp (PTS) value, which is set to the same value as the PCR. The rest of the third packet is then filled with NAL units produced by the H.264 encoder separated by Access Unit Delimiter NAL [42] (AUD NAL) units. Normally, a completely encoded frame frame does not fit into the third TS packet and must be therefore distributed over multiple TS packets. The last TS packet of a frame that still contains data is padded with an artificially overblown adaptation field if needed. This process is repeated for each frame in a segment. We tested this setup successfully with the following players: MPlayer [51], FFplay [52], VLC [53] and QuickTime [21].

2.2.3 MPEG-2 Overhead when Segmentation is Employed

Even though segments produced by our setup are playable by many players, the tradeoff is the size inefficient MPEG-2 packaging of the segments (compared to fragmented MP4 [44, 47]).

Each segment must contain a PAT and a PMT table. Each of these tables is encapsulated in a 188 bytes TS packet (PAT_{pkt} and PMT_{pkt}). Additionally, an adaptation field and PES packet must be included for every frame in the segment. This means an additional cost of 8 bytes and 14 bytes (or 19 bytes if DTS timestamp is included) per frame, respectively. Lastly, the last TS packet of every frame needs to be padded to have a size of 188 bytes and, for frame data, a TS packet header of 4 bytes is included every 184 bytes. The overhead of the packaging format can be expressed as follows:

$$\frac{PAT_{pkt}+PMT_{pkt}+(adaptationfield+PES+lastTSpadding)*fps*segmentDuration+4*\frac{segmentDuration*bitrate}{184}}{segmentDuration*bitrate}$$

$$= \frac{PAT_{pkt}+PMT_{pkt}}{segmentDuration*bitrate} + \frac{(adaptationfield+PES+lastTSpadding)*fps}{bitrate} + \frac{4}{184}$$

We see that the first term converges to 0 as the segment duration or the bitrate increases. The second term converges to 0 only if the bitrate increases. The last term is constant and independent of the bitrate or the segment duration. It is the minimal theoretical packaging overhead of MPEG-2, 2.17% (Riiser et al. [47] shows how this number compares to other formats when both audio and video is included, e.g., Microsoft Smooth Streaming or Move Networks. He concludes that the overhead of MPEG-2 is considerably higher than the overhead of MPEG-4 [44]).

The duration of a segment has only influence on the weight of the PAT and the PMT tables (first term) in the overhead estimation. However, bitrate influences also the weight of the per frame overhead (second term). Figure 2.4 shows that the impact of the bitrate is considerable

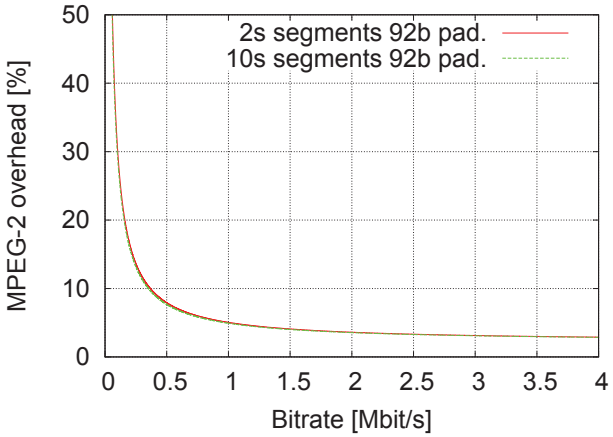


Figure 2.4: MPEG-2 packaging efficiency based on segment duration

and the impact of the duration is only minor. In Figure 2.4, the size of the padding in the last TS packet is assumed to be a random number of bytes between 0 and 184 bytes (the mean value of 92 bytes is used for the model).

Figure 2.5 shows the impact of the *last TS padding* variable on the overhead estimation. If the size of the encoded frame perfectly matches the TS packet size, i.e., no padding is required, the MPEG-2 container overhead is reduced considerably (see *2s segments 0b pad.* in Figure 2.5). However, if the last TS packet contains only 2 bytes of the data, the overhead is increased substantially especially for small bitrates (see *2s segments 182b. pad*). The plot also shows the overhead curve for perfectly constant bitrate encoding, i.e., constant (encoded) frame size (see *2s segments variable padding*). In this case, the padding depends on the bitrate and is computed by[11]:

$$lastTSpadding = 184 - [(\frac{bitrate}{fps} - 157)\ mod\ 184]$$

Our model implies that a subtle change in the bitrate can lead to a performance increase or decrease in terms of MPEG-2 segment packaging overhead. As such the implications should be considered when choosing a bitrate.

2.3 Segment Playlists

A video segment, as described in the previous section, constitutes an independent playable unit. Segments from the same video but of different quality can therefore be seamlessly combined to create the illusion of a continuous video stream. This is the way a player adapts the video quality (and bitrate) to the available bandwidth. A real trace of a VoD HTTP segment streaming session is shown in Figure 2.6. In addition to the changing segment bitrate, the time a segment was requested is plotted. The flat part at the beginning of the graph

[11] We first subtract the first packet size and then compute how many bytes will be in the last TS packet.

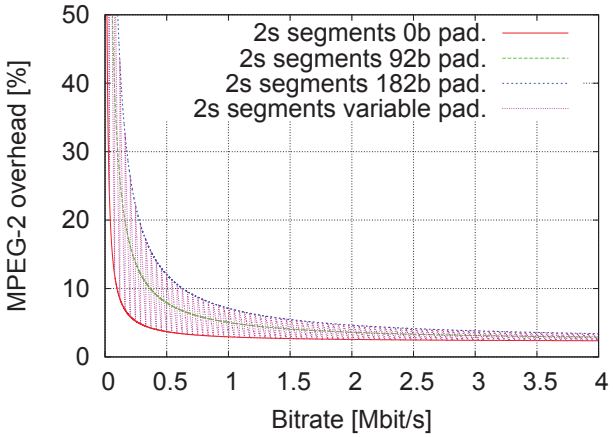


Figure 2.5: MPEG-2 packaging efficiency based on last TS packet padding size

indicates that the player requested multiple segments at about the same time. This is the time the client pre-buffered video segments in the beginning of the session. After this initial pre-buffering, a segment was requested approximately every segment duration.

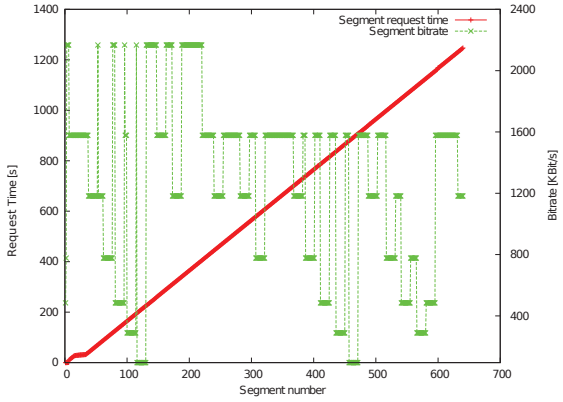
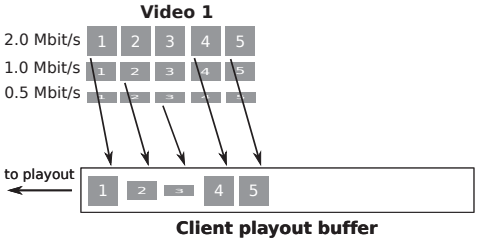
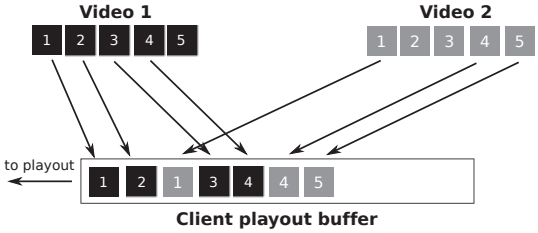


Figure 2.6: A trace of VoD streaming from www.comoyo.no (movie: "J. Edgar")

Moreover, segments from different video streams can be combined (if properly encoded) into customized videos. Both types of playlists, segments from the same video and segments from different videos, are illustrated in Figure 2.7. In the scope of this thesis, we implemented two systems that utilize the HTTP segment streaming playlist feature to enhance the user experience [54, 55, 56]. The creation of both of these systems was motivated by the fact that locating content in existing video archives like YouTube [10] is both a time and bandwidth



(a) Bitrate adaptation (segments come from the same video)



(b) Custom video stream (segments come from different videos)

Figure 2.7: Types of segment playlists

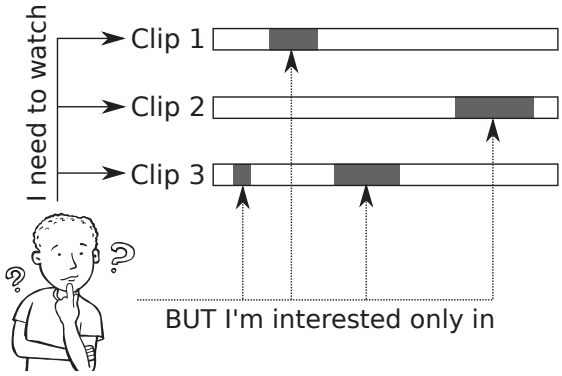


Figure 2.8: A user is only interested in certain parts of video clips that are relevant to his search query. Instead, the user must normally manually browse through complete videos to find the interesting information.

consuming process since the users might have to download and manually watch large portions of superfluous videos as illustrated in Figure 2.8.

2.3.1 Related Work

The possibility of combining segments from different videos into a continuous stream has a huge potential. However, to the best of our knowledge, there exists no service that exploit this feature as of now, at least in our target soccer and slide presentation application scenarios.

The main problem is that the commercial players can not seamlessly play segments from different videos without visible pauses or other artefacts during segment transitions. Therefore, the current popular video platforms like YouTube [10] offer playlist only in a form of playing multiple videos one after another with the player being restarted for every playlist item.

Moreover, none of the big video providers (e.g., YouTube and Dailymotion) provides fine grained video search. In other words, it is possible to search, for example, for the term "HTTP segment streaming", but the search results only include complete videos even though the search query would be answered only by a short part of the found video, e.g., a 1.5 hour lecture on streaming technologies where HTTP segment streaming is only mentioned for 10 minutes somewhere in the middle of the video may be returned and the user needs to manually find the interesting part.

The most featured system we could find in the slide presentation application scenario is the FXPAL's TalkMiner [57] and Altus vSearch [3] (Figure 2.9). FXPAL's TalkMiner returns decks of slides matching a search query that are each synchronized with video. Altus vSearch combines enterprise video search with PowerPoint synchronization and scrolling transcripts into an accessible and searchable video archive. It also allows the user to generate custom presentations for later reuse. However, both of these systems still miss the functionality to present a customized video for a selected set of slides, i.e., without manually going through each presentation.

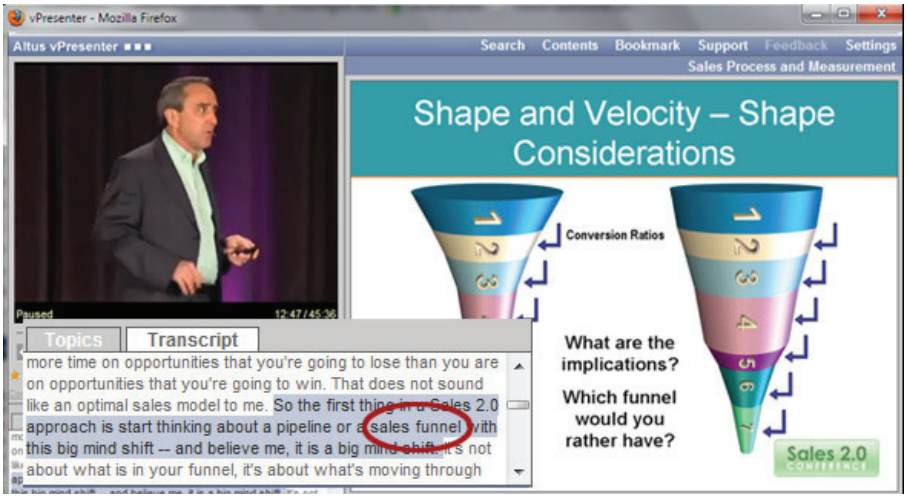


Figure 2.9: Altus vSearch [3]

In the area of video streaming applications for soccer (and sports in general), there are multiple available streaming solutions. Two examples are VG live [27] and TV2 Sumo [13].

These systems target the live real-time streaming over the Internet as an alternative to traditional television. They also provide limited search functionality where a user can search for games and some main game events, like goals. These systems, however, also lack the ability to compose customized, on-the-fly generated playlists of video.

2.3.2 vESP

General Idea Our first system prototype [54, 55] is based on the FAST enterprise search platform (ESP). It is called vESP (video ESP). The ESP itself is already a scalable, high-end enterprise search engine commercially used world-wide. In the next version, labeled FS14, improved contextual meta-data aggregation, more configurable relevancy models and support for in-page document browsing will give the users better and more relevant search results. In this context, our aim was to further enrich the search results with corresponding video data.

The example scenario was given by a large information collection from Microsoft giving searchable information about the Windows 7 operating system. This data set consists of Powerpoint presentations by both sales and technical personnel in Microsoft and corresponding videos that were recorded from the original presentation. Currently, FS14 is able to return a search result where the Powerpoint slides can be browsed within the search page, and alternatively another link (url) to the corresponding video, i.e., the user must manually browse each set of slides and seek in the video to find the interesting parts. The main idea of vESP is to align the video with the Powerpoint and then allow the user to select the interesting slides from possibly several presentations into a new, personalized slide deck (see the prototype user interface [12] in Figure 2.10). The relevant parts of corresponding videos are subsequently extracted and placed into a playlist as depicted in Figure 2.11, and both the slides and the aligned videos are played as one continuous presentation.

Architecture Figure 2.12 shows the architecture of our prototype. The inputs for the search index are besides the Powerpoint presentations also the transcripts of the speeches with the corresponding timing information. The videos of the presentations are stored in HTTP segment streaming format on a video server. When a users enters a query the search index of FS14 is searched and returns the relevant presentations. The user can then browse the in-page view of the returned slides and select those interesting to him/her. The selected slides are put on a slide deck, which can then be played back together with the video corresponding to each slide.

User experience evaluation The efficiency of the technical solution as well as the arbitrary composition of personalized videos was shown in [58]. For instance, a very short startup latency, smooth/hiccup-free playout of on-the-fly composed videos and seamless video quality adaption to available resources was demonstrated. Our focus in the scope of this thesis, besides the video integration into FS-14, was to investigate if the proposed integration of search and in-page document browsing within vESP adds value to the user experience. We therefore performed a user evaluation where the users evaluated three different versions of the system.

[12] For a demo, please visit `http://www.youtube.com/watch?v=fBj1UHljzoE`.

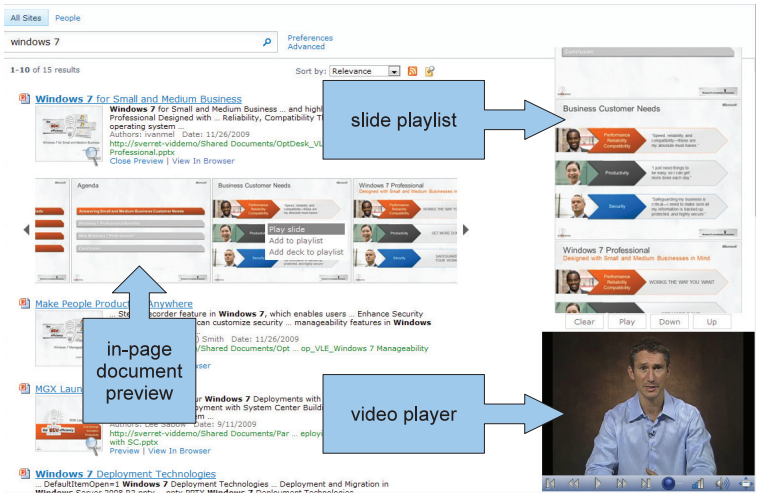


Figure 2.10: vESP user interface. The user can quickly browse through the slides of a presentation in the in-page document preview. Slides can be selected and added to the slide playlist, which can be afterwards played out.

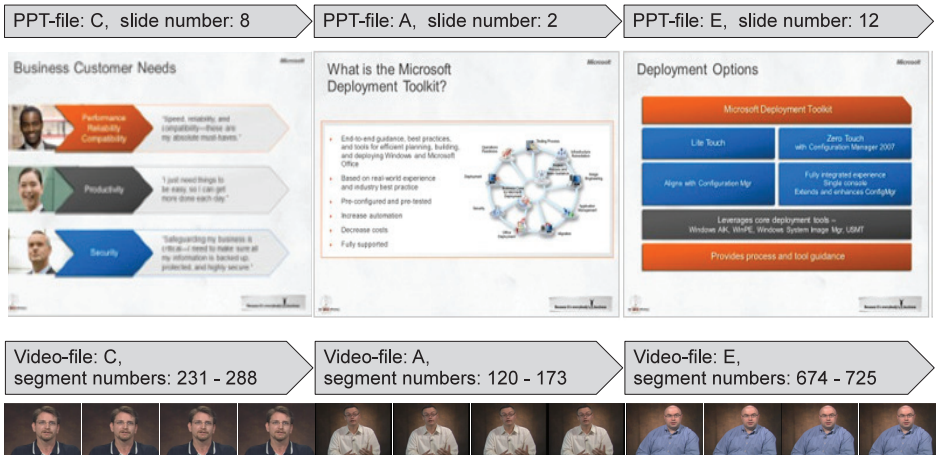


Figure 2.11: vESP slide playlist contains slides from different presentations. Each slide has an associated video consisting of multiple video segments. Even though the video segments come from different videos, our player is able to play them all in a seamless manner.

The first system presented search results using a textual, Google-like interface where returned documents had to be opened using an external program like PowerPoint. The second system added the possibility to browse slide thumbnails in-page (FS-14), and the third system (vESP) added support for presenting the corresponding video when the user clicked on a particular slide. For the experiments, we used the described data set with technical presentations from Microsoft. The users evaluated the features of the systems and after the subjective tests they

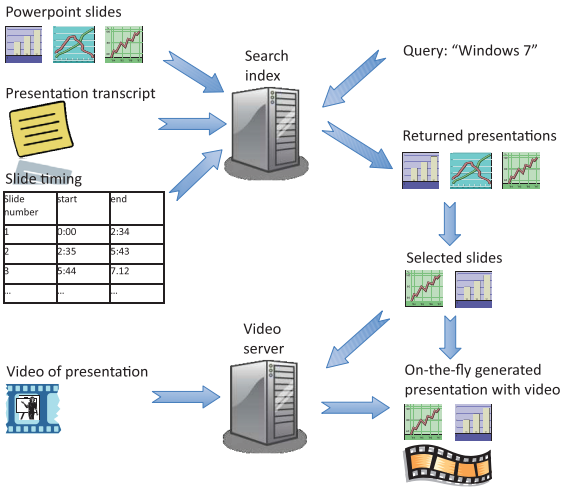


Figure 2.12: vESP architecture

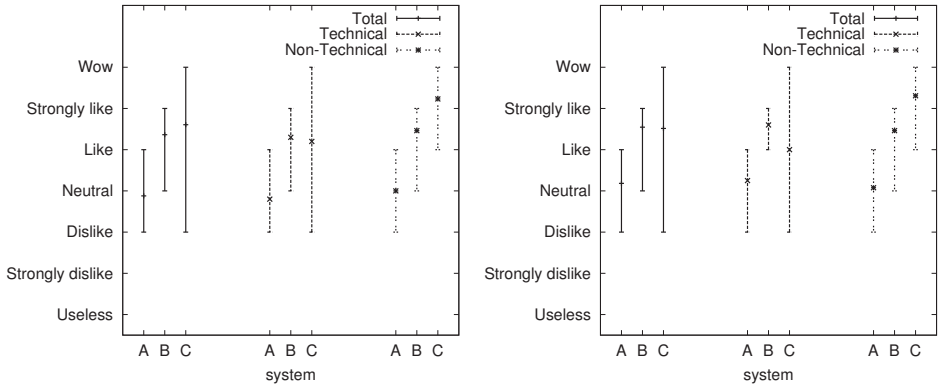
answered a short questionnaire about their impression of the systems and how they valued the functionality. The order in which assessors tested the different systems was changed to see if this affected the results.

For each system, we asked what the assessor's first impression of the system was and what the value of such a system was (if they would like to use it). Furthermore, we used a 7-point Likert scale with balanced keying to express user satisfaction. In total, we had 33 assessors, all familiar with and frequent users of search systems. Their job positions divided the group in two, i.e., technical (20) and non-technical (13) people. The results are presented in Figure 2.13. First, we did not observe any difference with respect to the order they tested the system. Then, we evaluated if the results are statistically significant (unlikely to have occurred by chance) using a non-parametric statistical Friedman test[13] to analyze the user data. Using a p-value (probability of the outcome) of 1%, both tests were significant regardless of group.

With respect to the user evaluations, we first grouped the whole assessor group together (labeled "total"). Then, we observed that the overall impression of the search result page was improved when adding in-page document preview and again slightly more when adding the features in vESP. For the system value test, the trend is almost the same, but the willingness is approximately the same to use the document preview system with and without video. In this respect, we observed that the degree of excitement for the added video feature value heavily depends on the background of the assessors and how they use search.

Non-technical people gave very positive feedback like "tremendous value" and "very impressive", and they would really like to use the added video feature. For this group, the deviation between the different assessors was also quite small. All scores were between "liked" (5) and "wow" (7). The technical people group, was slightly more reluctant. The scores varied more (larger gap between max and min, but also a larger standard deviation), i.e., between

[13] http://en.wikipedia.org/wiki/Friedman_test



(a) First impression of the search result page

(b) System value, willingness to use the system

Figure 2.13: User evaluation results with max/avg/min scores (A=plain, B=document preview, C=document preview with video)

"disliked" (3) to "wow" (7). The average is approximately as for the system with document preview. Some liked it a lot. However, they also had comments like "watching a video may take a too long time", even though some added "for me as a developer". A problem here seemed to be that some of the assessors did not see the feature as one that *could be* used, but one that must be used all the time. Nevertheless, in total, vESP was ranked higher than the other systems and was the only system given the top score "wow" – even in the group of technical people. 7 out of 33 in total gave vESP this highest score. The high ranking of the playlist feature makes HTTP segment streaming interesting for deployment even in a company network, i.e., possibly on a relatively small scale.

2.3.3 Davvi

General Idea In the second prototype implementation [56] of a system based on HTTP segment streaming, we focused on the soccer domain. We call the system Davvi [56]. The idea was to provide a flexible user friendly system that can deliver summaries of football events (e.g., goals or fouls). The interface of our prototype is shown in Figure 2.14[14]. The user first types the desired search query into the search box. Upon this the search/recommendation subsystem is queried and the matching events are presented to the user in a horizontally scrollable box. The user can drag the events he is interested in to the playlist box on the right-hand side of the screen. Another possible way to generate a playlist is to enter the query and select the desired playlist duration (length in minutes) with a slider at the top of the screen and hit the go button. The playlist is then automatically populated with the search engine results not longer in total than the specified duration. Because the event boundaries might not necessarily be perfect[15] each playlist event has a slider associated with it. The user

[14]For a demo please visit `http://www.youtube.com/watch?v=cPtvZ2kbt0w`

[15]A user can prefer a goal event A to start 10 seconds before the goal was actually scored whereas for the goal event B he might prefer 15 seconds instead.

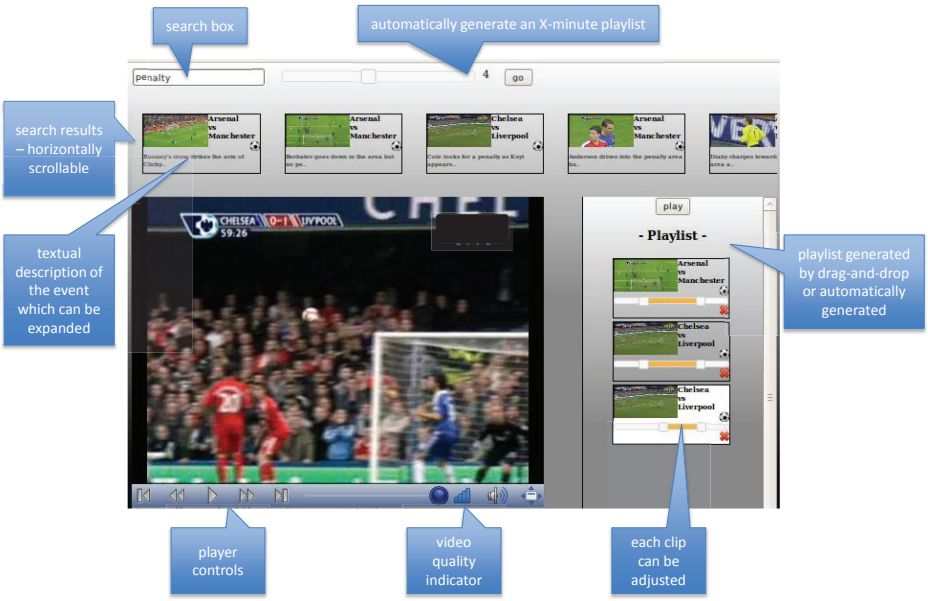


Figure 2.14: Soccer scenario user interface

can adjust the event's start and end time with the slider. Figure 2.15 shows a playlist and the information available for each entry. Furthermore, it shows how the playlist is actually composed of segments coming from different video files and zooms in into one of these segments to show that a segment always starts with an I-frame and represents a closed Group of Pictures (GOP).

Architecture The system is composed of three subsystems as shown in Figure 2.16. A live stream is captured by cameras installed at a soccer stadium. The stream is then encoded as described in the Section 2.2.2, and the segments are uploaded to regular HTTP servers. This is the streaming subsystem (number 2 in the figure). This and especially the user interface (described above) to the whole system is the contribution of this thesis in the scope of the soccer project. The two subsystems which were not developed as part of this thesis are the metadata subsystem and the search/recommendation subsystem. We describe them here briefly anyway to give the complete picture.

The metadata mining subsystem (number 3 in the figure) acquires metadata about the game. For example it crawls textual live commentary that is available in the Internet and performs video analysis on the captured stream. The output is a stream of events like tackle, save by the goalkeeper, foul, yellow/red card etc. annotated with the time of the event. The time annotated events are stored in a database (for detail on the search system please refer to [56]).

The event database serves as the knowledge base for the search and recommendation subsystem (number 1 in the figure). Given a textual query like "penalty", the subsystem queries

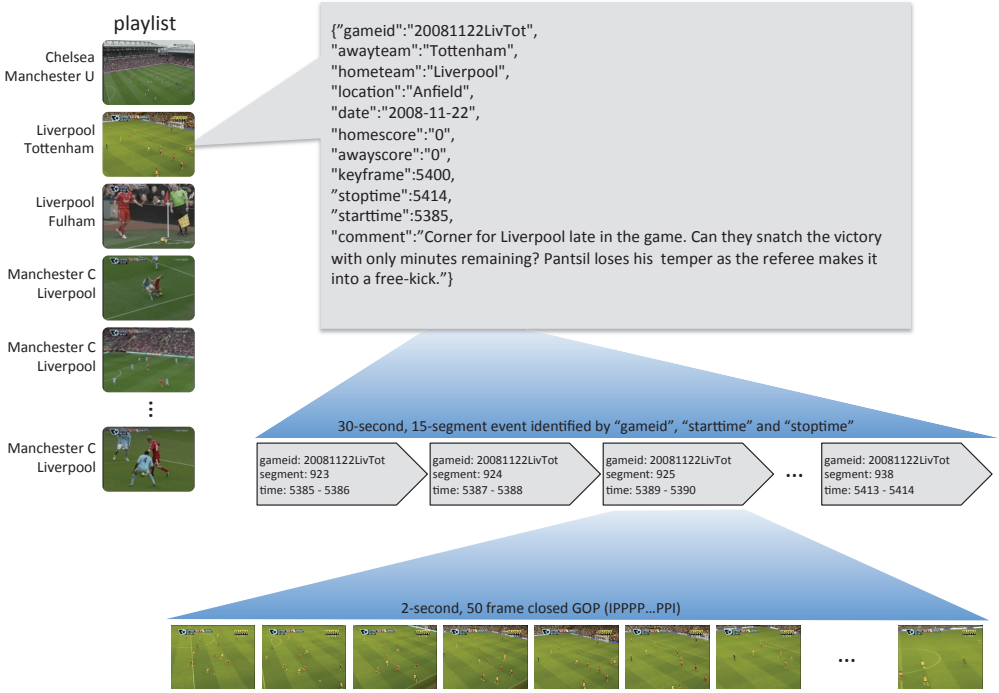


Figure 2.15: Playlist architecture

the event database and produces a list of matching events. A list item includes the name of the game and the corresponding time when the event happened within the game. In the scope of this thesis we developed a unified interface to the streaming and search/recommendation subsystem as described above.

We also added an extra layer between the search/recommendation subsystem and the user interface. This layer learns the new event boundaries based on how the users adjust the event start and end slider. For this, we used the following exponential moving average algorithm:

$$start_{new} = \alpha * start_{old} + (1 - \alpha) * start_{slider},\ 0 < \alpha < 1$$
$$end_{new} = \beta * end_{old} + (1 - \beta) * end_{slider},\ 0 < \beta < 1$$

The α and β parameters can be tuned based on the providers preferences. A value close to 1 will make the system very dynamic, on the other hand a value close to 0 will make the system very conservative. Additionally, users can be categorized into groups with similar preferences and the event boundaries can be adjusted for each group independently based on group specific parameters α and β.

User Experience Evaluation To evaluate Davvi in the soccer domain, we used 20 assessors (both male and female, age 22–45, with and without interest for soccer) in a controlled experiment. The assessors were exposed to both Davvi and a baseline system, the popular

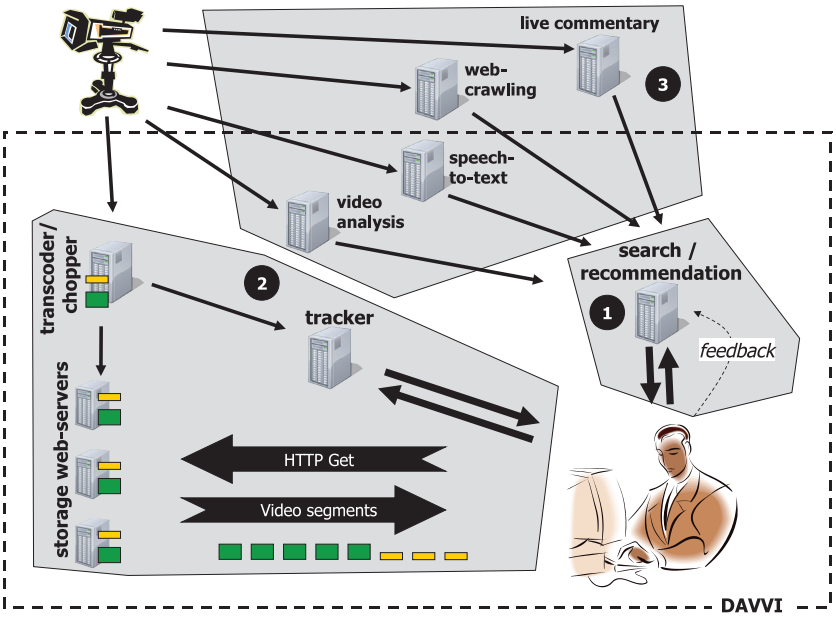


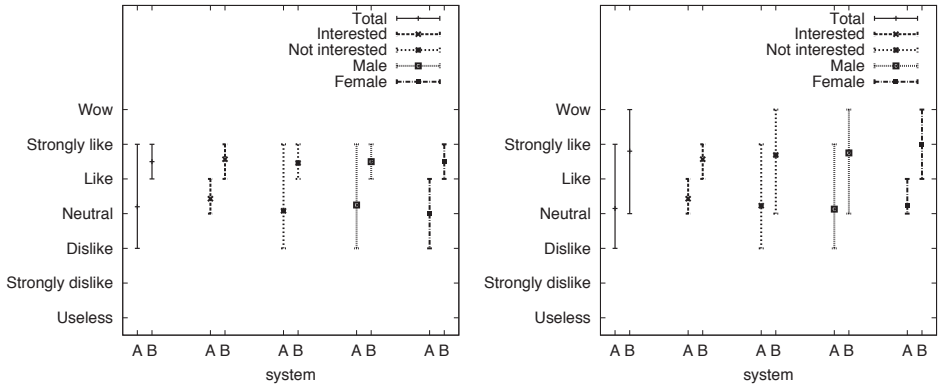
Figure 2.16: Davvi soccer user interface. A live football game stream is distributed by system 2, which is an HTTP segment streaming solution. System 3 gathers metadata about the happenings in the game from different information sources. System 2 uses this data to generate recommendations (or playlists) for a given user query. The user is able to influence future system 1 recommendations by supplying feedback on the current recommendations.

commercial portal VG Live operated by Schibsted. VG Live provides a real-time soccer streaming service, but also an archive of previously streamed games, manually produced highlights, a coarse-granularity search service, statistics and the like.

The results of the user study are presented in Figure 2.17, and the experiments show no major difference between order, gender, age, or scenario interest. Furthermore, the statistical significance was analyzed using a standard Wilcoxon signed-rank test[16]. The results are statistically significant since the probabilities that the results happened by chance are 1.335E-05 for system value and 2.670E-05 for first impression.

The overall trend shown in Figure 2.17 is that the users immediately appreciated the added functionality Davvi provides. Davvi on average scored 5.5, while VG Live scored more neutral 4.2. Even better results were obtained when asked about longer term value of this functionality. The average value of Davvi increased to 5.8, compared to an average of 4.15 for VG Live. The assessors typically liked the easy way of finding particular events and the way one could combine results on-the-fly into a playlist. They commented that this saves time compared to the benchmark system, and Davvi was the only system that was rated with the highest score ("wow"). This is more than an indication that these users would use Davvi for soccer event searching and summarizations on a more regular basis. Furthermore, the Davvi streaming component demonstrates the flexibility of the HTTP segment streaming.

[16]http://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test



(a) First impression of the search result page

(b) System value, willingness to use the system

Figure 2.17: User evaluation results with max/avg/min scores (A=VGLive, B=Davvi)

2.4 Conclusions

This chapter discussed the reasons that are behind the success of HTTP segment streaming, which is based on reliable transport protocols as opposed to the classical streaming, which is based on unreliable transport protocols. Furthermore, it showed how to construct segments for HTTP segment streaming. In particular, segments based on MPEG-2 transport stream [4] were discussed in detail, and a model for the packaging overhead was presented. The model states that the overhead of the MPEG-2 container mainly depends on the bitrate at which segments are encoded, which in turn defines the amount of padding that is needed for each frame.

We furthermore described two working prototypes, one from a business education domain [54] and one from a sports domain [56], that use HTTP segment streaming to deliver customized video playlists. Note that the way how different videos are chosen to be in a playlist is out of the scope of this thesis. The demonstration of playlist composition based on segments from different videos is the main contribution. We had additionally shown for both domains with user studies that the users like and value the playlist enhancements.

In summary, we believe that HTTP segment streaming has a lot of potential, both with respect to scalable delivery of current/traditional video services, but also for composing new services. In this respect, server-side scalability is a key issue, and in the next chapter we look at a day in a real world streaming provider's life where scalability is a major concern.

Chapter 3

Analysis of a Real-World HTTP Segment Streaming Case

In the previous chapter, we showed the basic features and some of the potentials of adaptive HTTP segment streaming solutions. An important question then is how such systems perform and scale to large numbers of users. In this chapter, we therefore present the analysis of logging data provided by a popular Norwegian streaming provider Comoyo [16]. Comoyo is serving both on-demand and live content. The on-demand service is available for customers that want to rent a movie from an extensive movie database. The rented movie is accessible for 24 hours[1] and within this time can be viewed unlimited number of times. The live service customers can watch the live transmission of the Norwegian Premier League [59]. The football[2] games are mostly played on Sundays and some on Wednesdays, and hence these two days are the busiest times for football streaming.

The network infrastructure for both live and on-demand services is the same (see Figure 3.1). The difference is only the source of the stream (it can be live or pre-recorded). There are two technologies used to stream the on-demand movies. These are the classic progressive streaming based on Windows Media Video (wmv) and Microsoft's Smooth Streaming [22] based on HTTP segment streaming. Microsoft's Smooth Streaming is also used for live streaming.

In the following, we focus on the Smooth Streaming, which we already described in Section 2.2.1. As Figure 3.1 shows, a number of IIS [45] servers with installed Live Smooth Streaming extension [60] is deployed behind a load balancer that distributes the incoming requests. The load balancer is connected directly to the Internet, i.e., ISP 1 in the figure. However, content is not served exclusively to subscribers of ISP 1, but also to subscribers of other ISPs as illustrated in the figure. Besides the IIS servers, an analytics server is deployed and connected to the Internet. This server collects various information from the video clients. For example, the clients notify the analytics server when the video playout is started, stopped or when the quality changes.

For our study, we were provided with two types of logs. We call these the server log and the client log based on where the information is collected and logged. For the study

[1] Comoyo is starting with a subscription service like Netflix in 2013.

[2] US readers: read soccer

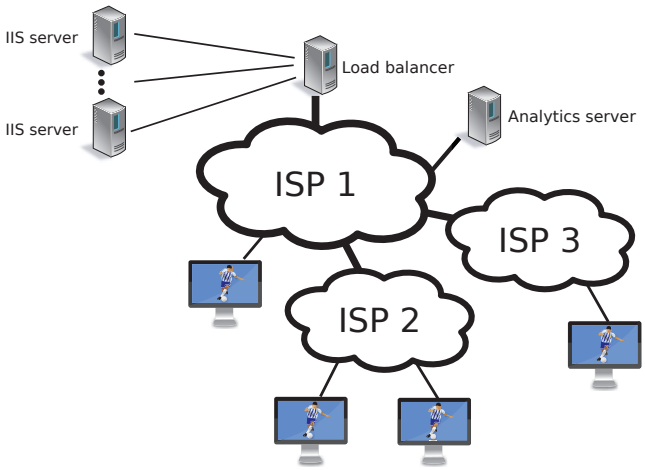


Figure 3.1: Streaming network infrastructure

in this chapter, we picked one server log and the corresponding client log from the same day. These two logs were found representative for the 3 days that we got data for. Both logs are 24-hour logs collected on 23.05.2012 when 8 games were played. The server log contains information about every segment request as received by the load balancer. The most important information are the time of the request, the request URL, client's Internet Protocol [61] (IP) address, response size and streaming type, i.e., the streaming type can be progressive streaming, on-demand Smooth Streaming or live Smooth Streaming. Table 3.1 lists all the other important information that is logged about each request. We analyse **live** Smooth Streaming only. We focus on live streaming because we want to study the scaling of HTTP segment streaming and live streaming has the most viewers watching the same content - it would be hard to study for example the caching of segments with on-demand streaming since the client per content ratio is very small (Comoyo's movie database consists of about 4000 movies).

The client log is a collection of events collected from the live Smooth Streaming clients by the analytics server while the clients are streaming. Collected events are described in Table 3.2. The information logged for every event is described in Table 3.3. Specifically, every client event includes the information to which user in which session and at what time the event happened. Note that within a session, a user can only stream one content, and a session can not be shared between multiple users. A typical sequence of events in one session is shown in Figure 3.2. In our analysis, we first analyse the logs separately and then point out the differences and inconsistencies.

3.1 Related Work

The popularity of HTTP segment streaming led to many real world studies for example, Müller et al. [62] collected data using different commercial streaming solutions as well as

Client IP	Client's IP address
HTTP method	HTTP method used (e.g., GET)
Request URL	The URL of the request
Response size	Response size in bytes
Useragent	Client's user agent
Cookie	Cookie information in the request
HTTP status	Status code of the response
Time taken	Time taken by the server in ms
Streaming type	The type of streaming. Possible values are progressive streaming, on-demand Smooth Streaming, live Smooth Streaming
Timestamp	Timestamp of the request in UNIX epoch format

Table 3.1: Information about every request in the server log file

play	The start of the playout
playing	Regularly sent when the playout is in progress
position	Sent when the user seeks to a different position in the video
pause	Sent when the user pauses the playout
bitrate	Sent when the player switches to a different bitrate, i.e., resutling in a different video quality
buffer	Sent when an buffer underrun occurs on the client-side
stop	Sent at the end of streaming
subtitles:on	Turn on subtitles
subtitles:off	Turn off subtitles
fullscreen:on	Switch to fullscreen
fullscreen:off	Quit fullscreen mode

Table 3.2: The reported client events

User device info.	This information includes the type of browser, OS, etc.
User id	Identification of the user
Content id	Identification of the content
Event id	See Table 3.2
Event timestamp	Time of the event in second resolution
Event data	E.g. for bitrate event the bitrate the player switched to
Session id	The session identification
Viewer	Always SilverlightPlayer v. 1.6.1.1
Client IP	Client's IP address as seen by the analytics server
Geographic info.	This information includes country, city etc.
ISP	Client's ISP

Table 3.3: Information about every client event

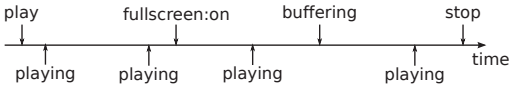


Figure 3.2: An example of events sent within a session

the emerging MPEG-DASH standard over 3G while driving a car. They compared the average bitrate, number of quality switches, the buffer level and the number of times the playout was paused because of re-buffering. A similar study was performed by Riiser et al. [63]. In their paper, they evaluated commercial players when on the move and streaming over 3G. They additionally proposed a location aware algorithm that pre-buffers segments if a network outage is expected on a certain part of a journey (based on previous historical records for that location). Houdaille et al. [64] focused on a fair sharing of network resources when multiple clients share the same home gateway. Akhshabi et al. [65] thoroughly compared rate adaption schemes of Smooth Streaming, Netflix, Adobe OSMF and their own AdapTech streaming player using synthetic workloads. Cicco et al. [66] analysed the behavior of Akamai's High Definition[3] video distribution from the client perspective and in [67] they proposed server-side rate control algorithm based on control feedback theory, however the evaluation was focused on the client-side (max. 2 clients were used for evaluation).

In general, studies, that we are aware of, focus all on the client-side. In other words, they measure the user experience in one form or an other from the client-side perspective. We believe this is mainly because the commercial content providers are protective of their data, and it is not easy to get any statistics or logs. However, the server-side performance is equally important in order to scale the service to 1000s of users. In this respect, we have a collaboration with a commercial streaming provider Comoyo [16], and we present the insights gained from both server- and client-side log analysis in this chapter.

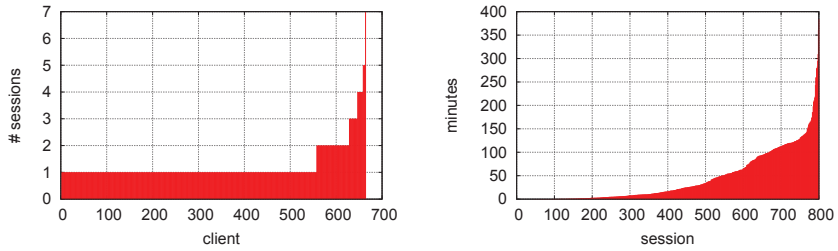
3.2 Server Log Analysis

Our analysis of the server log showed that there were 1328 unique client IP addresses in the log over the tracked 24-hour period. From all the IP addresses, 56% was tagged as live Smooth Streaming, 6% as on-demand Smooth Streaming and 44% as progressive streaming. 6% of all IP addresses was associated with more than one type of streaming, e.g., showed up one time as live Smooth Streaming and another time as progressive streaming. In other words, if each user had a unique IP address there would had been 748[4] people watching the live stream.

Sessions Since the IIS server is basically a "dumb" HTTP server, it does not keep state about an ongoing session. It therefore does not log the user identification, content id or the

[3]Akamai's High Definition is similar to segment streaming with the difference that it is the server that switches quality based on the client's feedback.

[4]This is a much smaller number compared to what is reported by the analytics server (see Section 3.3).



(a) Sessions per client IP

(b) Session duration

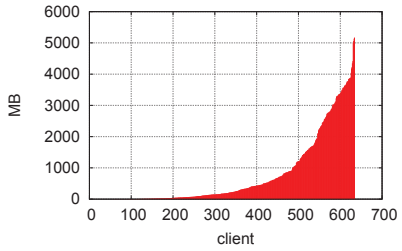
Figure 3.3: Sessions statistics based on the server log

session id (as is done by the analytics server in the client log, see Table 3.3). For this reason, we assumed (only for the server log) that a client is uniquely identified by its IP address, i.e., we assumed there is one client per IP address[5] (this proved to be a reasonable assumption, see Section 3.3).

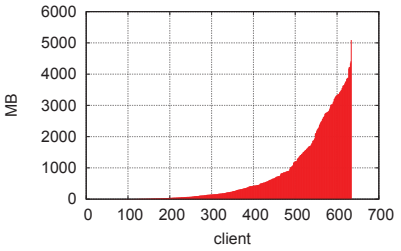
The content streamed can be extracted from the URL that is included in the server log, since the URL has a fixed format for Smooth Streaming as is described in Section 2.2.1. It is therefore possible to find out the content id, bitrate and the playout time in seconds of the segment within a video stream by parsing the URL. Unfortunately, it does not include the session id. We therefore approximated the session id with the combination of the client IP address (= client id) and the content id, e.g., if a client downloaded URLs with the content id *A* and *B* we say that the client had two sessions. Figure 3.3(a) shows that most of the clients had only one session. This is not what we find in the client logs as we will see later. It is also interesting to measure the live session duration. To do this, we calculated the time difference between the first and last request within a session. The session duration distribution is shown in Figure 3.3(b). The average session duration is 42 minutes (related to the break after 45 minutes in the game). Note that 107 clients are associated with multiple sessions, that is why there are more live sessions than live clients.

Byte transfers We noticed that some segments with the same playout time (included in URL, see Section 2.2.1) and within the same session were downloaded more than once, each time in a different bitrate. We conjecture this can happen because of two different reasons. First, multiple clients with different bandwidth limitations can be hiding behind the same IP (which can not be distinguished based on the available information). Second, the bitrate adaptation algorithm changes its decision and re-downloads the same segment in a different bitrate. Either way, bytes are unnecessarily downloaded, i.e., if there were multiple clients behind the same IP they could have downloaded the segment once and shared it via a cache; if the adaptation algorithm downloaded multiple bitrates of a segment it could have just downloaded the highest bitrate. We calculated the number of wasted bytes as the total number of bytes downloaded minus the bytes actually used. For this purpose, we assumed that only

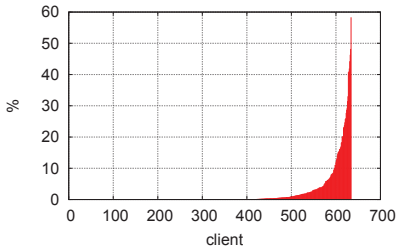
[5]We assume that, for example, there are no two clients sharing the same IP because they are behind the same NAT.



(a) Downloaded bytes



(b) Used bytes ($Downloaded - Wasted$)



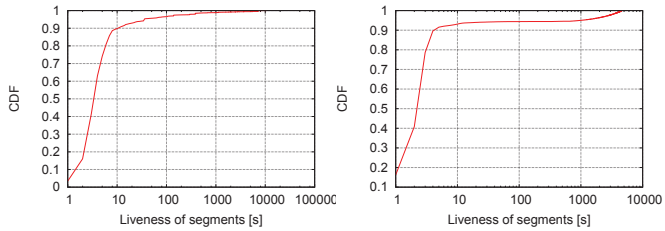
(c) Wasted bytes in percent of the total bytes downloaded

Figure 3.4: Per client bytes statistics

the highest bitrate segment for the same playout time within the same session is used by the client, i.e., if the same IP address downloads for the same playout time segments with 1 Mbit/s, 2 Mbit/s and 3 Mbit/s bitrate, only the 3 Mbit/s segment is played out and the other two segments are discarded (wasted). Based on the server log, approximately 13 GB out of the 467 GB downloaded in total were wasted this way. The 13 GB or 3% of the whole data could have been possibly saved if a different adaptation strategy or an HTTP streaming aware cache was used.

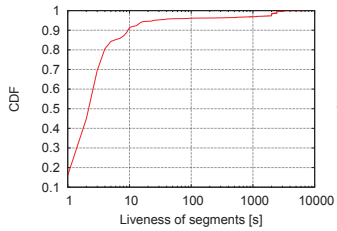
Figure 3.4 shows that there are clients that waste more than 50% of their downloaded bytes. Specifically, there is one client from Greece that contributes with 15.5% (2 GB) to the total amount of wasted bytes, even though its share on the total amount of downloaded bytes is only 0.85%. We think that this person might have recorded the stream in every quality, so there is also room for improvement on detecting and preventing behaviour like this (if stated in the service agreement).

Liveness Each log entry, i.e., request of segment i of a particular game, includes the timestamp when the server served that particular download request, T_i^d. It also includes the timestamp of the playout time of the video segment, T_i^p. Since we do not know the relationship between T_i^d and T_i^p (we just know that they increase with the same rate), we find the minimum of $T_i^p - T_i^d$ over all segments i for each game and assume that this is the minimal liveness (time the client is behind the live stream) for that game. In other words, the video segment

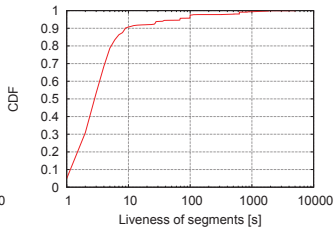


(a) Game 1

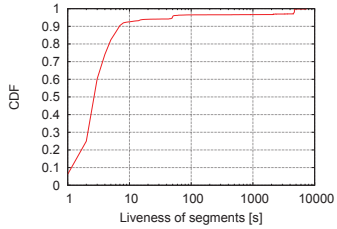
(b) Game 2



(c) Game 3



(d) Game 4



(e) Game 5

Figure 3.5: Liveness (absolute value) of segments based on the server log

with the minimal $T^p - T^d$ has liveness 0, i.e., is as live as possible. In this respect, we compute the liveness of all other segments in each game. The results are plotted in Figure 3.5 for the 5 most popular games. We see that about 90% of all segments in each game was less than 10 seconds behind the live stream (the liveness measured relatively to the most live segment as explained above). This means that 90% of the requests for a particular segment came within a 10 seconds period. This is interesting and calls for solutions to optimize concurrent segment access at the server-side (we look into this challenge in Chapter 5). There are also some video segments that were sent with quite some delay. One plausible explanation is that these are from people that joined the stream later and played it from the start, i.e., the catch-up window that is 2-hours long.

Number of IP addresses	6567
Number of users	6365
Number of sessions	20401 (or 3.21 per user on average)
Number of sessions with at least one buffer underrun	9495 (or 46% of total)
Number of sessions with at least one bitrate switch	20312 (or 99.5% of total)
Number of content ids	15
Number of countries	36
Number of cities	562
Number of ISPs	194
Number of users with multiple ISPs	113 (or 2% of all users)
Number of IPs with multiple users	31

Table 3.4: Statistics from the client log

3.3 Client Log Analysis

Every player that streams a live football game reports to the analytics server the events described in Table 3.3. The type of events reported is summarized in Table 3.2. The client log includes all reported events for a duration of 24-hours. The general statistics based on the client log are summarized in Table 3.4.

Client Location There were 6567 unique client IP addresses in the client log. This is significantly more than the corresponding 748 live streaming client addresses found in the corresponding server log. Not surprisingly, most of the IP addresses was located in Norway (Figure 3.6), but there were also IP addresses from almost all corners of the world (Figure 3.7). The international IP addresses correlated with the areas that are know for high Norwegian population like Spain and England.

Furthermore, the assumption in the server log analysis was that an IP address matches one user/client. Figure 3.8 shows that this assumption is quite reasonable. Only a very small number of users used more than one IP address, and an IP address was mostly used by only one user.

Furthermore, Figure 3.10 shows that most of the users used only one ISP (an IP address is hosted by one ISP only). Moreover, we saw that the majority of users used either Canal Digital Kabel TV AS (2560) or Telenor Norge AS (2454) as their ISPs. The other 1442 users were served by other providers (192 different network providers in total). We could see that the quality of the streaming in terms of the number of buffer underruns (periods the streaming is paused because of slow download) depends on the ISP, see Figure 3.9. For example, the majority of sessions at the Oslo Airport experienced buffer underruns (the airport provides a WiFi network).

Proxy Caching We hypothesise that the significant difference in the number of IP addresses in the server log and the client log is due to the use of cache proxies as known from traditional HTTP networks. The proxies reply to client requests before they get to the load balancer (Figure 3.1) and are logged. We expect the caches to be somehow related to ISPs, i.e., we

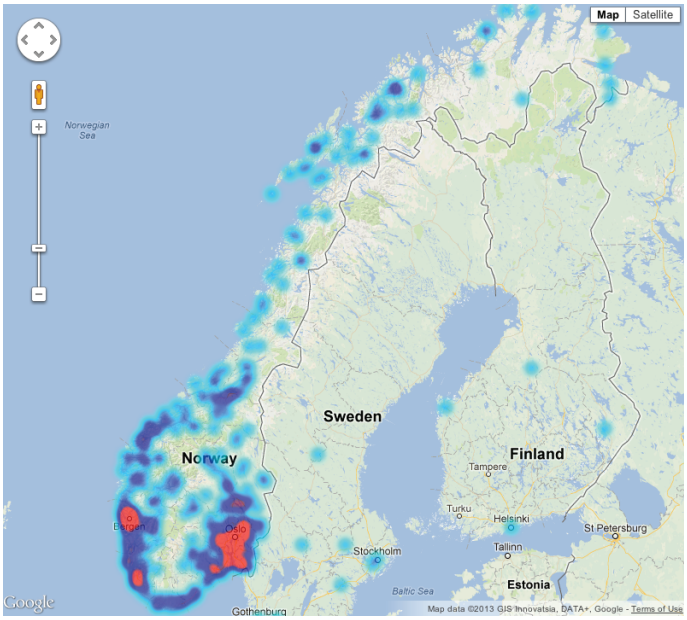


Figure 3.6: Geographical client distribution in Norway (the highest density of clients is in the red areas).

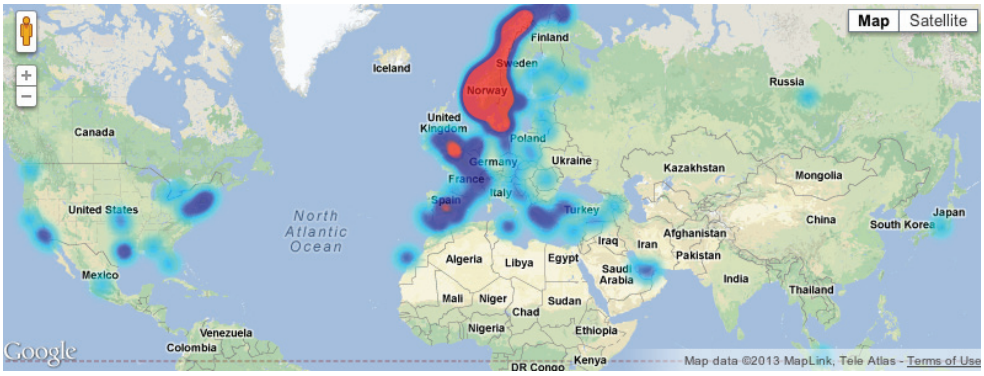
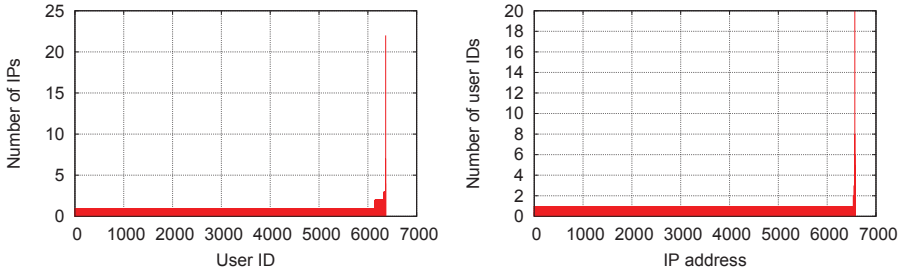


Figure 3.7: Geographical client distribution in the world (the highest density of clients is in the red areas).

would expect a cache on the ISP level to reply to all but the first request of the same segment, e.g., the open source proxy Squid [68] only sends the first request to the origin server all consecutive requests are served from the partially downloaded data in the cache.

Our hypothesis is supported by the IP to ISP ratio. The server log IP to ISP ratio for live streaming is 0.08 whereas the client log ratio is 0.03. In other words, we find 11% of the client log IP addresses in the server log, but find over 30% of ISPs from the client log in the server log (see Table 3.5 for exact numbers). We therefore hypothesise that HTTP caches located between the client and the server reply to a large number of client requests. This



(a) Number of IP addresses per user (b) Number of users per IP address

Figure 3.8: User to IP address mapping

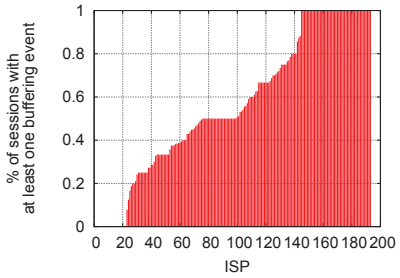
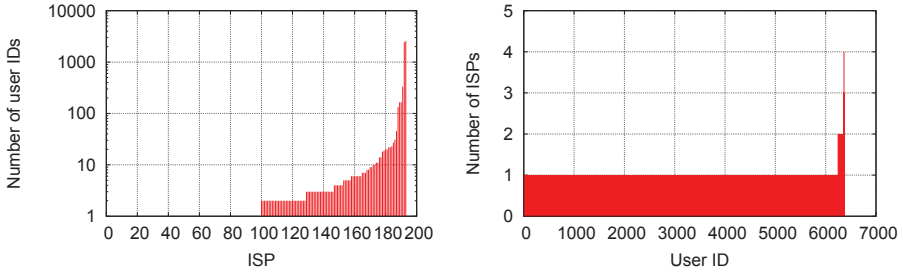


Figure 3.9: The percentage of sessions with at least one buffer underrun by ISP



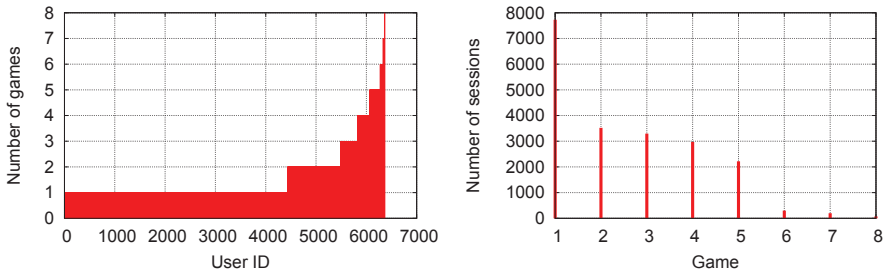
(a) Number of user IDs per ISP (b) Number of ISPs per user ID

Figure 3.10: ISP to user statistics

Type	**Server log**	**Match in client log**
live Smooth Streaming	748 IPs	723 IPs (or 97%) hosted by 58 ISPs
on-demand Smooth Streaming	74 IPs	5 IPs (or 7%) hosted by 3 ISPs
progressive streaming	581 IPs	390 IPs (or 67%) hosted by 32 ISPs

Table 3.5: IP to ISP statistics

means that, as expected, one of the large benefits of HTTP segment streaming is the reuse of existing HTTP infrastructure.



(a) Number of games streamed by a user

(b) Number of sessions per game (8 games were scheduled for that Sunday)

Figure 3.11: Content statistics

Content and Sessions On the content side, we observe in Figure 3.11 that only about 2000 users streamed more than one content, i.e., several football games. The most popular content was game 1 with over 7500 sessions followed by 4 games with about 2000 to about 4000 sessions.

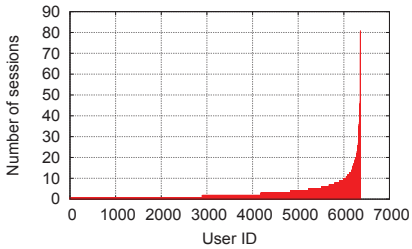
We also estimated the session durations based on the client log. This is not a straightforward task since only a small fraction of the sessions is started with a *play* event and terminated with a *stop* event (we suspect the user closes the browser window without first clicking on the stop button in the player). We therefore calculated the session duration as the time difference between the first and last *playing* event[6]. Session durations calculated this way are, in general, shorter than they really were because they do not include the time before the first *playing* event and the time after the last *playing* event. Note also that the session duration represents the time the player was playing. Particularly, it does not represent the time the player window was open, but rather the time the player was playing a stream, e.g., the time when the stream was paused is excluded. Figure 3.12(b) shows that the calculated session durations ranged from 100 seconds to more than 3 hours. Please note that 6826 sessions did not contain a *playing* event and therefore are not represented in this graph (e.g., they were too short for a *playing* event to be sent).

Figure 3.12(c) shows that in the majority of sessions the bitrate was switched at least 3 times. However, Figure 3.12(d) shows that even with bitrate (quality) switching, the clients were not able to prevent buffer underruns. The logs report that more than a half of the sessions experienced at least one buffer underrun.

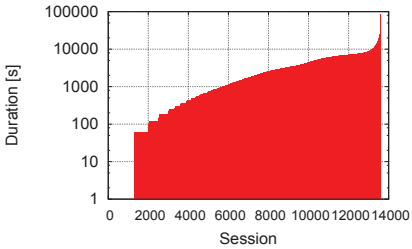
It is clear, from Figure 3.12(e), that the pressure on a streaming system like this is the biggest when a event begins. In this case all the games started at 5 PM UTC, and that is the time most of the clients joined. Figure 3.12(f) shows the number of active sessions over time.

Byte transfers We also estimated the number of downloaded bytes based on the client log. We were very conservative and our estimations were very likely smaller than what the reality was. We split every session into periods bordered by *playing* events and/or *bitrate* events so that there was no *pause*, *stop*, *position*, *play* or *buffer* (buffer underrun) event inside a period.

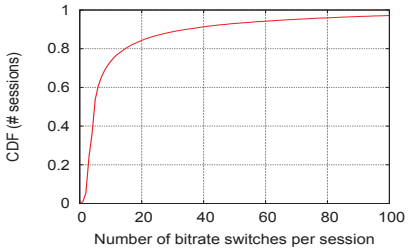
[6]The *playing* event is sent regularly when the live stream is playing.



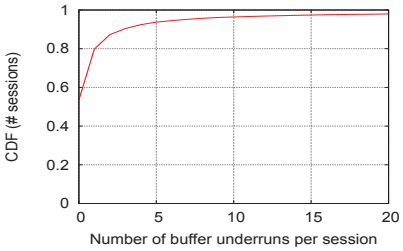
(a) Number of sessions per user



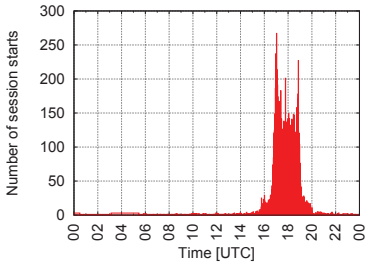
(b) Session duration based on the *playing* event



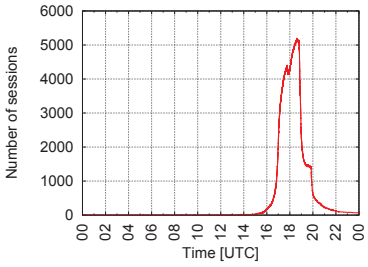
(c) Number of bitrate switches in a session



(d) Number of buffer underruns in a session



(e) Number of sessions that were started per minute (all games started at 17:00 UTC)

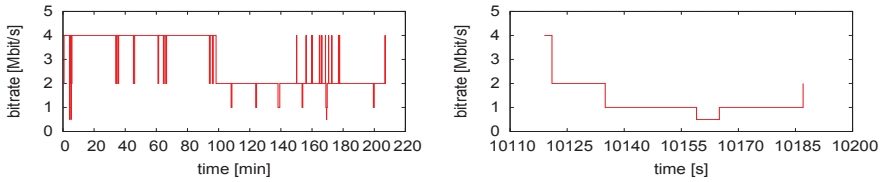


(f) Number of sessions at each point in time (minute resolution)

Figure 3.12: Session statistics based on the client log

This ensures that the player was really playing during a period of time defined like this (session examples with bitrate switching are shown in Figure 3.13). A bitrate (the video content was available in 0.5, 1, 2 and 4 Mbit/s, and the audio in 96 KBit/s) was additionally assigned to each period based on the client *bitrate* event reports. The period duration multiplied by the corresponding bitrate gave us a good estimate of the bytes downloaded by the client in that period since Constant BitRate (CBR) encoding is used for the live streaming. The sum of bytes received in each period of a session gave us the total number of bytes downloaded in a session.

It is interesting that the final sum of bytes received by all sessions is many times higher than the number of bytes served by the server. The server log reports 551 GB in total whereas



(a) Dynamic bandwidth session.

(b) Zoomed in to minutes 168 to 170.

Figure 3.13: Example of bitrate adaptation throughout a session based on client reports

the estimate based on the client log is about 4.5 times higher. Even if only the smallest available bitrate is assigned to every period, the total amount of data received by the clients is 2.5 times higher than the value from the server log. This is another evidence for the presence of HTTP caches that respond to a significant number of requests.

3.4 Conclusions

In this chapter, we analysed the logs of adaptive HTTP segment streaming provided by Comoyo [16], a Norwegian streaming provider. We analysed two types of logs; one from the origin server and one from the analytics server to which the clients report. The analysed data included all streaming sessions ranging from very static to very dynamic sessions, see example in Figure 3.13.

We observed that about 90% of the requests for the same segment in live streaming is sent within a period of 3 to 10 seconds depending on the content (analysed football game). This gives a great potential for caching. This also means that an HTTP proxy cache needs to cache a segment only a very short time and that most of the segment requests come during a very short time period.

We also deduced from the segments downloaded multiple times that at least 3% of the bytes downloaded could have been saved if more HTTP caches or a different bitrate adaptation strategy had been used. Moreover, based on the number of bytes downloaded and the IP address distribution in the server and the client log, we found evidence that segments must be delivered from other sources than from the examined servers. This suggests the presence of HTTP caching proxies in the Internet that had served the requests before they got to the load balancer. This means that the origin server is offloaded and that the idea of reusing the (unmodified) HTTP infrastructure really works in real scenarios.

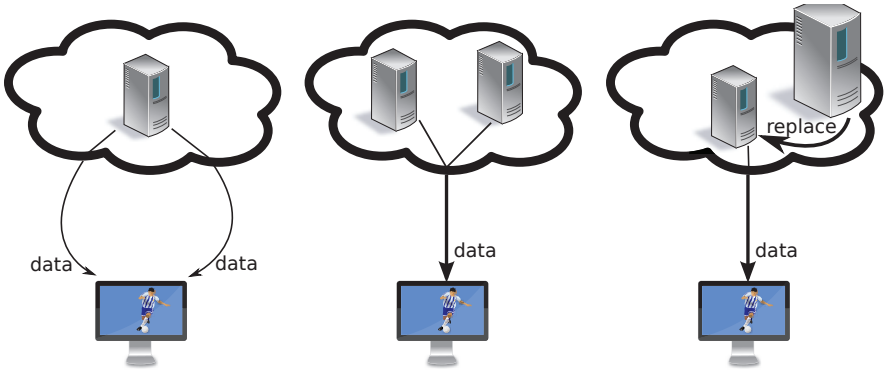
Furthermore, since a significant portion of the data is distributed by HTTP caches that are most likely not aware of what they are serving, we conclude that it is important to look at how HTTP streaming unaware server performance can be increased by client-side or very light server-side modifications. In other words, it is important to look at the performance of HTTP servers that actually deal with HTTP segment streaming traffic.

The increase of the performance of such a server is, of course, only interesting if there is some kind of bottleneck. We can eliminate the disk as being the bottleneck for live streaming as the segments are small and clients are interested only in the most recent segments of the

live stream. As such, the disk I/O poses no problem with current hardware. After the disk bottleneck elimination, we are left with the problem of a network bottleneck.

There are two types of bottlenecks - a client-side and a server-side bottleneck. The difference is that the client-side bottleneck can not be dealt with with a faster server, i.e., in this case the server has the capacity to serve higher bitrate segments, but the client link is not fast enough to receive them in time. The obvious solution is to throw money at the problem and upgrade the client link. In this thesis we use a different approach. We exploit the fact that more and more devices have multiple interfaces (Figure 3.14(a)) available and we propose an algorithm in Chapter 4 to use these effectively to increase the bitrate of the downloaded segments.

If the bottleneck is on the server-side, the server capacity needs to be upgraded to deal with the bottleneck. This can be either done by using multiple servers (Figure 3.14(b)) or by increasing the capacity of the server already deployed (Figure 3.14(c)). We look at how to increase the capacity of a server without upgrading its physical link (by using different client request strategy, changing the TCP congestion control etc.) in Chapter 5.



(a) Use of multiple links to deal with a client bottleneck.

(b) Use of multiple servers to deal with a server bottleneck.

(c) Use of a more powerful server to deal with a server bottleneck.

Figure 3.14: Types of throughput optimizations for different types of bottlenecks.

Chapter 4

Improving the HTTP Segment Streaming with Multilink

A lot of research assumes that it is often the case that it is the client link that does not have enough bandwidth to stream the best possible bitrate available, especially in wireless environments. We build on the observation that streaming video in wireless environments is often a subject to frequent periods of rebuffering and is characterized by low video quality. Furthermore, we observe that increasingly more wireless devices have more than one built-in network interface, e.g., WLAN and 3G. If client's interfaces are connected to independent networks, the simultaneous use of multiple interfaces can achieve a total transfer bit rate close to the sum of all the individual interfaces' throughput. In our work [69] we investigated the use of such interfaces for HTTP segment streaming[1].

Using parallel connections is a commonly used technique for improving the performance of content distribution systems as well as multimedia streaming. Parallel access schemes, like those described in [70, 71], shift the load automatically from the congested servers to the less utilized ones. However, these parallel schemes lack the notion of deadlines when the video segments must be available on the client for a quasi-live playout. Additionally, the quality adaptation of HTTP segment streaming is not taken into account by these schemas. In this chapter, we investigate how to use multiple interfaces to download video segments faster and in higher quality.

4.1 Parallel Scheduling Algorithm

The core of every parallel download scheme is the parallel scheduling algorithm. The parallel scheduling algorithm decides which portion of data is sent over each link or in our case interface. As part of this thesis, we designed and evaluated a parallel scheduling algorithm that is suitable for live HTTP segment streaming over multiple interfaces. In our approach we first divide each segment into smaller virtual pieces, so called partial segments, see Figure 4.1. Partial segments are the smallest data units our algorithm works with. We use 100 Kbytes partial segments, which is considered to be the optimal size [72]. The small size of the partial

[1] This chapter is based on the paper [69].

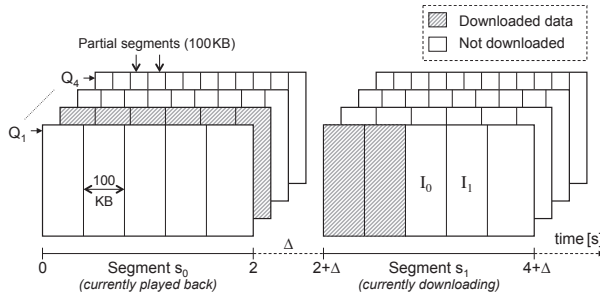


Figure 4.1: Segment division into partial segments for delivery over multiple interfaces

Algorithm 1 Request Scheduler *[simplified]*

```
 1: quality_level = "low"
 2: request(initial partial segment over each interface)
 3: request("pipelined" partial segment over each interface)
 4: while (more data available from server) do
 5:    data = receive()
 6:    I = interface that received data
 7:    estimate aggregated throughput
 8:    if (data == complete partial segment) then
 9:       if (data == complete segment) then
10:          queue_for_playback(segment)
11:          adjust quality_level to throughput
12:       end if
13:       request(next partial segment, I, quality_level);
14:    end if
15: end while
```

segments makes fine-grained scheduling decisions possible. Our scheduler, Algorithm 1, schedules the download of each partial segment over one of the available interfaces[2].

Initially (lines 2 and 3), two partial segments are requested on each available interface. One of these requests is immediately answered by the server the other is pipelined. The pipelining of requests ensures that the server always has data to sent, i.e., the link never becomes idle due to the server waiting for the requests from the client to arrive. After a partial segment is completely downloaded another partial segment request is pipelined on the same interface (line 13). If the downloaded partial segment completes the download of a segment the aggregated throughput of the available interfaces is estimated (line 10), video quality of the next segment is chosen accordingly[3] (line 11) and the first partial segment of the next segment is requested (line 13).

[2]The partial segments are virtual units mapped to physical units, the segments, on the server's disk. The client uses an HTTP/1.1 range request with the partial segment's beginning and end offset within a segment to get it from the server.

[3]Before a new segment is requested, the scheduler fetches the file size of the different quality levels using the HTTP HEAD method. The quality level of the requested segment is decided based on the number of bytes the client can receive within a segment duration (calculated using the average throughput).

4.2 HTTP Segment Streaming Metrics

Before diving into the results section, we introduce performance metrics that we used to measure the performance of HTTP segment streaming throughout the thesis. Many of these are already known from other types of streaming. However, even the familiar metrics have sometimes many names. For example hiccups, buffer underruns, deadline misses, stalls are all used to describe the same metric, i.e., the event when the player does not have data to play and the user is presented with a ”frozen” screen or another replacement for the actual content. In this respect, we define in this section the metrics that are used throughout this thesis with their respective names.

Client Buffer Size The client buffer size is the number of segments that the video player can store locally.

Start Up Delay The start up delay is the time that elapses between the click on the play button in a player until the first frame is shown. The duration of the start up delay directly depends on the available bandwidth and round trip time to the server. It also depends on the pre-buffering that is required before the playout is started (see Figure 4.2). So, for example, the start up delay is smaller if only 2 segments are pre-buffered instead of 4 segments of the same size, yet the exact time depends on the bandwidth and round trip time to the server. This metric is sometimes referred to as the start-up latency.

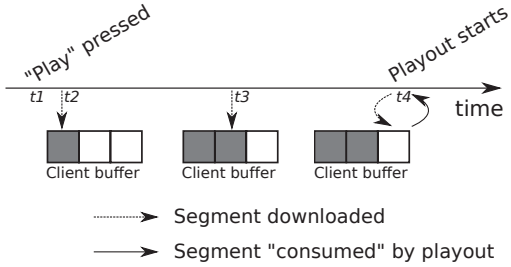


Figure 4.2: The time that elapses between the click on the "play" button and the time when the first frame appears on the screen is called start up delay. It is the time $t4 - t1$. It depends on the speed of the download (the difference between t_{i+1} and t_i) and the number of segments that must be pre-buffered.

Deadline Miss When the client starts presenting a video to the user, it implicitly sets the presentation times of all consecutive frames in the video. These are the deadlines when the video frames must be available for playout, i.e., the frames must be downloaded by then (we neglect the decoding time in this thesis). In HTTP segment streaming, frames do not arrive at the client singly, but come in groups - segments. Particularly, if the first frame of a segment misses its deadline (is late for playout) all frames in the segment miss their deadline. We therefore also speak about the segment missing its deadline.

The exception is when the player skips[4] the initial frames of a segment and so is able to catch up and present frames later in the segment without missing their deadline.

Liveness The liveness measures the delay between the time an event happens in reality and the time it is presented to the user (on a screen). For example, a goal is scored at 19:00:00, but the user sees the goal first at 19:00:10 on its screen at home. The liveness is in this case -10 seconds. In this thesis, we focus mainly on the aspects of network delivery of segments. We therefore neglect the encoding and decoding time of the segment as well as the time it takes to upload the segment to a server.

The minimal liveness is determined by the size of the client's buffer. If the client buffer size is for example 2 segments, the client must be at least 2 segments behind the live stream in order to have a chance to fill its buffer to maximal capacity (otherwise the playout consumes the buffered segments before the maximal buffer capacity is reached). The liveness can change during playout. This happens when the player does not skip segments in case of a deadline miss. An alternative name for liveness is application layer end-to-end delay, which is the absolute value of liveness.

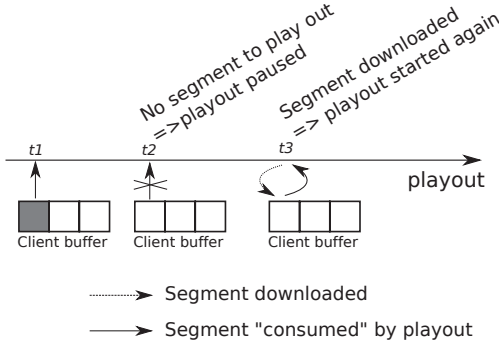


Figure 4.3: Every segment duration the playout starts playing a segment from the client buffer. If there are no segments in the buffer the playout is paused (time $t2$) until a segment is downloaded (time $t3$) and the playout continues again. The time $t3 - t2$ is the deadline miss and the liveness is reduced by the deadline miss.

4.3 Experimental Results

We evaluated the suitability of the proposed scheduler (Algorithm 1) for live HTTP segment streaming. We tested our algorithm in an emulated, fully controlled environment using Linux 2.6.31 with netem/Hierarchical Token Bucket [73] (HTB) as well as in real environment using a client equipped with WLAN IEEE 802.11b and HSDPA [25] interfaces. For all our tests we used the same 1,5 hour video stream segmented into 3127 2-second segments. Each segment

[4]The player often has to decode the initial frames anyway because frames later in the segment are dependent on the them.

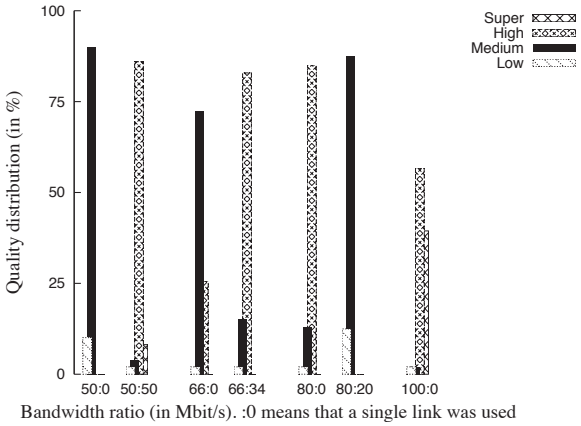


Figure 4.4: Segment video quality distribution in case of emulated bandwidth heterogeneity

was available in 4 different qualities, namely low (avg. 0.5 Mbit/s), medium (avg. 1 Mbit/s), high (avg. 2 Mbit/s) and super (avg. 3 Mbit/s).

4.3.1 Bandwidth Heterogeneity

When a client has multiple interfaces it is likely that these are technologically different and have different bandwidth. We therefore investigated the impact of bandwidth heterogeneity on the performance of our scheduler. In all our experiments we combined 2 links with possibly different bandwidths but the same aggregated bandwidth of 3 Mbit/s.

Figure 4.4 shows the effect of bandwidth aggregation on the distribution of segments' video quality. For instance, the right most columns (100:0) show the segment quality distribution when a single 3 Mbit/s link is used. This is the best case achievable since all data is downloaded continuously over one interface. The less expected result is the bad performance of 80:20 setup (two links with 2.4 Mbit/s and 0.6 Mbit/s respectively) compared to the 80:0 setup (one link with 2.4 Mbit/s). This is caused by the last segment problem [72] that occurs when a client with a small client segment buffer tries to stay too close to the live stream[5], i.e., since no new segments are available on the server, the faster interface is idle once for every segment while it waits for the slow interface to download the last partial segment. This can be mitigated by using a larger buffer as we have shown in [69]. The last segment problem demonstrates itself also in the increased deadline misses as shown in Figure 4.5.

All in all, we have found that bandwidth heterogeneity plays a significant role when aggregating the bandwidth of two interfaces. Moreover, if the bandwidth difference is huge the client is better off using exclusively the faster link.

[5] In all experiments, the client had a buffer of 2 segments, i.e., it lagged 4 seconds behind the live stream

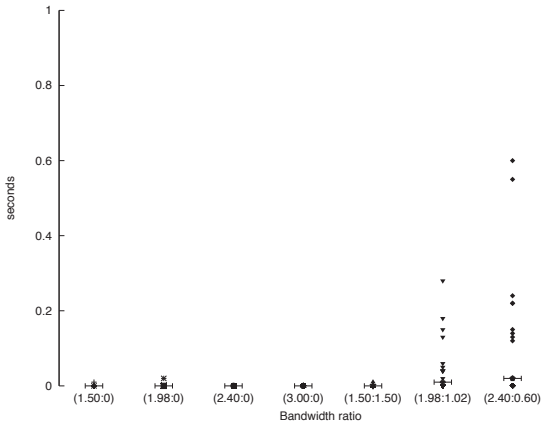


Figure 4.5: Deadline misses in case of emulated bandwidth heterogeneity

4.3.2 Latency Heterogeneity

As with the bandwidth, the chances that a client has multiple interfaces with the same latency are low. We therefore investigated how the latency heterogeneity influences the aggregated interface performance with respect to HTTP segment streaming. We emulated a client with two interfaces. One interface had a RTT of 10 ms while the other interface's RTT was varied between 10 ms and 100 ms. The bandwidth of both emulated links was 1.5 Mbit/s.

Figure 4.6 shows that the latency heterogeneity effect on the segment quality distribution is minimal. However, the RTT heterogeneity affects the number and severity of deadline misses, as seen in Figure 4.7. The buffer size limits the frequency of segment requests, and because a complete segment is not requested before the previous is finished (typical for Smooth Streaming [22], Apple HTTP Live Streaming [21]), it takes one RTT before the first bytes arrive. However, the observed lateness is not significant compared to the complete segment length. The average lateness for all bandwidth ratios is close to 0 s and the maximum observed lateness is less than 300 ms. The lateness can be fixed, as in the case of bandwidth heterogeneity, by increasing the client's buffer size. For example, increasing the buffer size from 2 to 3 segments (4-6 seconds) eliminated all deadline misses.

4.3.3 Dynamic Bandwidth Links

In the previous paragraphs the link bandwidth was set at the beginning of the experiment. A constant bandwidth like this is good for figuring out causes and effects. However a constant bandwidth link occurs only seldom in best effort networks like the Internet. We therefore investigated the performance of our scheduling algorithm in networks with dynamic bandwidth links.

Figure 4.8 shows the average throughput (over 40 runs) achieved per segment using two links separately and together. For this set of experiments the bandwidth changes were emulated so that the sum of the bandwidths was always 3 Mbit/s, but at a random interval of t seconds, $t \in [2, 10]$, the bandwidth bw Mbit/s, $bw \in [0.5, 2.5]$, was changed. The RTT

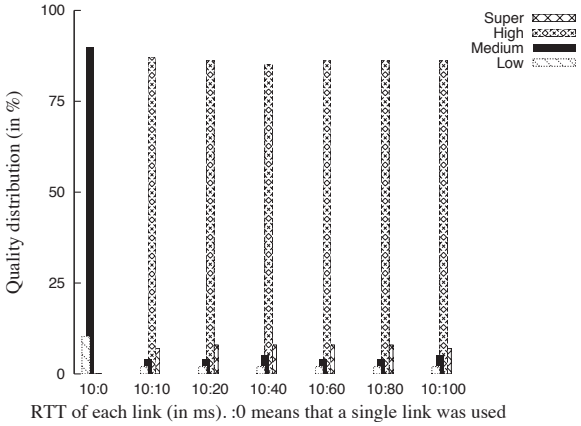


Figure 4.6: Segment video quality distribution in case of emulated latency heterogeneity

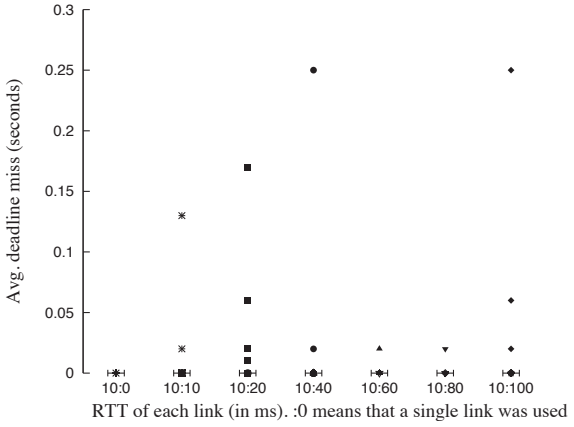


Figure 4.7: Deadline misses in case of emulated latency heterogeneity

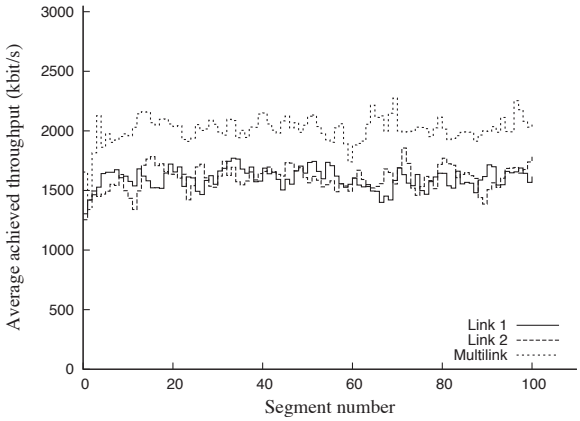


Figure 4.8: Average per segment throughput with emulated dynamic network bandwidth

of link 1 and 2 was randomly distributed between 0 ms and 20 ms, and 20 ms and 80 ms, respectively. When both links were used at the same time, the throughput exceeded the average bitrate-requirement for "High" quality almost for every segment. When single links were used, the achieved throughput per segment stayed between "Medium" and "High". Moreover, with single links, 18% of the segments had "Low", 35% "Medium", 20% "High" and 27% "Super" quality. In combination, 95% of the segments were in "Super" quality.

The real-world link aggregation results are shown in Figure 4.9. Here, we used a device equipped with a WLAN and a HSDPA interface. The average throughput experienced was 4.7 Mbit/s (max. 5 Mbit/s) and 2 Mbit/s (max. 2.5 Mbit/s) respectively. The average RTT was 20 ms (max. 30 ms) and 100 ms (max. 220 ms) respectively. We again averaged the throughput over 40 runs. The scheduler improved the performance and thereby the video quality significantly. With WLAN, the faster of the interface, 45% of the segments were in "Super" quality, compared to 91% when both links were used. The worst deadline misses observed over both links were only 0.3 s[6].

From the experiments described in this section we concluded that the proposed parallel scheduling algorithm achieves close to perfect throughput aggregation for high degree of latency heterogeneity. For links with heterogeneous bandwidths, there exists a complex trade-off between their disparity, the achievable video liveness, and the efficiency of throughput aggregation. For stable bandwidth interfaces the scheduler delivers almost perfect aggregation, but for heterogeneous interfaces like the combination of WLAN and HSDPA increased client-side buffering is required to obtain the full potential and avoid possible deadline misses.

4.4 Conclusions

We introduced and analyzed a pull-based scheduler for streaming real-time video over an aggregate of heterogeneous network interfaces. Using the file segmentation approach, which

[6]Because of the large interface heterogeneity client-side buffers of 6 segments were used.

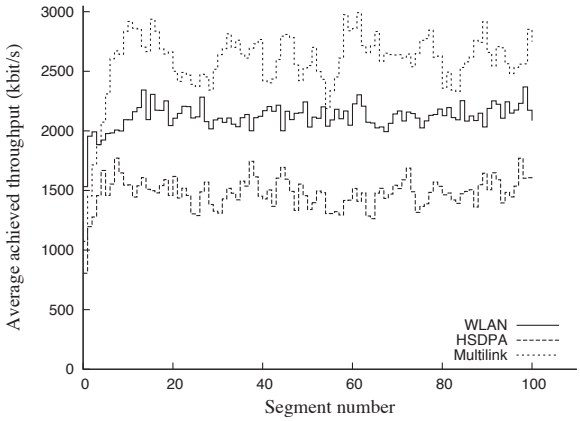


Figure 4.9: Average per segment throughput with real-world wireless links

is also used in popular commercial systems [21, 74, 22], the proposed scheduler adapts the video quality according to the combined throughput of all the available interfaces at the client.

Experiments conducted in a controlled emulation testbed showed that the scheduler achieves close to perfect throughput aggregation for high degrees of latency heterogeneity. For links with heterogeneous bandwidths, there exists a complex tradeoff between their bandwidth disparity, the achievable video liveness, and the efficiency of throughput aggregation. For two stable interfaces of equal bandwidth, the scheduler achieves close to perfect bandwidth aggregation, while maintaining a quasi-live video stream, i.e., never more than 4 seconds late (after a segment is available at the web server, thus excluding encoding and server uploading delay). If the bandwidth heterogeneity is higher, which is the case for a combination of WLAN and HSDPA, about 10 seconds of delay have to be accepted to obtain the full potential of aggregated throughput and best possible video quality.

Chapter 5

Enhancing the Server Performance

Upgrading the client link does not help if the server link, shared by many clients, becomes the bottleneck. The straight forward solution to this problem is to replace the server link with a link with a higher capacity. However, this is not always possible. In this chapter, we look at how to make the streaming more efficient so that the same server can serve more clients possibly with higher quality segments over the same link.

In the beginning of this chapter, we first introduce TCP concepts important to HTTP segment streaming. This is needed in order to understand the challenges TCP is having when faced with segmented streaming and in this chapter we need this level of detail. We later look at what kind of traffic pattern HTTP segment streaming exhibits. We point out the differences to a similar, but not the same traffic pattern, the web traffic pattern.

In the last part of this chapter, we look at and evaluate the possible solutions that could lead to a better server performance.

5.1 Transmission Control Protocol

Today, TCP is the protocol of choice for delivering content in the Internet. With the HTTP [19] application layer protocol it delivers both the static content and the dynamically generated content. The combination of TCP and HTTP is also used for HTTP segment streaming. Since TCP governs the transmission of video segments we shall describe its properties in this section.

Vinton G. Cerf and Robert E. Kahn described the workings of TCP (or the Transmission Control Program as TCP was originally called) in their paper *Protocol for Packet Network Intercommunication* [75] in 1974. In their work they described the fundamental features of TCP like for instance the segmentation of a byte stream produced by an application and the sliding window concept for the flow control. They also described the addressing of TCP segments as well as the fragmentation of TCP segments when they transit to a network with a smaller Maximum Transmission Unit (MTU). The original proposal was later split into the networking layer part and the transport layer part [76]. RFC 791 [61] specifies the networking part - the IP protocol. RFC 793 [18] specifies the transport layer part, i.e., it includes the original TCP [75] with additional improvements. Additionally, the User Datagram Protocol [17] was specified on the transport layer for applications that use only the addressing and

routing features of IP and do not need the features of TCP.

Even though there are many improvements and innovations to the original TCP from 1974, the main concepts still remain. Nowadays, TCP has the following five features:

Connection-orientation TCP peers establish a virtual connection (circuit) by performing a three-way handshake during the beginning of the connection. Note that data is exchanged first after the handshake is completed, i.e., the virtual connection is established. This is not enforced by the TCP standard, but is done in practice for security reasons. In the future, the TCP Fast Open feature [77] is going to enable the client and the server to exchange data while performing the three-way handshake, increasing the performance of short TCP connections.

Stream-orientation TCP, on the transport layer, reads the data supplied to it by the application layer as a stream of bytes. It internally frames the continuous stream into packets (segments) that the network layer can handle (in case of the Internet it is the packet based IP protocol).

Reliable When the sender application layer gives data to TCP, TCP guarantees that the data is going to be transmitted and then given to the receiver application in the same order and uncorrupted. If the data or parts of the data can not be transferred due to, for example, a network failure, TCP takes care that the receiving application gets as much continuous in order data as possible.

Flow Control The TCP on the receiving side uses a limited buffer to buffer the received data. The size of the buffer is stored in the receive window `RWND`. The application layer must pick up this data once in a while to create room for new data to be received. If the application layer does not pick up the data, the receiver would run out of buffer and would have to start throwing away received data. To avoid this, TCP implements a mechanism, the flow control, to keep the sender updated about the amount of free space in `RWND`. This way the sender can stop sending data if the receiver's buffer is full.

Congestion Control The sender TCP keeps an estimate of the connection's bandwidth. It then regulates its sending rate to match the estimated bandwidth. This is called the congestion control and is the subject of the next section.

5.1.1 Congestion Control

The early TCP's reaction to loss was a too aggressive retransmission, which led to congestion collapse in 1986 [78]. The term congestion collapse was coined by John Nagle in [79] and describes an overloaded network state in which senders react to data loss by sending even more data (retransmissions) and consequently overload the network even more. This was soon realized by the networking community and TCP was updated by congestion control techniques like the slow-start, exponential retransmit timer backoff, round-trip-time variance estimation, dynamic window sizing during congestion, fast retransmit [78]. The goal of all

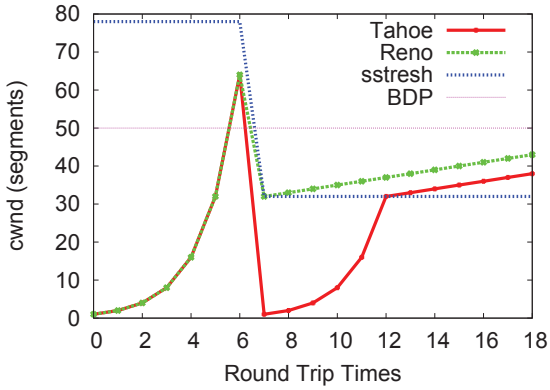


Figure 5.1: TCP congestion control in action

these techniques is a good estimation of connection's bandwidth and RTT, which in turn allows the sender to adjust its sending rate so as not to cause congestion in the network.

Sender's estimation of connection's bandwidth-delay product (BDP) is tracked by the congestion window variable (CWND). CWND represents the number of bytes the connection's link can store, i.e., that can be in transit from the sender to the receiver (also called *in flight* bytes). This is the sender's estimation of the number of bytes that it can send without receiving an acknowledgement (note that the receiver can further restrict this number with `RWND`, i.e., flow control).

Congestion control, in its basic form, is composed of two phases. The additive increase (AI) phase and the multiplicative decrease (MD) phase. In the additive increase phase the CWND is increased by one segment every RTT. Eventually, the window increases above the network bandwidth limit and a packet is dropped. When this happens, the CWND is reduced to half of its value. It has been experimentally shown that the AIMD approach leads to fair sharing of a bottleneck link.

Because the CWND is increased very slowly during the AI phase, an algorithm called *slow-start* is used to open the window initially. An example is provided in Figure 5.1. When the connection starts, CWND size is doubled every RTT (each ACK increases CWND by one). This is the *slow start* phase. The slow start phase lasts until `SSTHRESH`/`RWND` limit is reached or the first packet is dropped (round trip time 6). In case of a drop, the `SSTHRESH` is set to half of the CWND. `SSTHRESH` is the threshold for switching from the slow-start phase to the AI phase[1].

What happens after the first slow-start phase is over depends on the congestion control algorithm used. In case of Tahoe, the CWND is set to its initial value TCP Initial CWND [1] (IW) and the slow start phase takes over again. In case of Reno, the CWND is set to

[1]To achieve 1 Gbps with 1460 byte segments without slow-start would take about 4600 RTTs, that is 7.5 minutes for a RTT of 100 ms. However, with slow-start the full speed is reached in about 1.2 seconds.

SSTHRESH and the AI phase is entered. There are many more TCP congestion control algorithms besides Tahoe and Reno [80, 81, 82, 83, 84, 85].

5.1.2 TCP's State after an Idle Period

In this section we discuss what happens to the CWND when the connection is idle. There are three alternatives in use:

1. The default behavior of TCP after an idle period is specified in RFC 2581 [1]. This RFC states in section 4.1: *"Therefore, a TCP SHOULD set CWND to no more than TCP Restart CWND [1] (RW) before beginning transmission if the TCP has not sent data in an interval exceeding the retransmission timeout."*. This is the behaviour of Windows Vista, Net/FreeBSD, which do restart from Initial Window (IW).

2. Linux uses by default behaviour specified in an experimental[2] RFC 2861 [87]. This RFC states that the connection should reduce its CWND by half for every RTT the connection is idle and SSTHRESH should be set to the maximum of its current value and half-way between the previous and new CWND. The RFC also specifies that this should happen also in case in which the connection speed is application limited, i.e., when the application does not send enough data to utilize the full CWND. The motivation for decaying the CWND instead of reducing it to RW is to improve the speed right after the connection becomes active again. Figure 5.2 shows for a 1 Gbit/s link with around 1 ms delay by how much the CWND decreases based on the amount of time the connection was idle.

3. The third possibility is to keep the CWND as it was before the connection went idle, i.e., ignore the fact that it was not validated during the idle period. This is the default behaviour of Mac OS and Solaris.

5.2 Traffic Patterns

In this section we look at the traffic patterns that are induced by different streaming solutions. We compare traffic patterns as seen from the point of view of a single client as well as seen from the view of the server that serves possibly thousands of clients.

5.2.1 Related Work

On-off sources using TCP for delivering data have been studied in the literature a couple of times. Analytical models were provided for example for simplified TCP Tahoe [88, 89], Vegas [90] and Reno [91]. However, most of these models assume exponentially (randomly) distributed on and off period durations whereas in the HTTP segment streaming scenario the

[2]Experimental RFCs are for specifications that may be interesting, but for which it is unclear if there will be much interest in implementing them, or whether they will work once deployed [86].

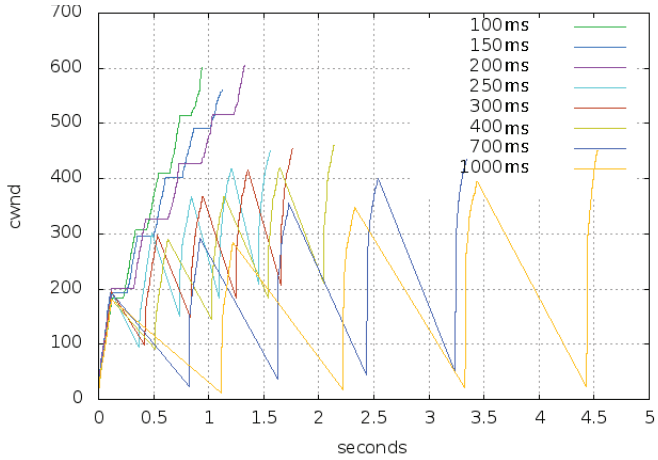


Figure 5.2: Example of Linux CWND after an idle period. Downloading five 10 MB segments over the same TCP connection with variable delay between the downloads.

on and off periods are distributed differently. Different is also that an HTTP segment streaming on period is first over when all bytes of a segment have been downloaded, not when a certain time has elapsed. Furthermore, others [92] assume perfect bandwidth sharing between TCP connections. This is of course a valid assumption if the on periods are adequately long to establish bandwidth sharing fairness, but in our scenario the stream segments are too small to make this a valid assumption (e.g., for Cubic's behaviour see [82]).

Adaptive HTTP segment streaming has been a hot topic over the past few years, but most of the research has focused on the observation and the improvement of a single connection [93, 62, 63]. Esteban et al. [94] looked at the interaction between TCP and adaptive streaming. They categorized TCP segment loss into three categories: initial burst losses, ACK clocking losses and trailing ACK losses. They identified the trailing ACKs losses as the worst ones, because they increase the duration of the segment download by one RTT and can potentially lead to a Retransmission Time-Out [95] (RTO). In their work, they investigated how application layer pacing could help reduce the amount of trailing ACK losses. However, their conclusion was that their efforts do not improve the situation. Akhshabi et al. [96] studied the *on-off* behavior of adaptive streaming connections. They showed for example the relationship between clients' on period overlapping and system stability (number of bitrate switches), as well as, the available bandwidth and the system stability. They also showed how the system stability depends on the number of clients given the total bandwidth stays the same.

The aforementioned works study single HTTP segment streaming connections or a small number of HTTP segment streaming connections (12 in case of [96]), i.e., they do not look at a big set of HTTP segment streaming connections and their interactions. However, in a big multi-user scenarios, the server will become the bottleneck [97], and we therefore need to look at how multiple connections behave when sharing a server.

5.2.2 Continuous Download

Continuous download is the most basic form of "streaming" videos over TCP. The video is made available as a file on a server and the client downloads it. To accommodate for bandwidth fluctuations, the client waits until a safe amount of data is downloaded before starting the playout in parallel to the ongoing download (Figure 5.3). Figure 5.4 shows that the download link is utilized at all times. Note that the goodput (the actual video data) comprises the majority of the transfered data, but there is also the TCP/IP overhead - TCP congestion control and protocol headers.

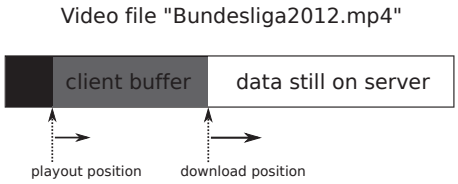


Figure 5.3: Continuous download: The situation on the client.

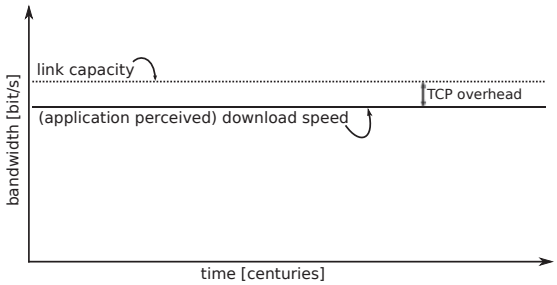


Figure 5.4: An example of continuous download traffic pattern (when sampled over a reasonable time).

The only limiting factors are the client buffer (how much data the client can pre-buffer), the network bandwidth, RTT and loss. Since TCP was optimized for bulk download of data, all the advantages like fair sharing (see Figure 5.5) and optimized utilization of a link do also apply for continuous download.

The disadvantages are mainly application (streaming) related. For example, the client might decide to stop the playout before the video is over. In this case, all bytes that were pre-fetched, and not watched yet, are wasted. Another disadvantage is that this type of streaming does not provide means of adapting to bandwidth changes when the playout has already started, i.e., the bitrate can be in many cases chosen before the stream is started, but can not be adapted (without playout interruption) to current bandwidth. If the bandwidth later increases, the client is not going to use it to get the video in higher quality. If the bandwidth

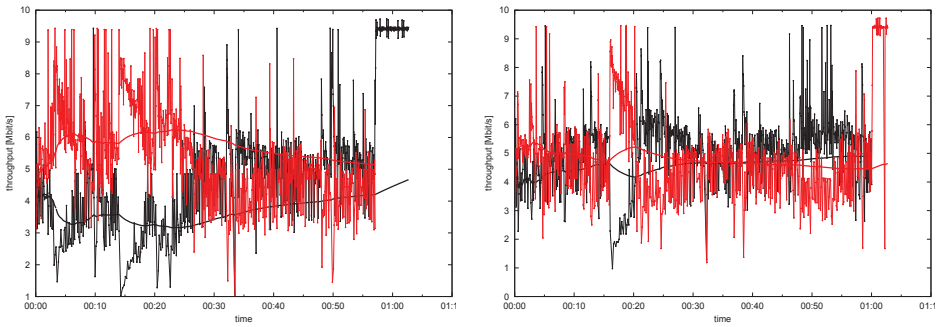


Figure 5.5: Two examples of bandwidth sharing between 2 greedy TCP connections on a 10Mbit/s link.

decreases, the client is going to eventually run out of the pre-fetched data and unpleasant "buffering" periods occur.

5.2.3 HTTP Segment Streaming's On-Off Traffic Pattern

HTTP segment streaming differs from continuous download by segmenting a video file into multiple segments, which are encoded in different qualities and made available for a client to download. It is up to the client how it combines the segments effectively, allowing the client to switch quality (bitrate) on a segment bases. The basic download pattern is shown in Figure 5.6 (for simplicity only segments of one quality are shown). The segmented file is available on a server in the form of multiple segments. The client requests the segments one after another. The link between the server and the client is fully utilized with the exception of short periods of time that the request for the next segment needs to get to the server. However, even this gap can be completely eliminated with request pipelining.[3]

The traffic pattern illustrated in Figure 5.6 is very similar to continuous download pattern. However, in practice, the client can buffer only a limited number of segments. As a consequence, when the buffer gets full, the client must wait with the request of the next segment until a segment is played out and free space in the client buffer becomes available. In this way the continuous download pattern degenerates to an *on-off* pattern, i.e., the client is switching between downloading and waiting for free space to become available in its buffer.

This behaviour is similar to live segment streaming in where the client is additionally limited by the fact that the segments become available on the server only in regular intervals. A live HTTP segment streaming client typically downloads each segment as it becomes available. Since there are only a finite number of bitrates to choose from, the chosen bitrate will most likely not match with the available bandwidth perfectly. To be sure the segment is

[3]The client needs to decide which segment to request before it sends the next request. If pipelining is used this decision needs to be taken before the previous segment is completely downloaded.

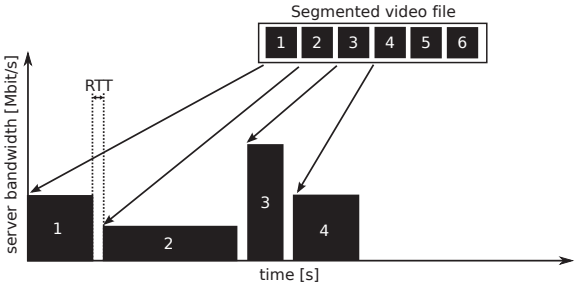


Figure 5.6: Example of VoD traffic pattern with unlimited buffer space.
(without bandwidth adaptation and request pipelining)

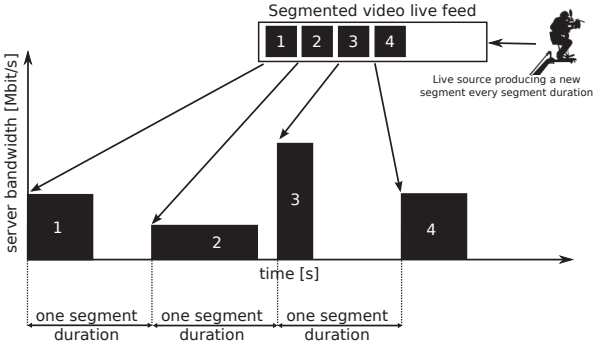


Figure 5.7: Example of live/VoD with limited buffer space traffic pattern.
(without adaptation)

going to be received in time, the client chooses the highest bitrate segment that it estimates is downloadable given the estimated available bandwidth. Because the bitrate of the chosen segment is smaller than the estimated bandwidth, it is very likely that the segment is going to be downloaded before the next segment becomes available on the server. This way an idle period arises between the downloads of two consecutive segments. Figure 5.7 illustrates this behavior. Figure 2.6 shows a trace from a real VoD HTTP segment streaming session. From the linearity of the segment request time line we see that the client requested the segments in regular intervals (the horizontal part in the beginning of the streaming session is the time when the client was pre-buffering segments). This real world trace shows that even a VoD traffic pattern degenerates to an *on-off* pattern. Additionally, Figure 2.6 shows the varying bitrate of the downloaded video segments - the bandwidth adaptation algorithm in action.

The *on-off* pattern generated by HTTP segment streaming has one more dimension. This dimension emerges when multiple streams share the same link, which is not an unrealistic scenario. Consider, for example, the live stream from the Olympic games in London in 2012. In situations like these, thousands of users access the stream and follow the happenings in

real time. The additional dimension emerges from observing the clients that share the same (bottleneck) link over time. It is reasonable to expect that each client wants to see the media presentation as close to real time as possible (you want see the goal before it is revealed to you by the cheering from the neighbor's apartment). Therefore, all clients have an incentive to download the video segments as soon as they become available on a server. Figure 5.8 extends the previous illustration with the additional dimension.

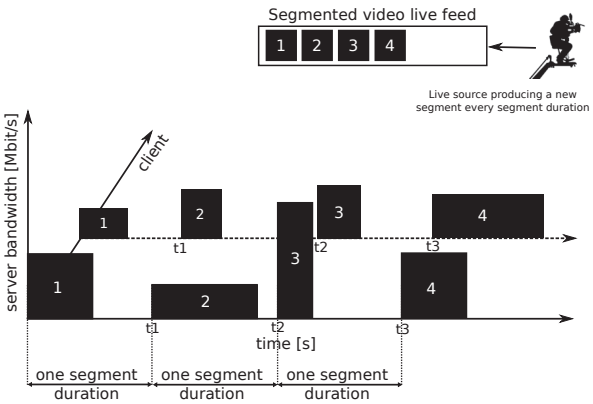


Figure 5.8: Example of live traffic traffic pattern with 2 clients. (without adaptation)

The major challenge from the client's point of view is to achieve the best throughput the network can provide given the *on-off* behaviour. This is challenging because TCP has been optimized for uninterrupted transfers and not for short, periodic *on-off* traffic [94]. The idle periods and rather small segments (a 2-second segment of a video with a bitrate of 1.5 Mbps is about 370 KB big) do not fit the optimization goals of TCP. Nevertheless, there are possibilities to improve the server performance by modifying TCP configuration and application layer behaviour and this is what we look at in this chapter.

5.3 HTTP Segment Streaming Performance

The data analysis performed in Chapter 3 has shown that not all the data received by the clients is served by the origin server. However, usually it is only the origin server that is under the control of the streaming provider. Especially the segments that are cached are delivered from servers that are not under the control of the streaming provider. As such, these servers can only be optimized to a rather limited extent for HTTP segment streaming (even though the live stream can make up a significant part of their downstream traffic). It is actually one of the major strengths of HTTP segment streaming that stateless "of the shelf" HTTP server (infrastructure) can be used. In other words, extensive server modifications are not desirable and sometimes even not possible.

On the other hand, changes are possible on the client-side. One possibility is to write

players from scratch [47] or use a player that allows instrumentation like the player from Microsoft, which is based on the Silverlight™ development platform. Either way, it is usually the client-side that allows changes that can improve the HTTP streaming performance. In the next sections we look at different ways to modify the client and the server to improve the system performance. All the changes might have an effect on the performance of a single server in terms the number of concurrent clients it can support. These results were published in [98, 99] and we cite many times directly from these papers.

5.4 Simulation Suite

In the scope of this thesis, we developed a simulation suite to run and visualize different HTTP segment streaming scenarios. A screen shot of the main GUI interface to the suite is shown in Figure 5.9. As shown, one can choose different values for parameters like the number of clients, client arrival distribution, quality selection strategy, number of segments, name of the stream etc. After the selection is completed a simulation is started by clicking on the "Submit Query" button.

Here, we chose simulation over emulation, because of the speed, reproducibility and because a network consisting of hundreds of computers is practically not feasible in our research lab. We wrote a parametrizable ns-2 [100] script to simulate and record the performance of periodic *on-off* video streams in a multi-user server scenario. Our simulation setup is based on the observation that network bottlenecks are moving from the access network closer to the server [97]. The ns-2 script is parametrized with the parameters selected in the GUI and executed. The results of the simulation are stored in an SQL database for analysis.

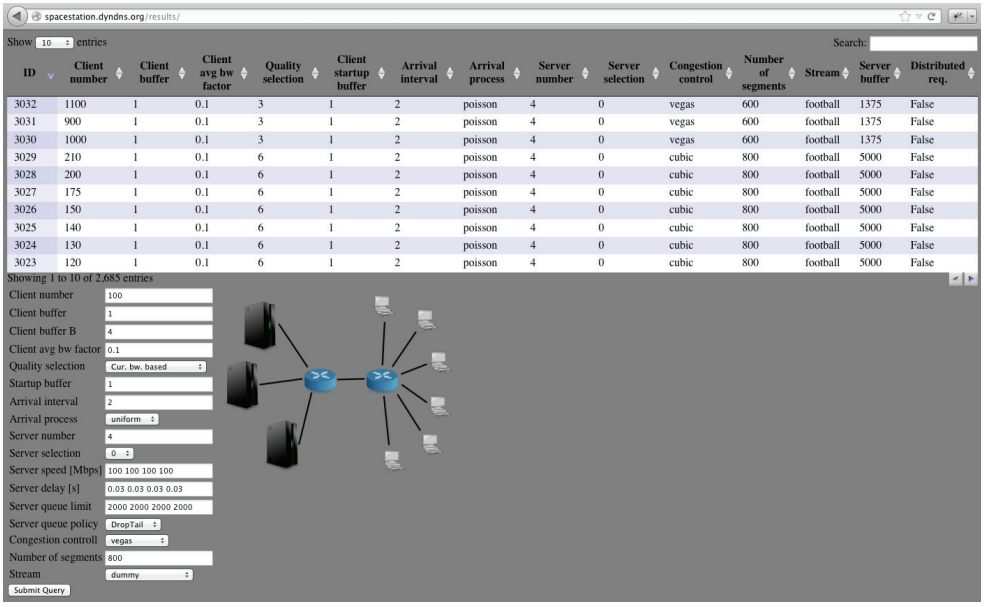


Figure 5.9: Simulation suite GUI

The analysis screen is shown in Figure 5.10. The user can quickly select between different types of graphs that help visualize different aspects of the simulation. In Figure 5.10, for example, we see the number of active clients (clients that are in the on period) over time. Altogether there were 200 clients in this particular simulation and we see that they were all active in periodic intervals.

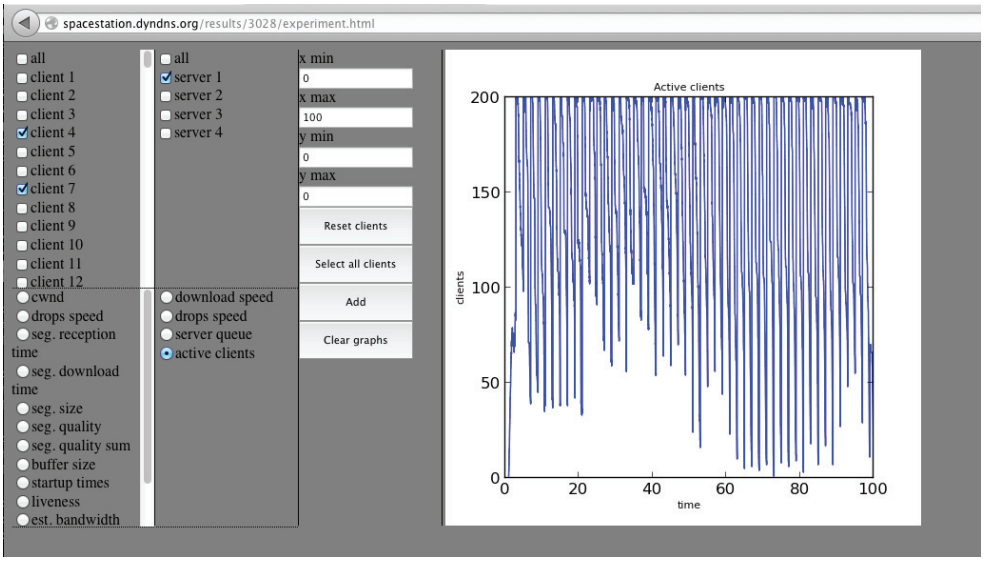


Figure 5.10: Analysis tools GUI

5.5 Simulation Setup

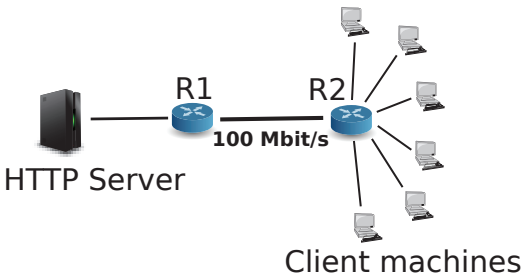


Figure 5.11: Simulation setup

The simulation suite setup that was used in the following sections (unless stated otherwise) to measure the various performance metrics is shown in Figure 5.11. The links between the client machines and the R2 router are provisioned for more than the highest media bitrate

Quality	Bitrate
0	250 kbit/s
1	500 kbit/s
2	750 kbit/s
3	1000 kbit/s
4	1500 kbit/s
5	3000 kbit/s

Table 5.1: Stream bitrates

(Table 5.1). The link between the HTTP server and the router R1 is overprovisioned and can support the highest media bitrate for all clients. The link between R1 and R2 is the simulated bottleneck link at the server-side with a capacity of 100 Mbit/s (we have also tested with a 1 Gbps bottleneck link, but the trends are the same, i.e., only the total number of clients is scaled up).

The delays between the clients and the server are normally distributed with an average of $\mu = 55ms$ and a variation of $\sigma = 5ms$. Distributing the delays prevents phase effects in the simulation, and it also seems to be a reasonable assumption for an ADSL access network [101]. The router queue is limited to 1375 packets, which corresponds to the rule of thumb sizing of bandwidth delay product. Furthermore, we modeled client arrivals as a Poisson process with an average inter-arrival time $\lambda = \frac{number\ of\ clients}{segment\ duration}$, i.e., all clients join the stream within the first segment duration. This models the case when people are waiting in front of a 'Your stream starts in ...' screen for the start of the game or TV show so they do not miss the beginning.

On the TCP part, we chose to evaluate the Linux version of the algorithms implemented in ns-2, because, according to [102], over a half of all web servers in the Internet is running the Linux operating system.

Finally, to have a realistic set of data, we used the soccer stream used in [103], encoded with variable bitrate (Table 5.1) as in a real HTTP segment streaming scenario.

5.6 Varying Parameters without Quality Adaptation

To understand the workings of HTTP segment streaming, we first investigated a live streaming scenario in which the stream was available only in one quality, namely 500 Kbit/s, that is quality 1 from Table 5.1. In this way we can observe the system's performance without introducing too much complexity that quality adaptiveness brings with it. For our simulation we used a 600 segments long video stream. Each segment was exactly 2 seconds long, which means that each simulated run took about 20 minutes.

In the previous chapter we described how even HTTP segment streaming solutions with client-side buffering degrade from continuous download (this is the time when the buffer is being filled) to an *on-off* pattern when the buffer is full. To create the *on-off* traffic pattern in our simulation right away, we set the client buffer to only one segment. In other words,

we remove the pre-buffering period and the client starts playing right after the first segment is received. Note that the client was not allowed to skip video content in order to keep up with the life stream, i.e., even if a segment is received after its deadline (its scheduled playout time) it is played out completely and the deadline of all other segments is moved into the future. Having only one segment buffer, the clients are forced to enter the download and wait cycle right in the beginning of the streaming. As mentioned above, there are two cases when this happens in reality. The first case is when all clients start the streaming from a point in time as close to the live stream as possible. The second case occurs when all clients catch up with the live stream and have to wait for the next segment to become available. We present the results for different parameter variations for the setup described above in the following sections.

5.6.1 Varying TCP Congestion Control

A typical development of CWND during the *on-off* cycle is shown in Figure 5.12. This is the second case from Section 5.1.2, i.e., RFC 2581 [1]. CWND opens during segment transfer (on period) and collapses to RW when the client is waiting for the next segment. In other words, in the majority of all cases[4] the on (download) period starts with a TCP slow start phase.

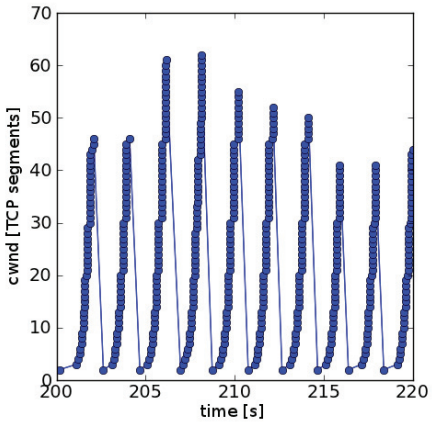


Figure 5.12: Observed TCP congestion window in ns-2

In this section we show how different TCP congestion control algorithms cope with the *on-off* traffic pattern. The default TCP congestion control algorithm on Linux is Cubic [82]. However, Linux is an open source operating system that attracts researchers and developers of TCP congestion control algorithms and as such many other different algorithms are implemented in Linux as well. It is just a matter of changing the
`/proc/sys/net/ipv4/tcp_congestion_control`
variable in the virtual `/proc` filesystem. Because of the ease we believe that this is not

[4]CWND collapses to RW only if the off period is longer than RTO [1].

a difficult modification to the machine running the HTTP server and so is in line with the philosophy of not changing and reusing the already deployed infrastructure.

Figure 5.13 shows the achieved average liveness, average startup time and number of packet drops across all clients for the tested congestion control algorithms given the setup in Section 5.5[5]. The liveness, as described in Section 4.2, measures the time that the client lags behind the live stream. In the figures here it is specifically the liveness of the client at the end of the stream (after 600 segments). For example, a liveness of -4 seconds means that the client played the last segment 4 seconds after it had been made available on the server.

The server bandwidth of 100 Mbit/s should provide enough bandwidth for smooth playout for around 200 clients. However, the liveness graph shows that this is not achievable and as the client number grows, the liveness decreases. Because the start up times, Figure 5.13(b), do not increase, the decrease of liveness can only be contributed by deadline misses, or in other words by unpleasant playout stalls. The reason for the playout stalls is an inefficient use of bandwidth.

The reason for the inefficiency is found in the overflowing of the R1 router queue. In Figure 5.14(a) we see that the queue runs over approximately every segment duration at the time just after a segment became available (the end of the off period). When the queue is full, the new incoming data packets are discarded[6], Figure 5.13(c), and must be retransmitted.

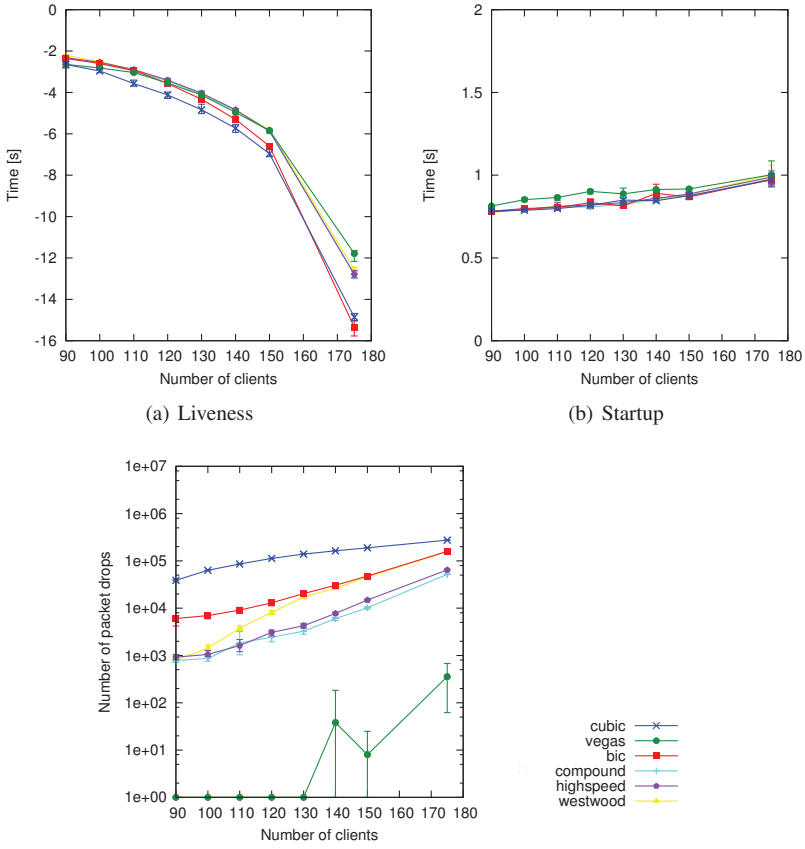
The loss-based, more aggressive variants of TCP congestion control algorithms like Cubic and Bic generate the highest losses and have the worst liveness. An interesting alternative is Vegas. Vegas is a delay based congestion control algorithm that backs off before the bottleneck queue overruns. We see that Vegas performs better than Cubic also in terms of liveness and can cope with an increased number of clients better. A sample development of the R1 router queue for 150 clients is shown in Figure 5.14(b). It is clear from Figure 5.13(c) that Vegas induces only very few packet losses (due to queue overruns) even though the download periods are short (around 2 seconds). However, we also want to mention that Vegas has been shown [104] to perform badly when competing with loss-based congestion controls such as Cubic or Reno. Therefore, unless the majority of streams running through the bottleneck is HTTP segment streams from machines configures for using Vegas (for example a bottleneck very close to an HTTP streaming server), one must consider carefully the deployment of Vegas.

5.7 Increased the Segment Duration

Recent congestion control algorithms were optimized for bulk transfer. However, the segmented approach of HTTP streaming creates a short interval *on-off* traffic rather than bulk traffic (commercial systems use standard segment lengths from 2 seconds [22] to 10 seconds [21]) and longer on-times than, e.g., HTTP/1.1. To see the effects of the segment length, we also changed this parameter.

[5]Every experiment was run 10 times with different arrival time distribution. The plots show the average value and the error bars the minimum and maximum value.

[6]R1 is running a drop-tail queue.



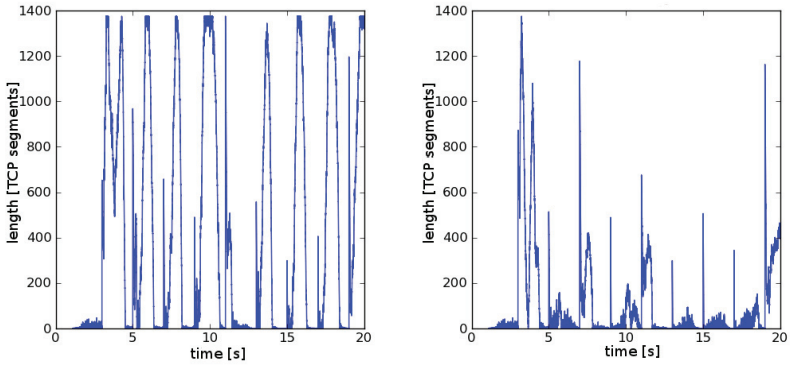
(a) Liveness (b) Startup

(c) Number of dropped packets

Figure 5.13: Performance of (ns-2 version) Linux TCP congestion control algorithms

Figure 5.16 plots the results using different segment lengths. For example, Figure 5.16(a) shows that the startup times are slightly higher for 10-second segments because they take a longer time to download than 2-second segments. Please note that 10-second segments are 5 times longer in duration than 2-second segments, but, due to increased compression efficiency, not necessarily 5 times larger in size. Quality switching between segments requires (see Section 2.2.2) that every segment starts with an intra encoded frame, but the rest of the frames in a segment can be lightweight P- and/or B-frames that occupy a much smaller fraction of the segment than an I-frame. Nevertheless, longer duration segments are larger than shorter ones and therefore require more time to download, prolonging the download (on) periods. This gives TCP more time to reach its operating state.

Our experiments show that the liveness is lower than in case of 2-second segments for multiple reasons. Firstly, we distributed the client arrivals also here across one segment duration, i.e., 10 seconds. Therefore a client arrives on average later and so decreases the liveness because of its increased startup time. However, note that the clients still synchronize



(a) Cubic (b) Vegas

Figure 5.14: R1 router queue: 150 clients

when they request the second segment. Therefore, in this respect, this scenario is similar to the 2-second segment scenario, but instead of synchronizing their requests at the third second, clients synchronize at the eleventh second of the simulation.

We can observe that client liveness changes from about -8 seconds to -24 seconds for 175 clients. The drop of liveness with the number of clients is very much comparable to the 2-second case. This indicates that longer download periods do not help TCP to cope with an increasing number of concurrent clients.

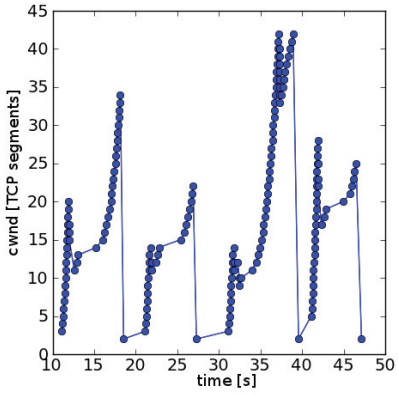
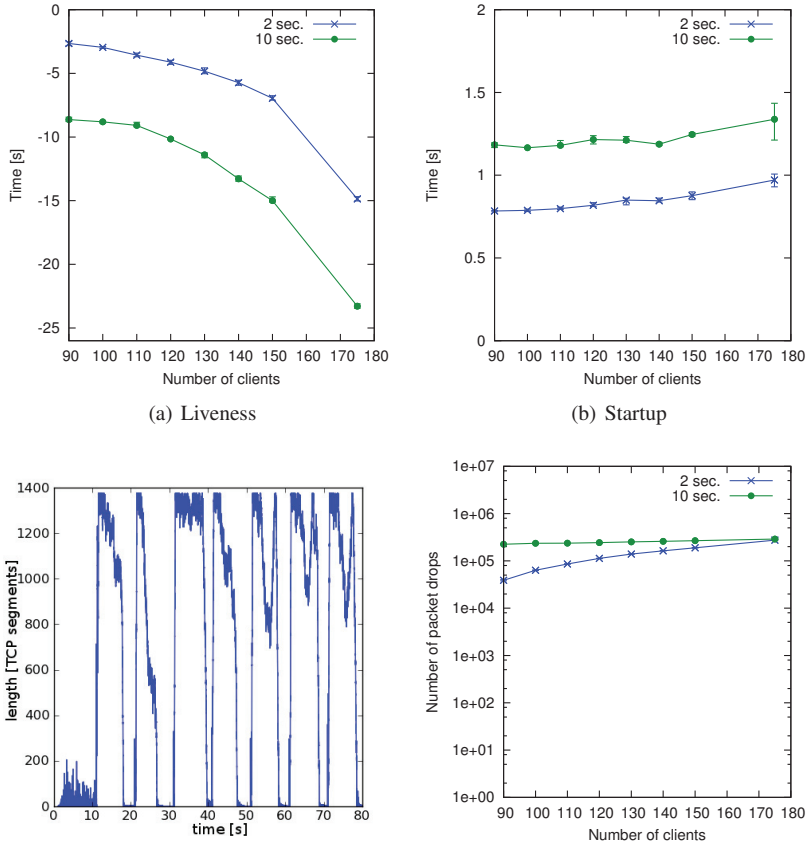


Figure 5.15: Sample TCP congestion window for 10-second segments

To prove that TCP reaches its envisioned operating state we plotted a trace of a congestion window using 10-second segments in Figure 5.15. Compared to the 2-second segment scenario in Figure 5.12, we see that the window size oscillates with a lower frequency, i.e., relative to the segment size. We can also see the characteristic curve of Cubic when it probes for available bandwidth. There is no time to do that in the 2-second segment scenario.



(a) Liveness

(b) Startup

(c) R1 router queue - 150 clients

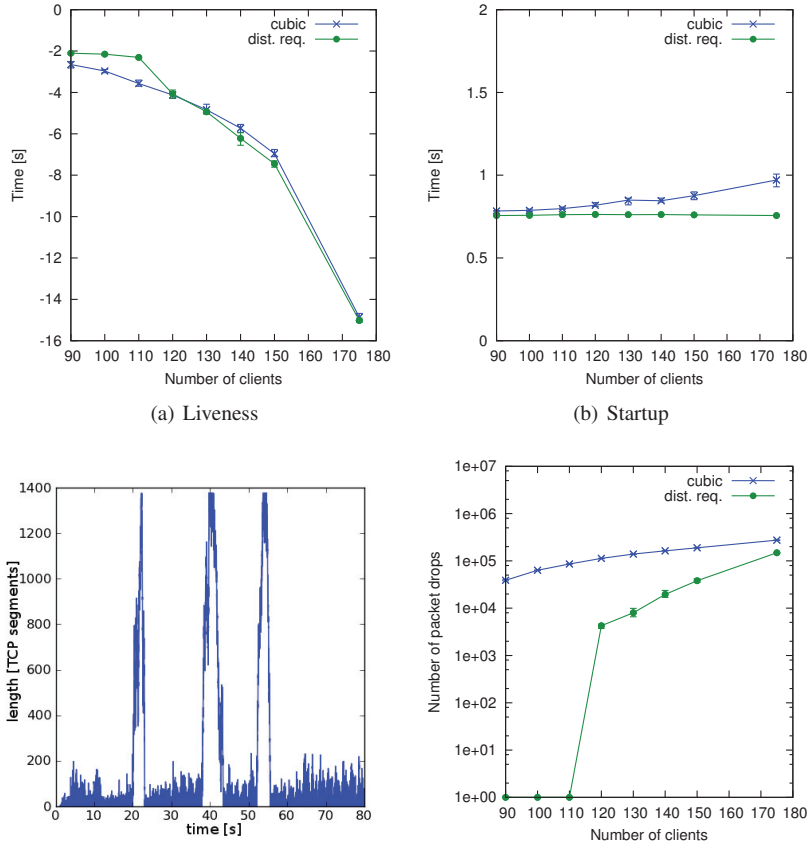
(d) Number of dropped packets

Figure 5.16: Performance of longer segments: 10-second segments (Cubic)

The probing for bandwidth leads to queue overruns as shown in Figure 5.16(c). These overruns trigger almost constant packet loss independent of the number of clients (Figure 5.16(d)). Based on this observation we would rather lean towards using 2-second segments because they give us better liveness and give the adaptation algorithm 5 times more chances to adapt stream's quality to the available bandwidth.

5.8 Requests Distributed over Time

In a live streaming scenario, the clients usually download the new segment as soon as it becomes available. Thus, the server will experience large flash crowd effects with a huge competition for the server resources. However, such segment request synchronization leads to reduced performance in terms of quality and liveness. The negative effects can be seen in the default configuration of the server where the router queue fills up (Figure 5.14(a)), and



(a) Liveness (b) Startup

(c) R1 router queue: 150 clients (d) Number of dropped packets

Figure 5.17: Performance of regular vs. distributed requests (Cubic)

packets are dropped (Figure 5.13(c)), i.e., the performance degrades.

To avoid this synchronization, we propose to distribute the requests over one segment duration, i.e., the time period between two consecutive segments' availability times. For example, after arrival, a client can request a segment every segment duration. This algorithm requires the clients to check the media presentation description only in the beginning of the streaming to find out which segment is the most recent. After that they can assume that a new segment is going to be produced every segment duration. This way the requests stay distributed over time also beyond the first segment.

In our experiment, the requests are exponentially distributed over the entire segment duration. The results show that distributed requests increase the liveness if the client number is small (Figure 5.17(a)). If the number of clients is high, clients that get more than one segment duration behind the stream implicitly build up the possibility for buffering, i.e., they break the download-and-wait cycle, go over to continuous download and so reduce the effect of distributed requests until they catch up with the live stream again. We also see that distributed

requests lead to less pressure on the router queue (Figure 5.17(c)), which can possibly leave more space for other traffic, and that the number of packet losses is greatly reduced (Figure 5.17(d)). In section 5.12.8, we explain that the reason for this is that distributing requests over time leads to less concurrent downloads (competing connections) at every point in time, which is the reason for less packet losses.

5.9 Limited Congestion Window

In Video-On-Demand scenarios, clients can download the whole stream as fast as their download bandwidth and their playout buffers permit. On the other hand, in live streaming scenarios, clients are additionally limited by segment availability. In the download-and-wait cycle, a fast download of a segment prolongs only the wait period. Hence, there is no need for the client to download a segment as quickly as possible as long as it is downloaded in time for playout.

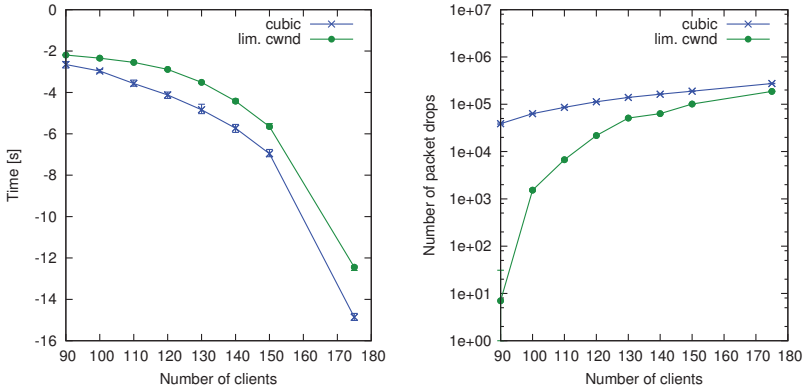
TCP's bandwidth sharing is fair for long running data transfers. However, for short transfers, the sharing is in many cases very unfair. To reduce this unfairness, we experiment with a limited TCP congestion window. The limited congestion window can lead to longer download times, resulting in a behavior similar to TCP pacing [105]. To avoid playout stalls due to congestion window limitation, we chose the congestion window so that a segment can easily be downloaded in one segment duration. We set the congestion window to 20 TCP segments, which equals a bandwidth 3 times bigger than the average bitrate of the stream (to account for bitrate variance). The startup time is low as in the other scenarios tested above, yet the average liveness is improved (Figure 5.18(a)). Furthermore, from Figure 5.18(b), we observe a significantly reduced number of dropped packets, which also indicates a lighter load on the bottleneck router queue, resulting in a more efficient resource utilization. This is because our TCP congestion window limitation prevents a single connection to grab more bandwidth than a bit more than what it really needs.

5.10 Varying Parameters with Quality Adaptation

In the previous section, we showed how different changes of default settings have a beneficial impact on HTTP segment streaming without adaptation. In this section, we look at the impact of these changes on the adaptive HTTP segment streaming. We chose a classic adaptation strategy that is very similar to the strategy used by Adobe's HTTP Dynamic Streaming [20, 106]. This strategy is known to follow the available bandwidth very closely [62, 63]. We use the following formula to estimate the available bandwidth:

$$b_s = \alpha * b_{s-1} + (1 - \alpha) * \frac{segment\_size_s}{download\_time_s}$$

The estimated bandwidth b_s after the download of a segment s is the weighted sum with aging (α was 0.1) of the estimated bandwidth after segment $s - 1$ and the quotient of the size of segment s and its download time $download\_time_s$. After the bandwidth is estimated, the



(a) Liveness (b) Number of dropped packets

Figure 5.18: Performance of a limited TCP congestion window (Cubic)

strategy finds the sizes of the next segment using an HTTP HEAD request and chooses to download the segment in the highest bitrate (quality) possible so that the estimated download time does not exceed one segment duration.

In the following subsections, we compare a quality adaptive scenario using a system with the default settings from Section 5.5 (*unmodified*) with systems that include modified settings over a 30 minute playback. Here, the startup times are in general lower than in the previous section since the first segment is downloaded in a low quality to ensure a fast startup like in most of the commercial systems. We focus therefore only on the user experience metrics liveness (smooth playout) and segment bitrate (video quality). Note that a segment can only be encoded when all (video) data in it is available, i.e., the best achievable liveness is one segment duration.

5.10.1 Alternative Congestion Control Algorithms

In the previous sections, we saw that the Vegas congestion control is slowing down connections' throughput to reduce packet losses, which leads in general, as observed in Figure 5.20(a), to much lower quality than that achieved with Cubic (for quality coding see Figure 5.19). However, because of its very careful congestion control, Vegas is not causing the clients to have many late segments. The liveness (and thus video playout) is therefore almost perfect without any hiccups. On average, clients lag only about 2 seconds behind the live stream independent of their number. In case of Cubic, the clients "risk" more to download higher quality segments and therefore would need to buffer at least 12 seconds for a smooth playout for small client numbers. When the number of clients rises, the competition for resources increases, forcing every client to select a lower quality and thereby improving liveness. Thus, Vegas should be considered if liveness is the goal, and the quality is secondary, or if clients cannot afford buffering.

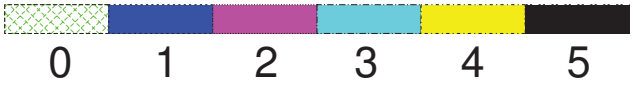
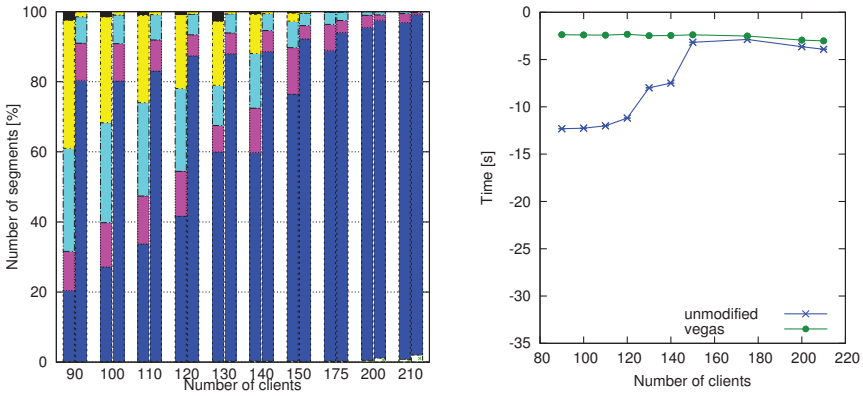


Figure 5.19: Quality coding in figures from low (0) to high (5)



(a) Segment quality (unmodified left bar)

(b) Liveness

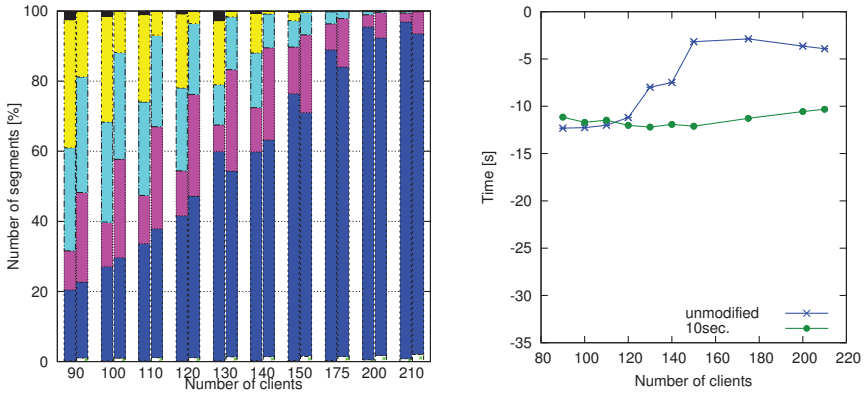
Figure 5.20: Alternative congestion control: Cubic vs. Vegas

5.10.2 Increased Segment Duration

In Section 5.7, we compared short segments with long segments from the system perspective where longer segments give TCP more time to reach its operating state. However, the results showed that 2-second segments lead in general to better performance than 10-second segments. The question answered in this section is how this impacts the video quality. In this respect, Figure 5.21(a) shows that the relative number of high quality segments is lower than in the 2-second case. The liveness seems also worse, but it is actually quite good for 10-second segments. The clients are just one 10-second segment behind the live stream. This means that the clients need to buffer only one segment, that is 10 seconds long, for a smooth playout. Nevertheless, it seems to be more efficient to use 2-second segments – both from the system and the user perspective. This raises the question if even shorter segments should be used, but this is discouraged by current research [107] that states that a very frequent changing of quality leads to unpleasant visual artifacts in the playout.

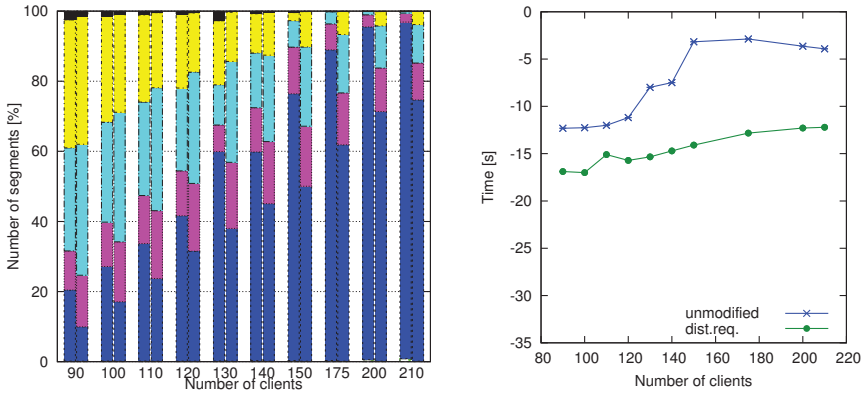
5.10.3 Requests Distributed over Time

Section 5.8 showed a better system performance when distributing the requests for a segment over an entire segment duration. Figure 5.22(a) shows the respective quality improvement when the requests are distributed as in the previous section. The quality improvement is obvious, especially, when the number of clients is high. However, we need to point out that



(a) Segment quality (unmodified left bar) (b) Liveness

Figure 5.21: Segment lengths: 2 vs. 10 seconds



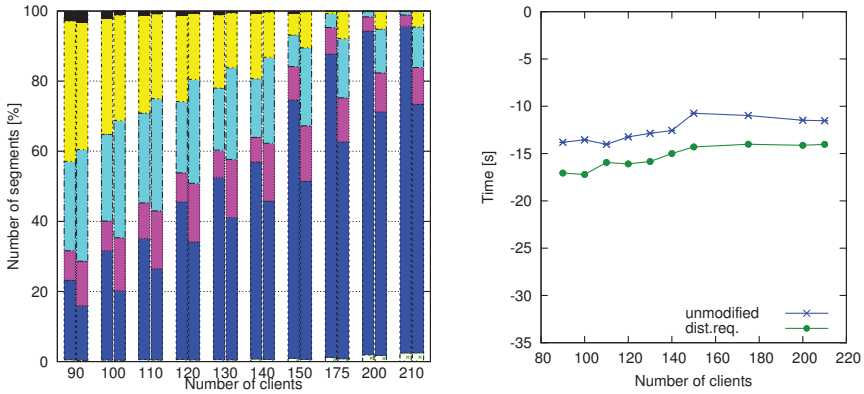
(a) Segment quality (unmodified left bar) (b) Liveness

Figure 5.22: Request distribution (1 segment buffer)

the liveness suffers. Figure 5.22(b) shows that because of lower quality, the strategy without any modifications is able to provide the clients with a more 'live' stream.

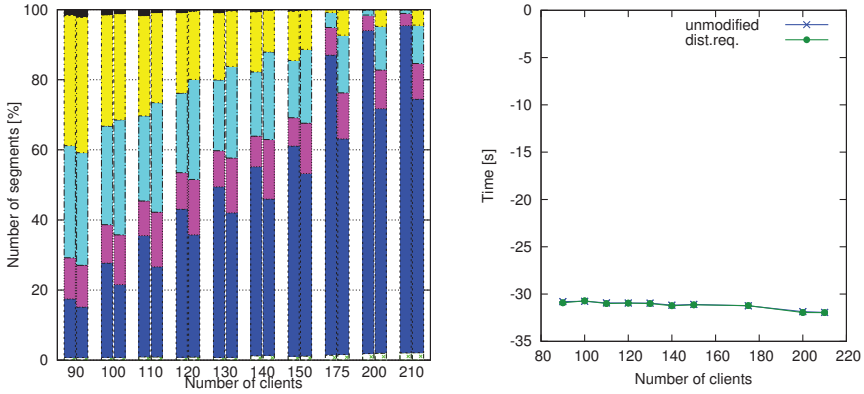
In practice, however, this would be handled by allowing buffering, which does affect liveness, but fixes interruptions in playout. Thus, a observed liveness of -16 seconds means the client would require at least a 14 seconds buffer in order to play the stream smoothly without any pauses. For live streaming, this means that a client cannot request the most recent segment from the server when it joins the stream, but has to request an older segment to have a chance to fill its buffer before catching up with the live stream and so entering the download-and-wait cycle.

Figure 5.23 shows the quality improvement and liveness when clients use 5 segment (10 second) buffers. We observe that the quality gain of request distribution is preserved



(a) Segment quality (unmodified left bar) (b) Liveness

Figure 5.23: Request distribution (5 segment buffer)



(a) Segment quality (unmodified left bar) (b) Liveness

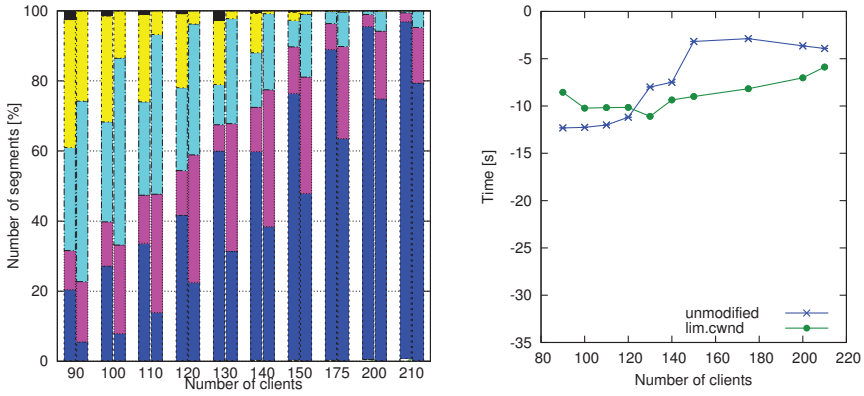
Figure 5.24: Request distribution (15 segment buffer)

while also improving the liveness. Moreover, Figure 5.24 shows that the trend of improved quality continues with a buffer of 15 segments (30 seconds) and additionally the liveness stays almost constant. We conclude therefore that distributing requests is important especially when client buffers are already in place.

5.10.4 Limited Congestion Window

Restricting the client's TCP congestion window and so distributing the download over a longer time has positive effects not only on the server and bottleneck resources (Section 5.9), but also on quality as shown in Figure 5.25(a). The interesting thing about reducing the congestion window is that for small numbers of clients both the liveness and the quality improve. As the number of clients grows, the quality of an unmodified adaptation strategy degrades

quickly, improving the liveness as shown in Figure 5.25(b). The modified settings prevent a single client to steal too much bandwidth from other clients and so cause their downloads to finish too late (thus decreasing their liveness). The liveness is kept about the same irrespective of the number of clients, which is a good thing for practical implementations when the client buffer size must be chosen upfront.



(a) Segment quality (unmodified left bar) (b) Liveness

Figure 5.25: Limiting the congestion window (Cubic)

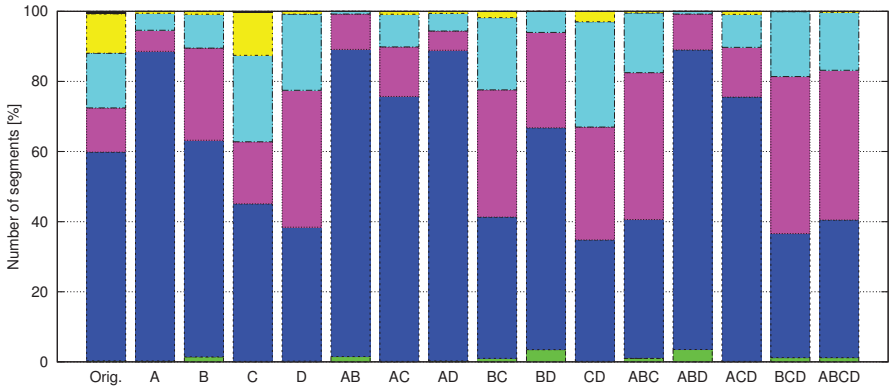
5.11 Combination of Alternative Settings

In this section, we outline and compare the effect of 16 different setting combinations (see Figure 5.26) using 140 clients as a representative example for segment quality distribution. It is again important to note that the graphs must be evaluated together, i.e., a gain in quality (Figure 5.26(a)) often means reduced liveness (Figure 5.26(b)) and increased packet loss (Figure 5.26(c)).

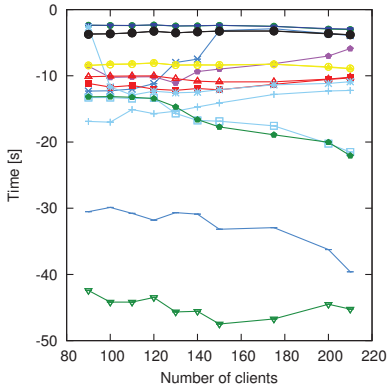
As seen above (Section 5.10.1), changing congestion control, i.e., replacing Cubic with Vegas, results in no or very mild loss that leads to much better liveness. We can observe that Vegas is able to avoid queue overruns also in combinations with the other modifications, which is good. On the other hand, using Vegas the clients do not probe for higher quality as aggressively as when using Cubic and therefore the quality suffers when Vegas is used.

Furthermore, 10-second segment combinations show lower quality than the same combinations with 2-second segments. The only exception is Vegas used with distributed requests, but it is still worse than other combinations. So even in combinations, 10-second segments do not show superior performance over 2-second segments as could be guessed based on the fact that the on periods are longer. Additionally, the liveness is naturally smaller using longer segments because of longer startup times.

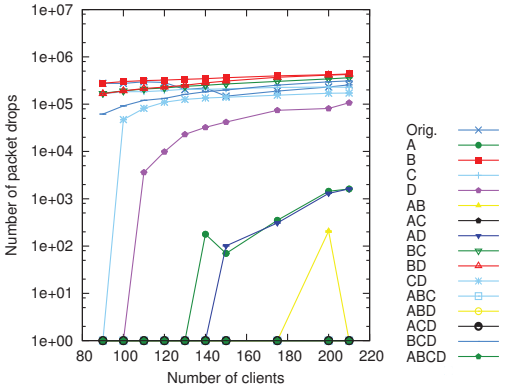
Moreover, in general, combinations with distributed requests result in higher quality, however, the higher quality is paid by decreased liveness in most of the cases. Finally, a congestion window limitation does not seem to have influence on liveness or quality apart from the



(a) Quality - 140 clients



(b) Liveness



(c) Drops

Figure 5.26: Performance of combined settings
(A=Vegas, B=10 second segments, C=distributed requests, D=CWND limitation)

case when it is the only setting modified, or when used with distributed requests in which case it performs better in terms of liveness, especially for small client numbers. Thus, there are several combinations that can be used, but it is usually a trade-off between liveness (which can be fixed by buffering and a startup latency) and video quality.

5.12 Client Request Strategies

A client request strategy is the way in which a client requests segments. The choice of a strategy is a client-side decision only. In this section we investigate the impact of different client request strategies on the server performance when the bottleneck is at the server.

The content of this section is based on [98]. Please note that paper [98] uses the alternative term for liveness, e2e delay. For consistency with previous discussions we replaced the e2e delay term with liveness: $e2e\ delay = -liveness$

5.12.1 The Segment Streaming Model

Figure 5.27 depicts the essentials of the model used in the following sections. At time t_i, a segment i has been encoded, uploaded and made available. Like in several commercial systems, the segment playout duration [58] is constant and equals $t_{i+1} - t_i$. The letter **A** denotes the point in time when a user starts streaming. We call this time the client arrival time.

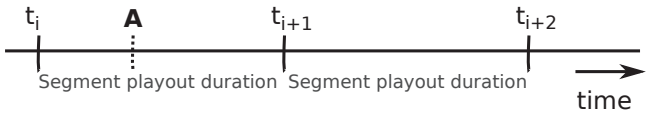


Figure 5.27: The segment streaming model

5.12.2 The Option Set of a Strategy

We considered four important options for a segment request strategy.

First Request Option (P_{fr})

At client arrival **A**, the client requests segment i, which can be the latest available segment or the next that will become available, i.e., t = 0 or 1 in Figure 5.27. We do not consider $i < 0$ because it implies low liveness and $i > 1$ because it implies high start up delay (the client needs to wait until the segment becomes available).

Playout Start Option ($P_{playout}$)

A client can start the playout immediately after the first segment is downloaded or delay the playout. To avoid low liveness already in the design of a request strategy, we limit the playout delay to at most one segment duration.

Next Request Option (P_{nr})

A client can request the next segment at several points in time. We consider two of them. A client can send the request some time before the download of the current segment is completed (download-based request) or send the next request some time ϵ before the playout of the currently played out segment ends (playout-based request).

Deadline Miss Handling Option (P_{miss})

When a segment is downloaded after the previous segment has been played out completely (a deadline miss), the client can skip the first part of the downloaded segment equal to the deadline miss and keep the current liveness or start the playout from the beginning of the segment

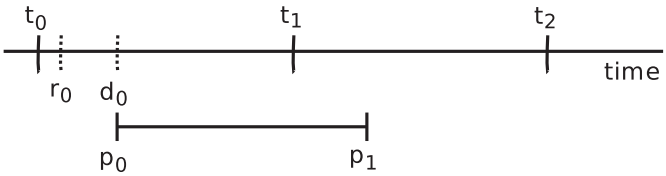


Figure 5.28: Immediate playout start with video skipping

decreasing the liveness. To reduce the complexity and make the results easily comparable, we decided to use only one quality adaptation algorithm. Here, the first segment is retrieved at lowest quality, and the quality of all other segments is based on the download rate of the previous segment and the time available until segment's deadline. To compensate for small bandwidth variations, the algorithm chooses the quality so that the download ends some time before the segment's deadline.

5.12.3 Reduction of Option Combinations

Each option (P_{fr}, $P_{playout}$, P_{nr}, P_{miss}) has two possibilities, resulting in 16 possible combinations. However, some combinations can be eliminated.

Figure 5.28 illustrates a case when $P_{playout}$ is set to immediate playout and P_{miss} to video skipping. Segment 0 is requested at r_0, and the download is finished at d_0. The playout starts at $p_0 = d_0$, and p_0 sets all consecutive deadlines (p_1, p_2, ..) implicitly. The time it takes to download the first segment which is downloaded at the lowest quality is given by $d_0 - r_0$. We see that $d_0 - r_0 \approx p_i - t_i$ which means that, under the same network conditions, the second segment can only be downloaded at the lowest quality, and the server is going to be idle between d_1 and t_2. This applies to all consecutive segments because P_{miss} is set to video skipping. We therefore ignore combinations where P_{miss} is set to video skipping and $P_{playout}$ is set to immediate playout.

By design, every deadline miss leads to a liveness lower than one segment duration, if $P_{playout}$ is set to delay the playout to the next t_i and P_{miss} is set to liveness decrease. Because these options lead to a very low liveness, we ignore combinations that have P_{miss} set to liveness decrease and $P_{playout}$ set to playout delay.

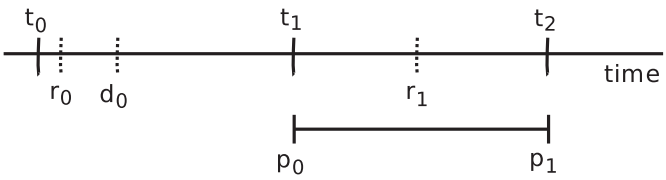


Figure 5.29: Delayed playout, video skipping and playout based requests

Figure 5.29 illustrates a scenario where P_{nr} is based on the playout time, P_{miss} is handled by video skipping, and $P_{playout}$ is delayed until the time next segment becomes available. All

clients that download their first segment between t_0 and t_1 start their playout at t_1. Given this, all consecutive deadlines are fixed at t_i, $i > 2$. All clients request the next segment at the same time (P_{nr}), e.g., r_1 in Figure 5.29. Consequently, the server is idle between t_1 and r_1. The resources wastage is apparent, and therefore, we ignore combinations with these options.

Out of the 6 remaining strategies, 4 decrease the liveness in case of a deadline miss (P_{miss} set to the liveness decrease). 2 of these 4 strategies wait with the first request until a new segment becomes available (P_{fr} set to delay the first request). Even though this leads to a very high initial liveness, it leads also to many deadline misses in the beginning of a streaming session, because the clients essentially synchronize their requests at t_i. This leads to bad performance, as we show later and the result are many user annoying playback stalls. We therefore do not further evaluate these two strategies.

We discuss the 4 remaining strategies in the following section, which we later evaluate with respect to the streaming performance.

Table 5.2: Evaluated strategies

Strategy	P_{fr}	$P_{playout}$	P_{nr}	P_{miss}
$MoBy$	immed.	immed.	download b.	decrease liveness
$MoVi$	immed.	immed.	playout b.	decrease liveness
$CoIn$	immed.	delayed	download b.	skip video
$CoDe$	delayed	delayed	download b.	skip video

5.12.4 Liveness Decreasing Strategies

The two remaining liveness decreasing strategies focus on keeping the liveness as high as possible, but do not delay the first request. We call them Moving Liveness Byte Based Requests ($MoBy$) and Moving Liveness Playout Based Requests ($MoVi$).

$MoBy$ is illustrated in Figure 5.30. A client requests the latest segment that is available at the time of its arrival **A**, i.e., segment t_0 in Figure 5.30. The playout starts immediately after the segment download is finished, $p_0 = d_0$. The next segment request is sent to the server when all but $link\ delay * link\ bandwidth$ bytes of the currently downloaded segment are fetched. This pipelining of requests ensures that the link is fully utilized [70]. Please note, that in Figure 5.30, the next segment is not available at a time that would allow request pipelining. Therefore, it is requested later when it becomes available[7].

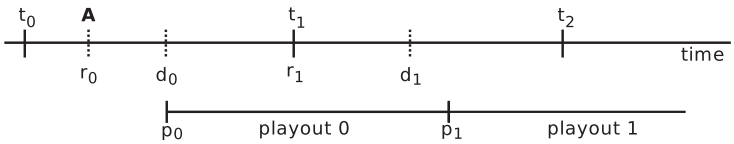


Figure 5.30: Strategy $MoBy$

[7]Segment availability can be checked by consulting the manifest file [58].

Figure 5.31 illustrates the $MoVi$ strategy. The only difference between $MoVi$ and $MoBy$ is that $MoVi$ sends the next request when there are $\frac{p_1-p_0}{2}$ seconds of playout left. This leads to a more evenly distributed requests over $[t_i, t_{i+1}]$ as shown in the result section.

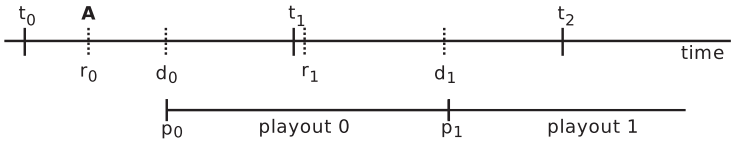


Figure 5.31: Strategy $MoVi$

5.12.5 Constant Liveness Strategies

The last two strategies of the remaining 4 strategies keep a constant liveness of one segment duration throughout a streaming session. If a deadline is missed, the first part of the segment equal to the deadline miss is skipped. Both of these strategies request a segment when it becomes available and their request synchronization leads, in theory, to fair bandwidth distribution among all clients. We call these two strategies Constant Liveness Immediate Request ($CoIn$) and Constant Liveness Delayed Request ($CoDe$).

Strategy $CoIn$ is illustrated in Figure 5.32. A client first requests the latest segment on the server at $r_0 = A$. The next segment is requested at t_1. The liveness is fixed and equals $t_{i+1} - t_i$, i.e., a segment that becomes available at t_i is presented at t_{i+1}. This means that the client has $t_{i+1} - t_i$ seconds to download the next segment.

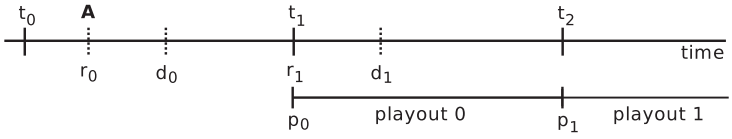


Figure 5.32: Strategy $CoIn$

Figure 5.33 illustrates strategy $CoDe$. The difference between $CoIn$ and $CoDe$ lies in the first request. $CoDe$ delays the first request until the next segment becomes available, i.e., $r_1 \neq A$. $CoDe$ forces all clients to send their requests at the same time starting with the first segment (in contrast to $CoIn$). All the strategy properties are summarized in Table 5.3. These properties only apply when there are no deadline misses.

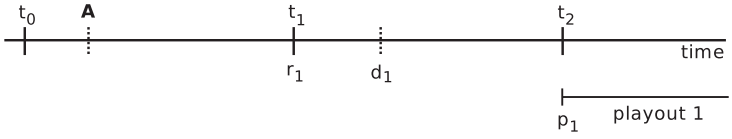


Figure 5.33: Strategy $CoDe$

Table 5.3: Strategy summary (s = segment duration)

Strategy	Startup delay	Liveness	Available download time
$MoBy$	$d_0 - t_0$	$d_0 - t_0$	$d_0 - t_0$
$MoVi$	$d_0 - t_0$	$d_0 - t_0$	$\leq 0.5s$
$CoIn$	$\mu = 0.5 \times s$	s	s
$CoDe$	$\mu = 1.5 \times s$	s	s

5.12.6 Emulation Setup

Since large-scale real-world experiments require both a lot of machines and users, we performed experiments in our lab using an emulated network with real networking stacks. Our hardware setup is shown in Figure 5.34. We used Linux OS (kernel v2.6.32) with Cubic TCP congestion control with SACK and window scaling on. For traffic shaping, we used the HTB qdisc [73] with a *bfifo* queue on the egress interfaces of the client and server machine. The queue size was set to the well known rule-of-thumb $RTT * bandwidth$ [108]. The queue represented a router queue in our approach. The client machine emulated requests from clients and the server machine emulated a webserver[8].

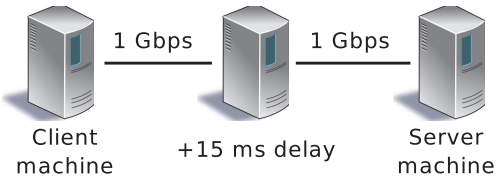


Figure 5.34: Emulation setup

For video quality emulation, we used low, medium, high and super segment qualities with fixed segment sizes of 100 KB, 200 KB, 300 KB and 400 KB, respectively (see Table 5.1). The segment duration was 2 seconds as proposed by [58, 107].

We ran two sets of experiments. The first set, called *short sessions scenario*, included 1000 clients, each downloading 15 segments, which equals approximately the duration of a short news clip. The client interarrival time was modeled as a Poisson process with 100 ms interarrival time (equals 600 client arrivals per minute). The second set included 300 clients each downloading 80 segments, which represent an unbounded live stream. We used the same Poisson process for interarrival times. This set is called *long sessions scenario*. To examine the influence of arrival time, we reran all experiments also with constant interarrival time of 100 ms.

The server served a peak of about 300 concurrent clients in both scenarios. The short sessions scenario was limited by the interarrival time of 100 ms and the number of segments per client. The long sessions scenario was limited by the number of clients.

[8] All data was served from memory to avoid non-networking factors like disk I/O speed.

The emulation program logged for each client the time the client joined the stream and left the stream, as well as for each segment, its quality and its presentation time. Further, it also logged the bandwidth consumption between the server and the client machine with a resolution of 100 ms.

We evaluated four bandwidth limitations restricting the maximal quality clients could download for both scenarios: 40 MB/s, 55 MB/s, 65 MB/s and 80 MB/s. The examined bandwidths provided each client (under optimal and fair conditions) with 133 KB/s, 173 KB/s, 217 KB/s and 267 KB/s respectively.

We repeated each experiment 20 times and we collected our results when the system was in a stable state. We consider the system to be in a stable state when the number of active clients over time is about constant, i.e., client arrival rate equals client departure rate. For the short sessions scenario, this means examining segments 30 to 50, and for the long sessions scenario, segment 20 and beyond.

5.12.7 Request Strategies Impact on HTTP Segment Streaming

We were interested in the interaction of concurrent segment downloads and its impact on video quality and liveness. We therefore analyzed the goodput (and the corresponding video quality) of each strategy as well as the parallelism induced by each strategy.

Strategy goodput

Figure 5.35 and Figure 5.36 show the collected goodput statistics for short and long sessions scenarios. Each box plot value[9] represents the sum of bytes received by all clients during one experiment divided by the stream's duration.

The goodput is, like a regular bulk download, reduced by the protocol header overhead, and additionally by the time the client spends waiting until the next segment becomes available. For short sessions, Figure 5.39[10] shows the number of concurrently active clients over time. We see that the number of clients drops (clients are idle waiting) towards the end of each 2-second interval. In the next section, we further discuss the effects and consequences of parallelism.

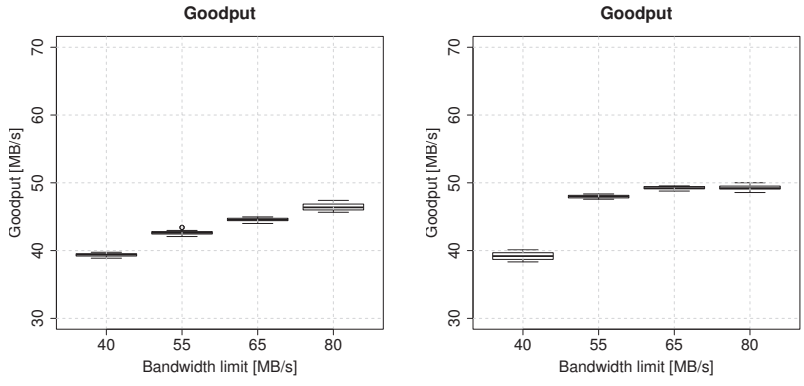
The goodput of strategies $CoIn$ and $CoDe$ increases as more bandwidth becomes available. The reason for this behaviour is that these strategies keep a constant liveness and so provide a full segment duration of time for segment download.

Figure 5.40 and Figure 5.41 show the improvement in terms of segment quality. The height of the graph corresponds to the total number of segments downloaded. The number of segments of each quality is represented by the grey scale starting at the top with the lightest color representing the super quality. The trend is clear: the more bandwidth, the higher the quality.

Strategies $MoBy$ and $MoVi$ do not quite follow the same trend as $CoIn$ and $CoDe$ though. $MoBy$ and $MoVi$ increase the goodput up to the bandwidth limitation of 55 MB/s.

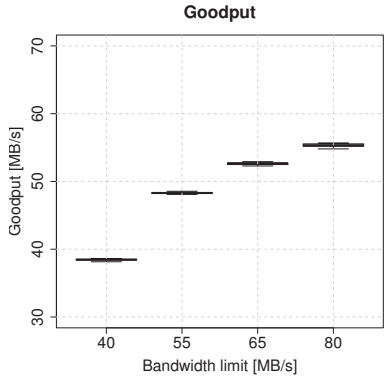
[9] Minimum, maximum, quartiles, median and outliers are shown. Please note that in many cases, these overlap in the graphs.

[10] The same pattern was observed also for long sessions scenario.

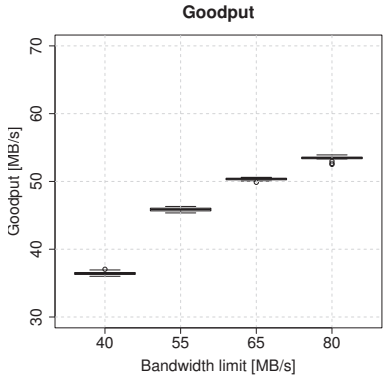


(a) MoBy

(b) MoVi



(c) CoIn



(d) CoDe

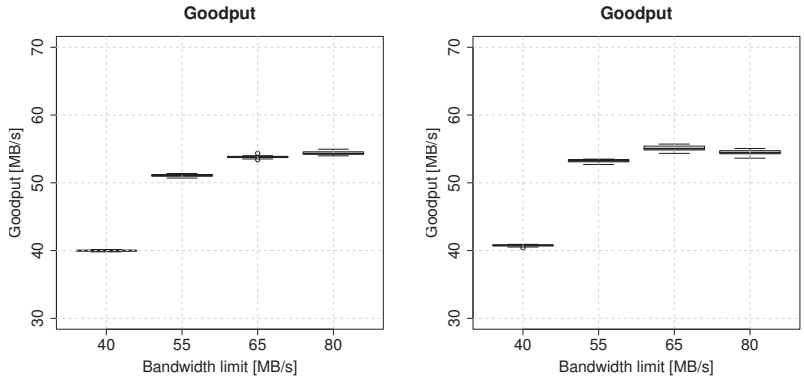
Figure 5.35: Short sessions scenario goodput

The goodput increases slowly or stagnates for higher bandwidths. We can observe the same for segment quality. Figure 5.42 and Figure 5.43 indicates that these two strategies are trading goodput for higher liveness, i.e., the stream is more "live".

It is clear from Figure 5.40 and especially Figure 5.41 that strategy $MoVi$ is able to achieve high quality quickly and still keep a high liveness. The reason lies in parallelism, which we discuss in the next section.

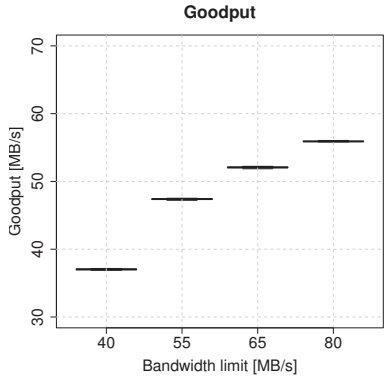
5.12.8 Parallelism and its Consequences

Figure 5.39 shows a representative 20 second snapshot of the number of concurrently active clients. The strategy that stands out is $MoVi$. The maximum number of concurrently downloading clients is about one third compared to the other strategies. Yet, as discussed in Section 5.12.7, $MoVi$ does not suffer severe quality degradation (and it does not discard segments).

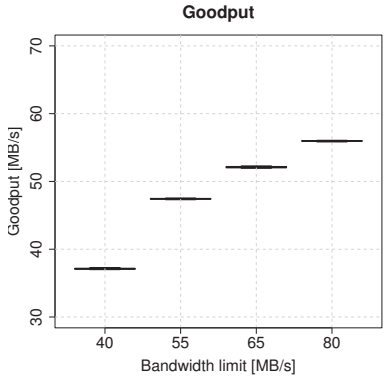


(a) MoBy

(b) MoVi



(c) CoIn



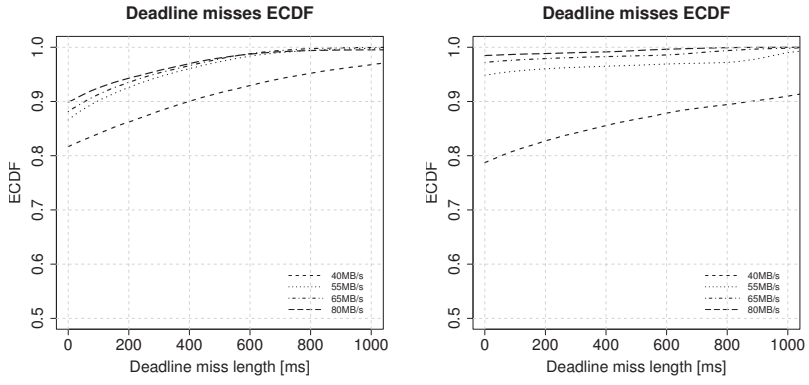
(d) CoDe

Figure 5.36: Long sessions scenario goodput

The mixture of P_{nr} based on playout and $P_{playout}$ set to immediate playout makes $MoVi$ requests distributed according to client arrival distribution, i.e., the requests are distributed over each interval. This is an important difference to the other strategies.

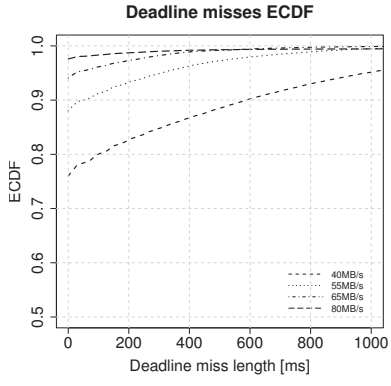
We found that the reason for $MoVi$'s good overall performance is the number of dropped packets by traffic shaping. Figure 5.38 shows a representative plot of the number of packets dropped by the emulated router queue. The queue size is set to $RTT * bandwidth$ and accepts therefore only a certain number of packets. We observed that synchronized requests overflow the router queue much more often than requests distributed over time as is the case for $MoVi$.

Because the P_{nr} option of $MoVi$ is based on playout time, more specifically a client requests the next segment $\frac{p_1 - p_0}{2}$ seconds before the playout of the previous segment ends (Section 2.3), $MoVi$ clients have at most half of the time available to download a segment compared to $CoIn$ and $CoDe$. Yet, they are still able to download the same or higher quality segments (Figure 5.40 and Figure 5.41). Moreover, the liveness is in many cases very high (above a segment duration) (Figure 5.42 and Figure 5.43). The only time the strategy has

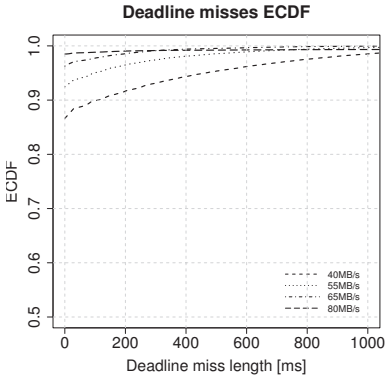


(a) MoBy

(b) MoVi



(c) CoIn



(d) CoDe

Figure 5.37: Short sessions scenario deadline misses' empirical distribution function (ECDF)

a problem is when there is not enough bandwidth available to sufficiently distribute client downloads and the downloads start to overlap too much.

Figure 5.44 shows a box plot of segment download speed as experienced by the clients. The graph includes all bandwidths experienced by all clients for every segment in every experiment run. Because of the small number of concurrent clients, $MoVi$ is able to make more out of the available bandwidth. The other strategies loose more data in competition for space in the router queue.

If a $MoVi$ client determines that enough time is available for downloading higher quality segments, it probes by downloading a higher quality segment at the next opportunity. It then experiences more competition because it enters the download slots of other clients, and withdraws immediately if there is not enough bandwidth. This approach reduces the number of concurrent clients and thus, the pressure on the router. The other strategies can never probe. The requests of active clients are always synchronized. The increased competition leads to more unpredictable bandwidth distribution among clients and makes it harder for the clients

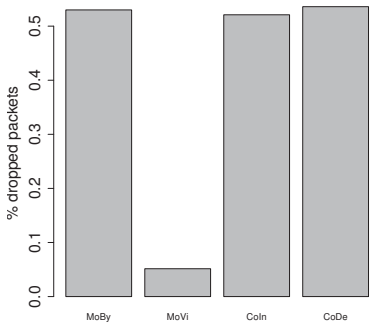


Figure 5.38: Packets dropped by the emulated router queue for 55 MB/s bandwidth limitation

to estimate available bandwidth based on the download of the previous segment.

5.12.9 Deadline Misses and the Bandwidth Fluctuations

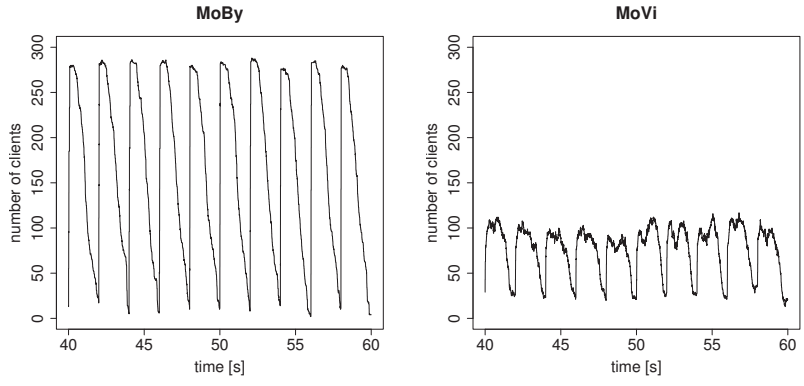
There is, theoretically, only one factor that influences the deadline misses, i.e., bandwidth fluctuation. Under perfect conditions without bandwidth fluctuations, clients never overestimate the bandwidth available, and a deadline miss never occurs.

However, the large number of concurrent streams combined with TCP congestion control creates fluctuations. In the case of $MoVi$, the fluctuations do not have such a big impact because the number of concurrent clients is small (Figure 5.39), and the download bandwidths are high compared to the other strategies. The deadline-miss graphs in Figure 5.37 (long sessions scenario results are very similar) indicate that the $MoVi$ strategy is able to estimate the bandwidth as well or better than the other strategies.

In general for all strategies, the number of deadline misses shown in Figure 5.37 indicates that the bandwidth estimation, and therefore also the quality adaptation, is not always perfect. However, we see that with more bandwidth available, the number of deadline misses decreases.

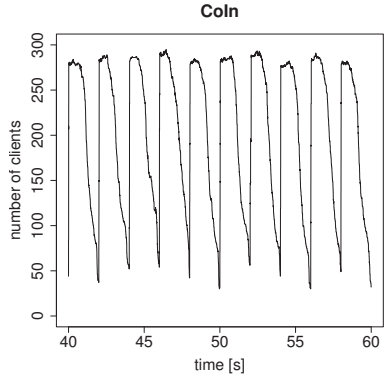
We believe that the reason for deadline miss improvement is the usage of time safety. The adaptation algorithm chooses segment quality so that the download ends at least ts seconds before segment's deadline. The minimal value of ts is the time safety[11]. A deadline miss occurs only if the download time proves to be longer than the estimated download time plus the time safety. To make the impact of the time safety on results as small as possible we used a small value of 150 ms for all scenarios. However, the number of bytes that can be downloaded within the time safety increases with available bandwidth. This way the safety basically grows with bandwidth, which results in fewer deadline misses as the bandwidth grows. In this respect, one should choose a larger time safety if more bandwidth fluctuations are expected. One could also adjust the time safety dynamically based on the observed bandwidth fluctuations.

[11]The function of the time safety is similar to the function of the "slack parameter" in [96]

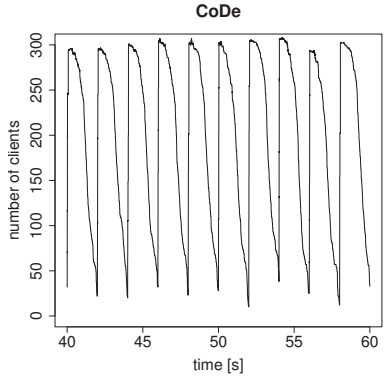


(a) MoBy

(b) MoVi



(c) CoIn



(d) CoDe

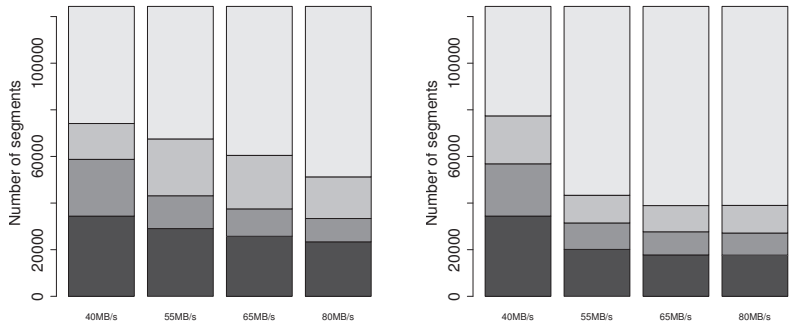
Figure 5.39: Concurrent downloads in the short sessions scenario (55MB/s)

5.12.10 Influence of Client Interarrival Times Distribution

For reference to the naturally, exponentially distributed interarrival times we reran short and long session scenarios with constant client interarrival times. Our results showed no major differences. We observed very similar results for segment quality distribution, number of concurrent clients over time, liveness and deadline misses.

5.12.11 Implications for Multi-Server Scenarios

There are implications of our findings for CDNs based on Domain Name System (DNS) load balancing. Figure 5.45 shows the principle of DNS load balancing in CDNs. When a client wants to download a segment, it issues a request to its local DNS server. If the information is not cached, the request is forwarded to the root DNS server. The root DNS server redirects the local DNS server to a CDN provider's DNS server. This server finds the best server



(a) MoBy

(b) MoVi

(c) CoIn

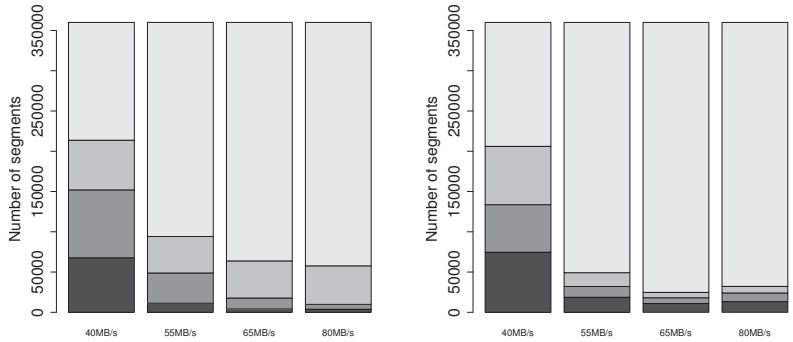
(d) CoDe

Figure 5.40: Short sessions quality distribution of downloaded segments (from super quality at the top to low quality at the bottom)

available and replies with the server's IP address. The local DNS server caches and forwards the information to the client. The client downloads the data from the provided server.

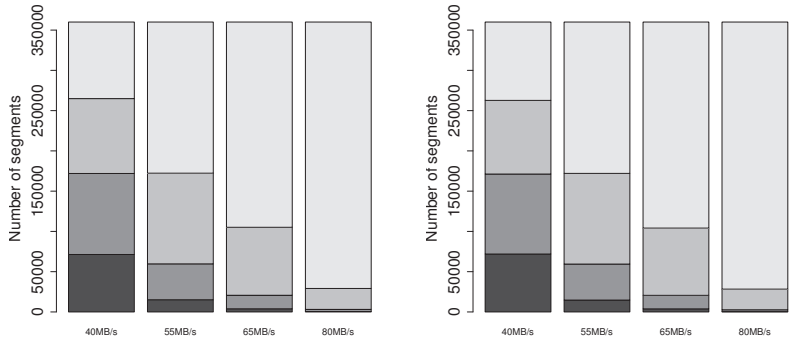
A small time to live (TTL) of the DNS cache entry leads to frequent DNS requests. On the other hand, a large TTL leads to less load balancing opportunities for the CDN. Therefore, a compromise between granularity of load balancing and the DNS server load is required. Akamai CDN, for example, uses a default value of 20 seconds [109]. In fact, the TTL value is not enforcible, and providers might choose a higher value in order to provide faster response times or to reduce the risk of DNS spoofing [110].

DNS load balancing relies on separate connections for each web object. However, separate connections are inefficient in case of live HTTP segment streaming. To save the overhead associated with opening a new TCP connection, it is reasonable for a client to open a persistent TCP connection to a server and reuse it for all segment requests. Requests are sent over this connection as segments become available. This differs from a persistent connection used



(a) MoBy

(b) MoVi

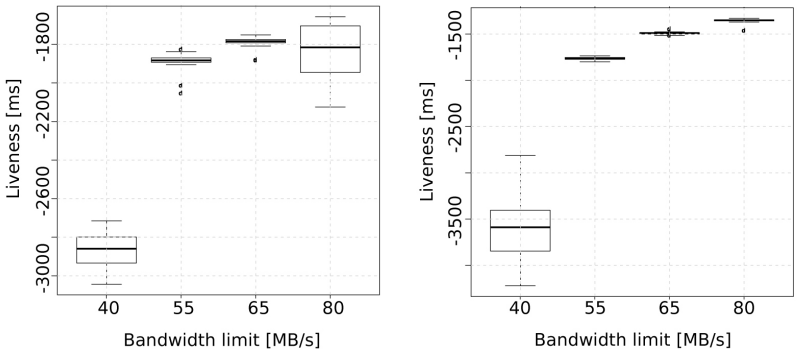


(c) CoIn

(d) CoDe

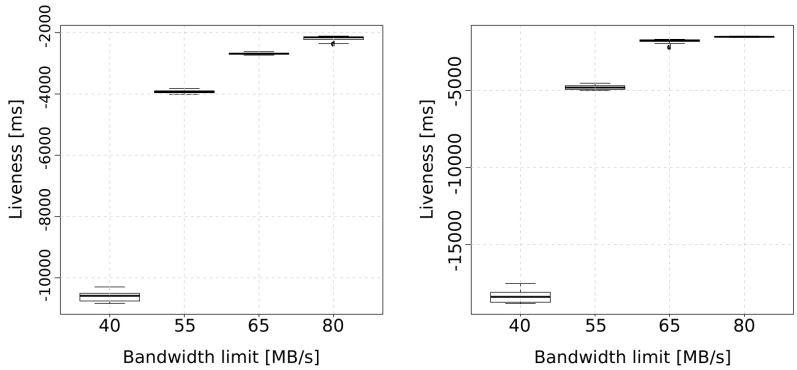
Figure 5.41: Long sessions quality distribution of downloaded segments (from super quality at the top to low quality at the bottom)

by web browsers. Firstly, the connections last longer (a football match takes 90 minutes). Secondly, clients do not download data continuously (Figure 5.39). This makes it difficult to estimate the server load by algorithms that assume that all clients download data continuously. Moreover, the adaptive part of the streaming leads to jumps in bandwidth requirements rather than to smooth transitions. These are all reasons that make it harder for conventional DNS load balancing to make an accurate decision. An HTTP redirection mechanism in the data plane could solve this. However, to the best of our knowledge, this mechanism is not implemented by current CDNs. The insufficiencies of server-side load balancing for live adaptive segment streaming can be overcome by client-side request strategies.



(a) MoBy (b) MoVi

Figure 5.42: Short sessions scenarios liveness (note: liveness y-axes have different scale)



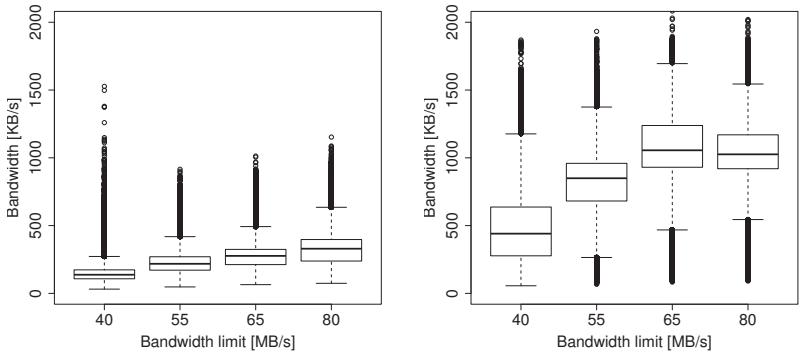
(a) MoBy (b) MoVi

Figure 5.43: Long sessions scenarios liveness (note: liveness y-axes have different scale)

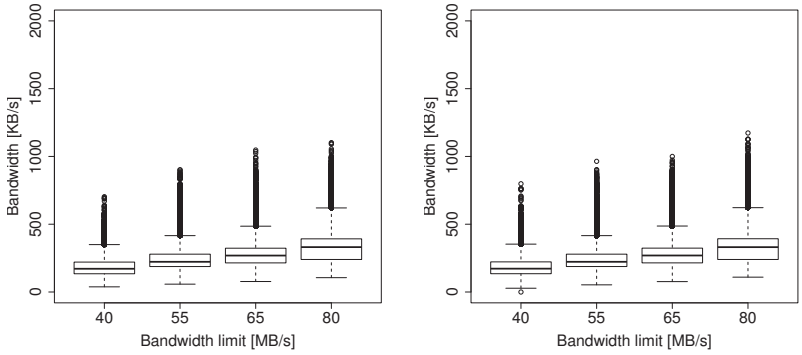
5.13 Conclusions

We have first studied the performance of HTTP segment streaming in a multi-user server-side scenario. It turns out that because of the different traffic pattern of HTTP segment streaming, the server is able to handle much fewer clients than its network bandwidth would allow. We proposed and tested four different methods (and their combinations) ranging from network-layer to application-layer modifications to improve the situation. Additionally, we re-evaluated these methods also in a scenario where clients used a quality adaptation strategy.

Our results show that other congestion control algorithms like Vegas avoid queue overruns, also in case of short (up to 2 seconds) connections. Therefore, the use of Vegas leads to very few late segments, which means an increased liveness with less buffer underruns. On the other hand, Vegas's careful approach leads to lower the quality. Moreover, we do not find



(a) MoBy (b) MoVi



(c) CoIn (d) CoDe

Figure 5.44: Client segment download rates in the long sessions scenario

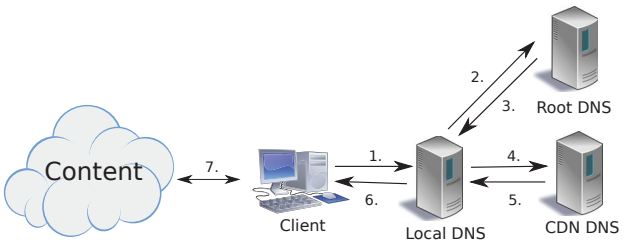


Figure 5.45: CDN DNS load balancing

a convincing reason for using 10-second segments. In a single-user scenario, it uses the congestion window better, but in a multi-user scenario with competition for the resources neither the video quality nor the liveness are improved over 2-second segments. We further observe that the distribution of requests over time leads to very good results in terms of quality and

in case of client-side buffers also liveness. Limiting the congestion window contributes to better fairness among the connections and gives a better bandwidth estimation. This results in good quality and stable liveness for different numbers of clients, but only if used as the only modification.

Our evaluation of client-side request strategies for live adaptive HTTP segment streaming shows that the way the client requests segments from a server can have a considerable impact on the success of the strategy with respect to achievable liveness and video quality. The chosen strategy influences not only the video quality and liveness, but also the efficiency of bandwidth usage.

Starting with the assumption that it is desirable for clients to request video segments from a server that provides live content as soon as they become available, we examined three strategies that do request segments as soon as they become available and a fourth strategy that requests them later. The request pattern of the synchronized strategies leads to synchronized requests and server responses. The unsynchronized strategy does not initiate downloads based on segment availability but on the playtime of earlier segments.

While it was obvious that synchronized requests lead to competition for bandwidth, we have shown that this competition leads to a severe amount of bandwidth wastage. Synchronized clients can only base their adaptation decision on average goodput in a complete segment length, because they can never avoid competition with other clients. With strategies that do not synchronize requests, on the other hand, bandwidth wastage is avoided because the number of concurrently active clients is much lower. Clients can probe for higher qualities, and withdraw when their bandwidth demands increase the amount of competition and therefore the segment download time. We have finally considered how these observations could help scheduling decisions in a multi-server scenario.

Chapter 6

Conclusion

In the recent past, video streaming delivery went from using unreliable protocols like UDP to using a reliable transport protocol, TCP. The first approach using TCP for video delivery was progressive download. Progressive download means that the client downloads and plays a video at the same time. The problem that arises is when the download is too slow and the playout caches up with the download and must be interrupted until more data is downloaded. This problem can be avoided if the video bitrate can be adjusted according to the varying network bandwidth. The solution to this problem is adaptive HTTP segment streaming, which splits a video into multiple video segments and encodes each of these in different bitrates (qualities). The client then chooses, based on its currently available bandwidth, at which bitrate it can download a segment. The segments are encoded in a way that enables the client to combine them seamlessly without the user noticing the transitions between segments. Recently, adaptive HTTP segment streaming has grown to be the dominant streaming technology for live and on-demand video delivery in the Internet. The technology has potential far beyond the traditional linear delivery, but there are still performance issues to address. It is the potentials and the performance issues related to adaptive HTTP segment streaming that this thesis explored.

6.1 Summary and Contributions

When searching with nowadays popular video search engines like YouTube [10] or Dailymotion [111] only complete videos are returned. For more fine-grained results the user must search the videos manually. This is partly a limitation of the search engines, which do not return sub-video level results. However, it is also a limitation of the streaming technology, progressive download, that cannot deliver videos combined from multiple video parts in an efficient and seamless manner. Adaptive HTTP segment streaming removes this limitation. In Chapter 2, we described how to encode video segments in a way that allowed video segments to be played out seamlessly (without interruption in playout) even if the video segments came from different videos or had a different bitrate. As a proof of concept we implemented two prototypes. The first prototype enhanced the search results of Microsoft Enterprise Search Platform (ESP) [112] with video. In the enhanced version called vESP [54], the user not only got to browse through the presentation slides that matched the search query, but could

also watch video clips of the presenter synchronized with the slides. Particularly, the need to manually search through the video of the whole presentation was removed. Additionally, the user had the possibility to put slides on a deck and play an automatically generated video playlist for the deck of slides. We performed user studies to evaluate the user response to the new video feature of the system. Our user study showed that the new feature significantly improved the user experience (Section 2.3.2) compared to the original system with only the slide preview. Our second prototype enhanced the results of a football clips search engine. The search engine delivered precise timing of events like "sliding tackle" or "goal" within a football game video, however, the video streaming component was not able to deliver parts of the video (similarly to a YouTube like system). This changed with our adaptive HTTP segment streaming integration. Instead of showing the entire 90 minute video of the game, only the precise moment of interest was presented. Furthermore, we noticed that the automatic analysis of a football game video sometimes resulted in imprecise clip boundaries, e.g., a goal clip did not begin at a time suitable for the viewer who would like to see the action. Therefore, we implemented an uncomplicated learning algorithm based on exponential averaging that was able to learn the preferred event boundaries taking into account previous (historic) user actions (rewind, fast forward). Similarly to vESP, we also evaluated the user experience gains with a user study and observed that the improvement was statistically significant (Section 2.3.3).

Even though non-linear video playout systems like the vESP are interesting, already the linear playout systems are challenging to designed and scale up. To get a better idea of the scaling of a real-world adaptive HTTP segment streaming system, we analysed logs provided by a Norwegian streaming provider Comoyo [16] (Chapter 3). Our analysis shown that about 3% of the network data could have been saved if more HTTP proxy caches or a different bitrate adaptation strategy were used. We also observed that the amount of downloaded data as reported by the clients was much higher (more than 2 times) than the amount of data served by the origin server. Moreover, the number of distinct IP addresses as seen by the origin server was significantly lower than the number of distinct IP addresses as reported by the analytics server, which all clients report to. This means that a significant percentage of clients downloaded their segments from places different than the origin server. In other words, this means that the caching of segments really happens and works, because the video segments did come from somewhere and it was not the origin server. Our next observation was that about 90% of the same segment requests happened within a 10 second time window (note that we analysed live streaming of football games). This gives a great potential for caching, i.e., an HTTP proxy cache needs to cache a segment only a very short time to achieve a good cache hit ratio. The temporal dependency of requests also means that servers serving live content like this can keep a few segments in memory and so almost completely eliminate expensive I/O to disk. The elimination of disk I/O as a performance bottleneck, leaves the network throughput as a possible bottleneck.

In Chapter 4, we proposed a way to eliminate a client-side network bottleneck through the use of multiple client interfaces. Particularly, we described a new pull-based download scheduling algorithm for using multiple interfaces (e.g., WLAN and 3G) to increase the bitrate (quality) of segments that the client can download. The proposed scheduling algorithm

efficiently distributed the download of partial segments over multiple interfaces and so increased the effective total bandwidth available to the client. The assignment of partial segments to interfaces happened based on the current utilization of each available interface. Our experiments showed that the scheduler was able to achieve next to perfect throughput aggregation, at least in a controlled testbed environment. For real-world networks, we observed that if the bandwidth of the interfaces differed too much (more than a factor of 4), a tradeoff between achievable video liveness and the efficiency of throughput aggregation (achievable bitrate) had to be made, i.e., the client buffer had to be increased to balance out the slower interface. We found for example that a liveness of -10 seconds (the client was 10 seconds behind the live stream) must be accepted for good quality in case of WLAN and HSDPA aggregation where the bandwidth difference is high.

Furthermore, we proposed in Chapter 5 modifications that can help increasing the performance in terms of bitrate and liveness when the network bottleneck is at the server. A bottleneck on the server-side results from multiple streams being active at the same time. In this thesis, we investigated such a set of live streams streaming the same content. We observed that each live stream behaves as an *on-off* source. Particularly, most of the time a stream is actively sending data when a segment is being downloaded and after that waits for the next segment to become available on the server or for the playout to "consume" a segment from the client buffer to make room for the next segment. With such a stream behavior in mind, we evaluated the impact of different modifications on the client-side and some light modifications on the server-side. We evaluated these with simulation and some with emulation. The results showed that TCP Vegas, a delay based congestion control algorithm, is still able to slow down its TCP CWND growth before causing congestion and does therefore not overrun the bottleneck queue even though the connections are short, i.e., 2 seconds. However, the clients that use Vegas are very conservative in their bandwidth estimation and therefore do rarely request high quality segments. We also tried increasing the segment duration from 2 seconds to 10 seconds. Even though our intuition told us that the longer a segment is, the easier it is for TCP to estimate the available bandwidth, we could not find a persuasive reason for the use of 10-second segments over the 2-second segments. The problem with loss based TCP congestion control algorithms (e.g. Cubic or Reno) is that their fairness is first established after some time [82], i.e., even if two TCP connections are started at about the same time, one of them often gets an considerably better throughput during the first seconds. We therefore tried limiting the TCP CWND and so enforce a bandwidth cap on each connection. We saw that the TCP CWND limitation does contribute to better fairness and bandwidth estimation, which in turn results in good quality and stable liveness when the number of clients increases (but only if this modification is used alone). Furthermore, we found that in case of live streaming the clients have a tendency to synchronize their request. This is because every client waits for a new segment to become available and then they all request it at about the same time. We therefore proposed and evaluated a simple, yet effective method to distribute the client requests over time (even more than they are naturally distributed by different RTTs to the server). Using this method, the client remembers the time difference between the time it requested the first segment and the time the segment was uploaded to the server (file modification time). For example, if a segment was uploaded at time 2.0 s and the client requested

it at time 2.1 s, the time difference is 0.1 s. Note that this time difference is different for every client. The client then uses this time difference to offset all of its future requests. Because each client has a different offset the requests stay distributed over time. We found that request distribution leads to improved results in terms of segment bitrate and, if the client has sufficient buffers, also liveness.

The idea of distributing requests came from our evaluation of different ways clients request segments. For example, a client can request the latest segment on the server, or it can wait with the request. It can also request a segment when it becomes available or wait a bit. We proposed another simple algorithm for request distribution, which is based on playout time. Using this algorithm, the client requests a segment when there is only a certain amount of playout data left. In this case, when clients are in different stages of playout, a natural good request distribution follows. We found that strategies that do not lead to client request distribution overflow the router queue more. The overflowing of router queue then leads to packet loss, inefficient bandwidth usage and therefore smaller segment bitrate (quality) and decreased liveness and thus should be avoided.

6.2 Future Research Directions

We presented two non-linear video streaming HTTP segment streaming applications, but there are certainly more application domains that can benefit from the flexibility of video segments. Even though the basic principles are the same, per domain solutions and benefits do vary, and more research, development and standardization is needed.

From the networking perspective, HTTP segment streaming seems to be a good solution, but more empirical data is needed from the streaming providers to better understand the challenges that they are facing. We find that most of the data available is based on performance measurements with single or multiple clients, but there is, to the best of our knowledge, no aggregated data from a streaming provider available, i.e., data that would show how thousands of clients behave when streaming the same content. When such data is available, more accurate models and assumptions can be made about the client arrival times, client request behaviour, bitrate switching etc. With the help of this information, theoretical models can be built that can quickly evaluate the impact of new bitrate adaptation and request strategies, benefiting not only the streaming providers through a higher income to expense ratio, but also their customers.

Last, but not least, the list of modifications evaluated in this thesis to target potential bottlenecks is not exhaustive. There are new technologies appearing which may have a positive impact on HTTP segment streaming and bring more possibilities on how to retrieve segments. Examples include SPDY [113] and TCP Fast Open [77]. SPDY is a new way to improve download speed in the Internet. Among other things, it supports the multiplexing of requests over one TCP connection as well as request prioritization and cancellation. This functionality is interesting for HTTP segment streaming, which may take advantage of it. The client can, for example, request the same segment in two bitrates at the same time giving the higher bitrate low priority. Then, when it is clear that there is enough bandwidth, it might cancel the low bitrate segment and focus only on the high bitrate segment. However, more

importantly the cancellation of requests can eliminate the *on-off* traffic pattern keeping the connection busy at all times (unless the bandwidth is sufficient for the highest bitrate). Elimination of the *on-off* traffic pattern makes the connection bulk again and TCP optimizations for bulk download can be fully effective again. TCP Fast Open allows data to be sent already during the 3-way TCP handshake. This eliminates the initial delay, and if the CWND is large from the beginning of the connection based on historical data, the client can request each segment over a new TCP connection without any delay or a throughput penalty. This makes it for example possible for a load balancer to send each segment request (TCP connection) to a different server.

Appendix A

Publications

This appendix lists all the scientific publications that were produced in the course of this PhD project. Each publication is briefly described, and the individual contributions of the different authors are explained.

A.1 Conference Publications

NOSSDAV 2010 Kristian R Evensen, Tomas Kupka, Dominik Kaspar, Pål Halvorsen, and Carsten Griwodz. Quality-adaptive scheduling for live streaming over multiple access networks. In Proc. of ACM NOSSDAV, pages 21–26, 2010. [69]

This paper describes and evaluates a practical implementation of the multilink framework developed by Kristian Evensen and Dominik Kaspar [72]. Tomas Kupka with Kristian Evensen developed the scheduling algorithm and did most of the programming. Kristian Evensen carried out all the performance experiments, and Dominik Kaspar contributed by writing the related work section and parts of the architectural section. All authors took part in the technical discussions and provided feedback on the textual content and structure.

ICDKE 2010 Pål Halvorsen, Dag Johansen, Bjørn Olstad, Tomas Kupka, and Sverre Tennøe. vesp: A video-enabled enterprise search platform. In Proc. of IEEE ICDKE, pages 534–541, 2010. [54]

This paper describes our efforts of integrating our existing video solution based on HTTP segment streaming with the FAST Enterprise Search Platform. The integration was performed by Sverre Tennøe and Tomas Kupka. The user evaluation was carried out at and by Microsoft. All authors took part in the technical discussions and provided feedback on the textual content and structure.

LCN 2011 Tomas Kupka, Pål Halvorsen, and Carsten Griwodz. An evaluation of live adaptive HTTP segment streaming request strategies. In Proc. of IEEE LCN, pages 604–612, 2011. [98]

This paper describes possible client request strategies for HTTP segment streaming. The paper also evaluates the strategies by emulation. Tomas Kupka designed the request strategies

and wrote the emulation software as well as carried out all the experiments. All authors took part in the technical discussions and provided feedback on the textual content and structure.

LCN 2012 Tomas Kupka, Pål Halvorsen, and Carsten Griwodz. Performance of On-Off Traffic Stemming From Live Adaptive Segmented HTTP Video Streaming. In Proc. of IEEE LCN, pages 405-413, 2012. [99]

This paper evaluates different client and server modifications in terms of HTTP segment streaming metrics (Section 4.2). Modifications are evaluated by simulation. The simulation framework was designed and written by Tomas Kupka. The simulation was performed by the same person. All authors took part in the technical discussions and provided feedback on the textual content and structure.

A.2 Demos at International Venues

ACM MM 2010 Pål Halvorsen, Dag Johansen, Bjørn Olstad, Tomas Kupka, and Sverre Tennøe. vesp: Enriching enterprise document search results with aligned video summarization. In Proc. of ACM MM, pages 1603–1604, 2010. [55]

Sverre Tennøe and Tomas Kupka integrated an existing video streaming in house solution with the FAST Enterprise Search Platform (ESP). The user study was carried out by Microsoft. All authors took part in the technical discussions and provided feedback on the textual content and structure.

A.3 Journal Articles

Multimedia Tools and Applications Dag Johansen, Pål Halvorsen, Håvard Johansen, Haakon Riiser, Cathal Gurrin, Bjørn Olstad, Carsten Griwodz, Åge Kvalnes, Joseph Hurley, and Tomas Kupka. Search-based composition, streaming and playback of video archive content. Multimedia Tools and Applications, 2011. [56]

This journal article is a summary of our work in the domain of HTTP segment streaming applications. All authors took part in the technical discussions and provided feedback on the textual content and structure.

Glossary

AUD NAL Access Unit Delimiter NAL [42].

CDN Content Delivery Network.

CWND TCP congestion window [1].

DCCP Datagram Congestion Control Protocol [34].

DNS Domain Name System.

HTB Hierarchical Token Bucket [73].

HTTP Hypertext Transfer Protocol [19].

IP Internet Protocol [61].

ISP Internet Service Provider.

IW TCP initial congestion window [1].

MDC Multiple Description Coding [38].

MTU The largest protocol data unit that can still pass through the link in one piece.

NAL Network Abstraction Layer [42].

NAT Network Address Translation.

PCR Program Clock Reference [4].

PES Program Elementary Stream [4].

QoS Quality of Service.

RTCP RTP Control Protocol [32].

RTO Retransmission Time-Out [95].

RTP Real-time Transport Protocol [32].

RTSP Real Time Streaming Protocol [33].

RW TCP restart congestion window after an idle period [1].

SCTP Stream Control Transmission Protocol [35].

SPEG Scalable MPEG [40].

SVC Scalable Video Coding [39].

TCP Transport Control Protocol [18].

UDP User Datagram Protocol [17].

VoD Video on Demand.

Bibliography

[1] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control." RFC 2581 (Proposed Standard), Apr. 1999. Obsoleted by RFC 5681, updated by RFC 3390.

[2] Cisco Systems, Inc., "Visual Networking Index." http://www.cisco.com/en/US/netsol/-ns827/networking_solutions_sub_solution.html, May 2013.

[3] "Altus vSearch." http://www.altus365.com/altus_vsearch.php, May 2013.

[4] "H.222.0: Information technology - Generic coding of moving pictures and associated audio information: Systems," *ITU-T recommendation*, June 2012.

[5] "RealPlayer." http://en.wikipedia.org/wiki/RealPlayer, May 2013.

[6] "ActiveMovie." http://www.microsoft.com/en-us/news/press/1996/mar96/actmovpr.aspx, May 2013.

[7] "QuickTime." http://www.apple.com/quicktime, May 2013.

[8] "Windows Media Player." http://www.microsoft.com/mediaplayer, May 2013.

[9] S. Mccanne and V. Jacobson, "vic: A Flexible Framework for Packet Video," in *ACM MM*, pp. 511–522, 1995.

[10] "YouTube statistics." http://www.youtube.com/yt/press/statistics.html, May 2013.

[11] "HBO." http://www.hbo.com, May 2013.

[12] "Viasat." http://www.viasat.no, May 2013.

[13] "TV 2 Sumo." http://sumo.tv2.no, May 2013.

[14] "NRK." http://tv.nrk.no, May 2013.

[15] "Netflix." http://www.netflix.com, May 2013.

[16] "Comoyo." http://www.comoyo.com, May 2013.

[17] J. Postel, "User Datagram Protocol." RFC 768 (Standard), Aug. 1980.

[18] J. Postel, "Transmission Control Protocol." RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1." RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[20] "HTTP Dynamic Streaming on the Adobe Flash Platform." http://www.adobe.com/-products/httpdynamicstreaming/pdfs/httpdynamicstreaming_wp_ue.pdf, May 2013.

[21] R. Pantos (ed), "HTTP Live Streaming." http://tools.ietf.org/html/draft-pantos-http-live-streaming-10, October 2012.

[22] A. Zambelli, "Smooth Streaming Technical Overview." http://learn.iis.net/page.aspx/-626/smooth-streaming-technical-overview/, March 2009.

[23] T. Stockhammer, "Dynamic Adaptive Streaming Over HTTP - Standards and Design Principles," in *Proc. of ACM MMSys*, pp. 133–144, 2011.

[24] M. E. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," in *Proc. of ACM SIGMETRICS*, pp. 160–169, 1996.

[25] "HSDPA." http://en.wikipedia.org/wiki/High-Speed_Downlink_Packet_Access, May 2013.

[26] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a Discipline," *ACM CACM vol. 32*, pp. 9–23, 1989.

[27] "VG Nett Live." http://vglive.no, May 2013.

[28] K. Nichols, S. Blake, F. Baker, and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers." RFC 2474 (Proposed Standard), Dec. 1998. Updated by RFCs 3168, 3260.

[29] F. Baker, C. Iturralde, F. L. Faucheur, and B. Davie, "Aggregation of RSVP for IPv4 and IPv6 Reservations." RFC 3175 (Proposed Standard), Sept. 2001. Updated by RFC 5350.

[30] "Microsoft Media Server (MMS) Protocol." http://msdn.microsoft.com/en-us/library/-cc234711, May 2013.

[31] "Real-Time Messaging Protocol (RTMP) specification." http://www.adobe.com/-devnet/rtmp.html, May 2013.

[32] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications." RFC 3550 (Standard), July 2003. Updated by RFCs 5506, 5761, 6051, 6222.

[33] H. Schulzrinne, A. Rao, and R. Lanphier, "Real Time Streaming Protocol (RTSP)." RFC 2326 (Proposed Standard), Apr. 1998.

[34] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)." RFC 4340 (Proposed Standard), Mar. 2006. Updated by RFCs 5595, 5596, 6335.

[35] R. Stewart, "Stream Control Transmission Protocol." RFC 4960 (Proposed Standard), Sept. 2007. Updated by RFCs 6096, 6335.

[36] E. Birkedal, C. Griwodz, and P. Halvorsen, "Implementation and Evaluation of Late Data Choice for TCP in Linux," in *Proc. of IEEE ISM*, pp. 221 – 228, 2007.

[37] B. Wang, J. Kurose, P. Shenoy, and D. Towsley, "Multimedia Streaming via TCP: An Analytic Performance Study," in *Proc. of ACM MM*, pp. 16:1–16:22, 2004.

[38] V. K. Goyal, "Multiple Description Coding: Compression Meets the Network," *IEEE Signal Processing Magazine vol. 18*, pp. 74–93, 2001.

[39] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the Scalable Video Coding Extension of the H.264/AVC Standard," in *IEEE Trans. on Circuits and Systems for Video Technology*, pp. 1103–1120, 2007.

[40] C. Krasic and J. Walpole, "Quality-Adaptive Media Streaming by Priority Drop," in *Proc. of ACM NOSSDAV*, pp. 112–121, 2002.

[41] F. H. P. Fitzek, B. Can, R. Prasad, and M. Katz, "Overhead and Quality Measurements for Multiple Description Coding for Video Services," in *IEEE WPMC*, pp. 524–528, 2004.

[42] "H.264: Advanced video coding for generic audiovisual services," *ITU-T recommendation*, April 2013.

[43] S. Carmel, T. Daboosh, E. Reifman, N. Shani, Z. Eliraz, D. Ginsberg, and E. Ayal, "Network media streaming," in *US Patent Number 6,389,473*, May 2002. Filed: Mar. 1999. Assignee: Geo Interactive Media Group Ltd.

[44] "Coding of audio-visual objects: ISO base media file format, ISO/IEC 14496-12:2005," *ITU ISO/IEC*, 2010.

[45] D. Nelson, "Internet Information Services." http://www.iis.net/learn/media/live-smooth-streaming, May 2013.

[46] Nullsoft Inc., "The M3U Playlist format, originally invented for the Winamp media player." http://wikipedia.org/wiki/M3U, May 2013.

[47] H. Riiser, P. Halvorsen, C. Griwodz, and D. Johansen, "Low Overhead Container Format for Adaptive Streaming," in *ACM MMsys*, pp. 193–198, 2010.

[48] S. Sægrov, A. Eichhorn, J. Emerslund, H. K. Stensland, C. Griwodz, D. Johansen, and P. Halvorsen, "BAGADUS: An Integrated System for Soccer Analysis (Demo)," in *Proc. of ACM/IEEE ICDSC*, 2012.

[49] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, and D. Johansen, "BAGADUS: An Integrated System for Arena Sports Analytics - A Soccer Case Study," in *Proc. of ACM MMSys*, 2013.

[50] "libx264." http://www.videolan.org/developers/x264.html, May 2013.

[51] "MPlayer - The movie player." http://www.mplayerhq.hu, May 2013.

[52] "FFmpeg." http://www.ffmpeg.org, May 2013.

[53] "VLC." http://www.videolan.org/vlc/, May 2013.

[54] P. Halvorsen, D. Johansen, B. Olstad, T. Kupka, and S. Tennøe, "vESP: A Video-Enabled Enterprise Search Platform," in *Proc. of ICDKE*, pp. 534–541, 2010.

[55] P. Halvorsen, D. Johansen, B. Olstad, T. Kupka, and S. Tennøe, "vESP: Enriching Enterprise Document Search Results with Aligned Video Summarization," in *Proc. of ACM MM*, pp. 1603–1604, 2010.

[56] D. Johansen, P. Halvorsen, H. Johansen, H. Riiser, C. Gurrin, B. Olstad, C. Griwodz, Å. Kvalnes, J. Hurley, and T. Kupka, "Search-Based Composition, Streaming and Playback of Video Archive Content," *Multimedia Tools and Applications*, 2011.

[57] TalkMiner, "FXPAL TalkMiner." http://talkminer.com, May 2013.

[58] D. Johansen, H. Johansen, T. Aarflot, J. Hurley, Å. Kvalnes, C. Gurrin, S. Sav, B. Olstad, E. Aaberg, T. Endestad, H. Riiser, C. Griwodz, and P. Halvorsen, "DAVVI: A Prototype for the Next Generation Multimedia Entertainment Platform," in *Proc. of ACM MM*, pp. 989–990, 2009.

[59] "Tippeligaen." http://en.wikipedia.org/wiki/Tippeligaen, May 2013.

[60] "Internet Information Services Smooth Streaming Extension." http://www.iis.net/-downloads/microsoft/smooth-streaming, May 2013.

[61] J. Postel, "Internet Protocol." RFC 791 (Standard), Sept. 1981. Updated by RFCs 1349, 2474.

[62] C. Müller, S. Lederer, and C. Timmerer, "An Evaluation of Dynamic Adaptive Streaming over HTTP in Vehicular Environments," in *Proc. of ACM MoVid*, pp. 37–42, 2012.

[63] H. Riiser, H. S. Bergsaker, P. Vigmostad, P. Halvorsen, and C. Griwodz, "A Comparison of Quality Scheduling in Commercial Adaptive HTTP Streaming Solutions on a 3G Network," in *Proc. of ACM MoVid*, pp. 25–30, 2012.

[64] R. Houdaille and S. Gouache, "Shaping HTTP Adaptive Streams for a Better User Experience," in *Proc. of ACM MMSys*, pp. 1–9, 2012.

[65] S. Akhshabi, S. Narayanaswamy, A. C. Begen, and C. Dovrolis, "An Experimental Evaluation of Rate-Adaptive Video Players over HTTP," *Image Communication*, vol. 27, pp. 271–287, 2012.

[66] L. De Cicco and S. Mascolo, "An experimental investigation of the Akamai adaptive video streaming," in *Proc. of USAB*, pp. 447–464, 2010.

[67] L. De Cicco, S. Mascolo, and V. Palmisano, "Feedback Control for Adaptive Live Video Streaming," in *Proc. of ACM MMSys*, pp. 145–156, 2011.

[68] "Squid." www.squid-cache.org/, 2013.

[69] K. R. Evensen, T. Kupka, D. Kaspar, P. Halvorsen, and C. Griwodz, "Quality-Adaptive Scheduling for Live Streaming over Multiple Access Networks," in *Proc. of ACM NOSSDAV*, pp. 21–26, 2010.

[70] P. Rodriguez and E. W. Biersack, "Dynamic Parallel Access to Replicated Content in the Internet," *IEEE/ACM Trans. on Networking*, pp. 455–465, 2002.

[71] G. G. Feng Wu and Y. Liu, "Glitch-free media streaming," *Patent Application (US2008/0022005)*, January 24, 2008.

[72] D. Kaspar, K. Evensen, P. Engelstad, and A. F. Hansen, "Using HTTP Pipelining to Improve Progressive Download over Multiple Heterogeneous Interfaces," in *Proc. of IEEE ICC*, pp. 1–5, 2010.

[73] M. Devera, "Princip a uziti HTB QoS discipliny," in *SLT*, pp. 149–158, 2002.

[74] Move Networks, "Internet television: Challenges and opportunities," tech. rep., November 2008.

[75] V. G. Cerf and R. E. Kahn, "A Protocol for Packet Network Intercommunication," *ACM SIGCOMM CCR*, pp. 71–82, 2005.

[76] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. B. Postel, L. G. Roberts, and S. S. Wolff, "A Brief History of the Internet," *ACM SIGCOMM CCR*, pp. 22–31, 2009.

[77] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, "TCP Fast Open," in *Proc. of ACM CoNEXT*, 2011.

[78] V. Jacobson and M. J. Karels, "Congestion Avoidance and Control," 1988.

[79] J. Nagle, "Congestion Control in IP/TCP Internetworks." RFC 896, Jan. 1984.

[80] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm." RFC 3782 (Proposed Standard), Apr. 2004. Obsoleted by RFC 6582.

[81] S. H. Low, L. L. Peterson, and L. Wang, "Understanding TCP Vegas: a Duality Model," *ACM SIGMETRICS*, pp. 226–235, 2001.

[82] S. Ha, I. Rhee, and L. Xu, "CUBIC: a New TCP-Friendly High-Speed TCP Variant," *ACM SIGOPS Oper. Syst. Rev.*, pp. 64–74, 2008.

[83] S. Floyd, "HighSpeed TCP for Large Congestion Windows." RFC 3649 (Experimental), Dec. 2003.

[84] C. Caini and R. Firrincieli, "TCP Hybla: a TCP Enhancement for Heterogeneous Networks," *International Journal of Satellite Communications and Networking*, 2004.

[85] L. Xu, K. Harfoush, and I. Rhee, "Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks," in *Proc. of IEEE INFOCOM*, pp. 2514–2524, 2004.

[86] S. Bradner, "The Internet Standards Process – Revision 3." RFC 2026 (Best Current Practice), Oct. 1996. Updated by RFCs 3667, 3668, 3932, 3979, 3978, 5378, 5657, 5742, 6410.

[87] M. Handley, J. Padhye, and S. Floyd, "TCP Congestion Window Validation." RFC 2861 (Experimental), June 2000.

[88] C. Casetti and M. Meo, "A New Approach to Model the Stationary Behavior of TCP Connections," in *Proc. of IEEE INFOCOM*, pp. 367–375, 2000.

[89] M. A. Marsan, C. Casetti, R. Gaeta, and M. Meo, "Performance Analysis of TCP Connections Sharing a Congested Internet Link," *Performance Evaluation vol. 42*, pp. 109–127, 2000.

[90] A. Wierman, T. Osogami, and J. Olsen, "A Unified Framework for Modeling TCP-Vegas, TCP-SACK, and TCP-Reno," in *Proc. of IEEE MASCOTS*, pp. 269–278, 2003.

[91] W. Elkilani, "An Analytical Model of Non-Persistent TCP Sessions Sharing a Congested Internet Link," in *Proc. of ICNM*, pp. 76 –82, 2009.

[92] H.-P. Schwefel, "Behavior of TCP-like Elastic Traffic at a Buffered Bottleneck Router," in *Proc. of IEEE INFOCOM*, pp. 1698–1705, 2001.

[93] I. Hofmann, N. Farber, and H. Fuchs, "A Study of Network Performance with Application to Adaptive HTTP Streaming," in *Proc. of IEEE BMSB*, pp. 1–6, 2011.

[94] J. Esteban, S. Benno, A. Beck, Y. Guo, V. Hilt, and I. Rimac, "Interaction Between HTTP Adaptive Streaming and TCP," in *Proc. of ACM NOSSDAV*, 2012.

[95] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing TCP's Retransmission Timer." RFC 6298 (Proposed Standard), June 2011.

[96] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen, "What Happens When HTTP Adaptive Streaming Players Compete for Bandwidth," in *Proc. of ACM NOSSDAV*, 2012.

[97] L. Kontothanassis, "Content Delivery Considerations for Different Types of Internet Video." http://www.mmsys.org/?q=node/64, May 2013.

[98] T. Kupka, P. Halvorsen, and C. Griwodz, "An Evaluation of Live Adaptive HTTP Segment Streaming Request Strategies," in *Proc. of IEEE LCN*, pp. 604–612, 2011.

[99] T. Kupka, P. Halvorsen, and C. Griwodz, "Performance of On-Off Traffic Stemming From Live Adaptive Segment HTTP Video Streaming," in *Proc. of IEEE LCN*, pp. 405–413, 2012.

[100] NS-2, "The Network Simulator ns-2." http://www.isi.edu/nsnam/ns/, May 2013.

[101] L. Plissonneau and E. Biersack, "A Longitudinal View of HTTP Video Streaming Performance," in *Proc. of ACM MMSys*, pp. 203–214, 2012.

[102] W3techs, "Usage statistics and market share of unix for websites." http://w3techs.com/technologies/details/os-unix/all/all, May 2013.

[103] H. Riiser, T. Endestad, P. Vigmostad, C. Griwodz, and P. Halvorsen, "Video Streaming Using a Location-based Bandwidth-Lookup Service for Bitrate Planning," *ACM TOMCCAP*, accepted 2011.

[104] L. A. Grieco and S. Mascolo, "Performance Evaluation and Comparison of Westwood+, New Reno, and Vegas TCP Congestion Control," *ACM SIGCOMM Comput. Commun. Rev. vol. 34*, pp. 25–38, 2004.

[105] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing," in *Proc. of IEEE INFOCOM*, pp. 1157–1165, 2000.

[106] T. Lohmar, T. Einarsson, P. Frojdh, F. Gabin, and M. Kampmann, "Dynamic Adaptive HTTP Streaming of Live Content," in *Proc. of IEEE WoWMoM*, pp. 1–8, 2011.

[107] P. Ni, A. Eichhorn, C. Griwodz, and P. Halvorsen, "Fine-Grained Scalable Streaming from Coarse-Grained Videos," in *Proc. of ACM NOSSDAV*, pp. 103–108, 2009.

[108] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," *Comput. Commun. Rev. vol. 34*, pp. 281–292, 2004.

[109] A. jan Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante, "Drafting Behind Akamai (Travelocity-Based Detouring)," in *Proc. of ACM SIGCOMM*, pp. 435–446, 2006.

[110] A. Hubert and R. van Mook, "Measures to prevent DNS spoofing." http://tools.ietf.org/html/draft-hubert-dns-anti-spoofing-00, 2007.

[111] "Dailymotion." http://www.dailymotion.com/, May 2013.

[112] "FAST Search Server 2010 for SharePoint." http://sharepoint.microsoft.com/en-us/-product/capabilities/search/Pages/Fast-Search.aspx, May 2013.

[113] M. Belshe and R. Peon, “SPDY Protocol.” http://tools.ietf.org/id/draft-mbelshe-httpbis-spdy-00.txt, May 2013.