UNIVERSITY OF OSLO
Department of informatics

Investigating the limitations of video stream scheduling in the Internet

Master thesis

Espen Jacobsen

May, 2009

Investigating the limitations of video stream scheduling in the Internet

Espen Jacobsen

May, 2010

Acknowledgments

I would like to thank my supervisors Carsten Griwodz and Pål Halvorsen, for their guidance, valuable feedback and helpfull attitude.

I would also like to thank the guys at the lab, for giving me a great and motivating work environment. Special thanks to Chris Carlmar for help with setting up the test network.

I would also like to thank my family, girlfriend and friends, for their moral support along the way.

Espen Jacobsen May, 2010

Contents

1 Introduction **2**

1.1 Background 2

1.2 Problem statement 4

1.3 Main contributions 5

1.4 Outline 6

2 Streaming- and communication- techniques **8**

2.1 Terminology 9

2.2 Protocols and standards 10

2.3 Multicast 11

2.4 Periodic Broadcast 14

2.4.1 Pyramid Broadcasting 15

2.4.2 Skyscraper Broadcasting 16

2.4.3 Pagoda Broadcasting 17

2.4.4 Harmonic Broadcast 17

2.4.4.1 Cautious Harmonic Broadcast 19

2.4.4.2 Quasi-Harmonic Broadcast 19

2.4.4.3 Polyharmonic Broadcasting protocol 20
2.5 Patching algorithms 20
2.5.1 Greedy patching and Grace patching 22
2.5.2 Periodic Buffer Reuse 22
2.5.3 Greedy Buffer Reuse 23
2.5.4 Lambda Patching 24
2.6 TCP-based streaming 24
2.6.1 Multimedia Streaming via TCP: An analytic Performance Study . 25
2.6.2 Quality-Adaptive Media Streaming by Priority Drop 26
2.6.3 DAVVI 29
2.6.3.1 Segmentation and dissemination 30
2.6.3.2 Metadata and annotation 31
2.6.3.3 Search and recommendation 31
2.7 Summary and discussion 31

3 Design **34**
3.1 System overview 35
3.2 Video stream format 37
3.3 Multicast server design 37
3.3.1 Channel creation 37
3.3.2 Channel management 38
3.3.3 Transmission of segments 39
3.3.4 Segment delivery time 40

3.4 Patching server design 41

3.5 Client design 42

3.5.1 Client overview 42

3.5.2 Client wait time 43

3.5.3 Quality adaptation 44

3.5.4 Scheduling decisions 45

3.5.5 Client buffer management 48

3.5.6 Receiving multicast video stream 49

3.5.7 Downloading patch video stream 50

3.5.8 Thread synchronization 51

3.5.9 Download module and player module communication 52

3.6 Summary 53

4 Implementation 54

4.1 Multicast server implementation 55

4.1.1 The Live555 framework 55

4.1.1.1 Live555 typical program flow 55

4.1.1.2 Live555 library description 56

4.1.2 Server components 57

4.1.3 Establishing client sessions 58

4.1.4 Channel creation 59

4.1.5 Scheduling and transmitting data 60

4.2 Client side implementation 62

4.2.1 Client functions 62
4.2.2 Client multicast buffer 63
4.2.3 Receiving multicast video stream 65
4.2.4 Get patch stream 66
4.2.5 Read from multicast buffer 68
4.3 Implementation challenges using curl 69
4.4 Summary 69

5 Experiments **72**
5.1 Test environment 73
5.1.1 Testbed 73
5.1.2 Media description 74
5.1.3 Tools and techniques 74
5.1.4 Test parameters 74
5.2 Experiments 76
5.2.1 Proof of concept 76
5.2.2 Bandwidth 80
5.2.3 Packet loss 90
5.2.4 Jitter and reordering 95
5.2.5 Delay 99
5.3 Summary 104

6 Conclusion **106**
6.1 Summary and contribution 106
6.2 Future work 108

Bibliography **110**

List of Figures

2.1 Unicast vs. multicast. By using unicast, the server has to send data to each client individually. With multicast the server has to send data out only once, regardless of how many receiving clients. 11

2.2 Pyramid broadcast with two videos. Client receives video a. *Adapted from [13]* 15

2.3 Skyscraper broadcast. *Adapted from [13]* 16

2.4 Pagoda broadcast. *Adapted from [13]* 17

2.5 Harmonic broadcast. *Adapted from [13]* 18

2.6 Cautious Harmonic broadcast. *Adapted from [13]* 19

2.7 Quasi-Harmonic broadcast. *Adapted from [15]* 20

2.8 The concept of patching 21

2.9 Receiving a patch stream. *Adapted from [13]* 21

2.10 Torrent-like HTTP streaming. 29

2.11 Quality adaptation in torrent-like HTTP streaming. 30

3.1 An overview of our system 35

3.2 Transmission of segments in different qualities on multicast channels. . . 40

3.3 HTTP GET RANGE request. 41

3.4 Client overview 42

3.5 Check if multicast segments are stored in the buffer. The check is done 1 second after the expected arrival time of a segment. 46

3.6 Scheduling of one segment. Client buffers three seconds and the patch is given one second to download. 47

3.7 Scheduling of patch request 48

3.8 Segment flow between buffers 48

5.1 Overview of testbed 73

5.2 Client receiving a patch stream and a multicast stream 76

5.3 Multicast server sending rate 77

5.4 Client receiving segments 78

5.5 Section of client receiving segments 79

5.6 Playback quality 79

5.7 Client receiving segments with bandwidth 2Mbps 82

5.8 Results with bandwidth 8Mbps 84

5.9 Results with bandwidth 6Mbps 84

5.10 Results with bandwidth 4Mbps 85

5.11 Results with bandwidth 2Mbps 85

5.12 Results with bandwidth 1Mbps 86

5.13 The % of received segments in different qualities 88

5.14 Results with packet loss 0.2% 91

5.15 Results with packet loss 0.5% 92

5.16 Results with packet loss 1% 92

5.17 Results with packet loss 3% 93

5.18 Results with packet loss 5% 93
5.19 Results with 185 milliseconds delay and 50% jitter 97
5.20 Results with 0 milliseconds added delay 100
5.21 Results with 50 milliseconds added delay 101
5.22 Results with 185 milliseconds added delay 102

List of Tables

4.1 Application header 65

5.1 Proof of concept: Test parameters 77

5.2 Bandwidth: Test parameters 81

5.3 Abbreviations for figure 5.7 82

5.4 Packet loss: Test parameters 90

5.5 Jitter: Test parameters 96

5.6 Delay: Test parameters 99

Chapter 1

Introduction

1.1 Background

Multimedia is content consisting of combinations of different media like text, audio and video, and often the multimedia services include user interactivity. Television broadcasters have been the main providers of media content consisting of audio and video. Consumers have been watching television broadcasts via their TVs for decades, and the VCR- and DVD- players have been the most important alternative for people to watch movies at home. During the recent years, powerful personal computers and high speed Internet connection have found their way into private households. This has opened for the possibility of providing multimedia content over the Internet, making the Internet a serious competitor to the good old TV and video player.

As the popularity of video streaming services is growing, people from all over the world are able to distribute their own personal media content on online web services like www.youtube.com and www.metacafe.com. The P2P technology has also been very important in the distribution of media content over the Internet. In a P2P system all clients act as servers, sharing their content with other clients. Clients and servers have equal roles. Popular applications like Sopcast and TVU Player use the P2P technology and have given private users the possibility to create their own channel, sharing their content with the whole world.

The growing trend of multimedia streaming on the Internet has also reached the television companies and the online news providers. To be able to maintain customer satisfaction and to stay competitive in the market, these companies have invested a lot in systems that can provide them with online multimedia services. Online news providers can today,

instead of just publishing articles, also provide the users with video reports. Norwegian newspapers like VG, Dagbladet and Aftenposten all have their own video streaming service. In addition to having video reports of news events, VG have their own channel called VG-TV where they broadcast live football. Television broadcast companies like NRK and TV2 provide the user with live TV broadcasts over the Internet, and they can publish their own productions for the user to watch at a later time. This gives the user the advantage of watching a video, that used to only be broadcasted on TV, whenever he or she wants, a concept known as video on demand. Another use of the video on demand concept is for online movie rental. This is meant to be a competitor to the DVD rental, which is still most popular today. The Online service, www.filmarkivet.no is an online movie rental service where the customer can rent a movie for 24 hours. The movie is streamed from their servers so the client does not have to download the whole video before watching.

All of these online services introduce challenges and issues for the network, the providers servers and the equipment of the end user. A system that is to deliver media content to a large number of users has to scale well. To be able to satisfy user requirements like timely delivery of data and an instant response to the client requests, the servers must have a lot of processing power to cope with the number of users. The network must also have enough resources to transmit the data to all the clients within acceptable time. This concerns issues like bandwidth requirement, delay and routing capabilities in the network. On client side, we face challenges regarding buffering and the performance power of the users equipment.

The most popular approach for delivering multimedia content to clients over the Internet today is by using unicast. Unicast-based systems do not scale well, as the amount of data that must be transmitted from multimedia streaming servers grows linearly with the number of connected clients. As the popularity of these multimedia streaming services grows, the providers are noticing that it is difficult to provide enough resources to serve the number of concurrent users. To cope with this scaling issue, we present multicast along with some stream scheduling schemes that are designed for better resource utilization. However, Internet does have some limitations that gives these schemes some challenges and issues that we have to deal with. The Internet is a heterogeneous environment and different users have different network resources available. This is a big issue, because users with 1Mbps available bandwidth are perhaps not be able to receive the same video as a user with 10Mbps available bandwidth. Providers should give their servers functionality to provide users with video they are able to receive. Another challenge is packet loss in the network. When users streaming a video and experience packet loss, the quality of the video drops as parts of images or sound can be lost. Multicast does not cope with packet loss any better than unicast over UDP does. Multicast based systems must therefore rely on other techniques, presented in this thesis, to neutralize the effect of packet loss in the Internet.

1.2 Problem statement

Streaming media files over the Internet have become highly popular in recent years, and the requirements for network resources, and client and server hardware have grown. Unicast is the popular scheme of data transmission in the Internet today. Compared to a multicast scheme, a unicast scheme has great limitations when it comes to handling large amounts of data and scalability. Multicast is used to reduce the use of network resources between the server and its clients, by sending data over each link in the network only once, regardless of how many clients. Copies of the data are created as the links to the different clients split.

There are many proposed and implemented streaming scheduling techniques, for use in a multicast environment, to reduce load on server, client and network resources. Periodic broadcast and patching are two stream scheduling techniques that use multicast.

A periodic broadcast server divides the video file into many segments and transmits these segments over multicast addresses. By using that clients play back the video file sequentially, reception of late parts of the video can be completed later than that of earlier parts. This technique lets clients arrive at different times and still lets them listen to the same multicast channels.

Patching[18] is used in a multicast environment to reduce server and network resource usage and achieve close to zero playback start-up latency, by giving the clients a unicast patch stream to satisfy the fact that individual clients join a multicast stream at different times. Clients that did not receive the first part of the video from the multicast stream, receive a unicast patch stream. This patch stream is for instant consumption while the data from the multicast stream is stored in the clients buffer for later playback.

In this thesis, we have a look at an implementation of patching in a multicast environment. We have chosen the patching scheme as it, compared to periodic broadcasting schemes, limits client buffering, it does not occupy as many multicast IP addresses, and it gives the client a shorter startup time. For our unicast patch stream, we use a torrent-like HTTP approach. It uses TCP and is therefore reliable when loss occurs in the network, and it is easy to use with the concept of quality adaptation. Quality adaptation of video is used to provide users with different video quality with respect to their network resources. In general, users with lower available bandwidth receive lower quality video than users with high available bandwidth. The concept of using a torrent-like HTTP approach for patching and quality adaptation in a multicast scheme has not been implemented or investigated in any other article that we are familiar with. However, the concepts of torrent-like HTTP streaming and quality adaptation have been used in unicast based systems with great success.

There are many papers that describe the theoretical performance of multicast stream scheduling techniques, so we know that quite well, but we are not familiar with articles investigating how stream scheduling techniques handle challenges like packet loss, jitter, reordering and bandwidth heterogeneity amongst clients. In this thesis, we perform experiments on a patching scheme to investigate how the different challenges in the Internet affect the network resources and the reception of video streams. We investigate how to cope with challenges like packet-loss, jitter and reordering in a multicast stream when sending in a real network, and how to cope with network resource heterogeneity like bandwidth and delay amongst clients by using quality adaptation.

We perform experiments on an implementation of a patching scheme that is combined and integrated with a modern HTTP-streaming solution delivering the patch streams, both for streaming the prefix and for repairing the multicast stream. The scheme must also support quality adaptation. We use the Live 555 library [24] for implementation of our multicast server and implement our client based on the DAVVI media player [20]. Originally, DAVVI is used for streaming video over a torrent-like HTTP unicast stream, but we implement functionality for multicast and patching for live and popular video transmissions.

1.3 Main contributions

In this thesis, we argue that today's unicast delivery scheme for video does not scale well when the number of clients is growing. The need for stream scheduling schemes is increasing as more media data is made available on the Internet and the number of clients using online media services is rising. Many have argued the need for multicast to cope with the problem om scalability, but the Internet Service Providers (ISP) have not switched it on. However, the ISPs are looking at solutions to the challenges that the growing amount of network traffic provides, and we believe that the multicast scheme is a solution that is to be taken into use in the future. In this thesis, we have investigated several stream scheduling techniques of periodic broadcast and patching. We have chosen to further examine the patching scheme as it, compared to periodic broadcasting schemes, limits client buffering, does not occupy as many multicast IP addresses, and gives the client a shorter startup time. The patching scheme can also be extended to cope with packet loss in the network as it can retransmit whatever data that is lost from the multicast stream.

The main contributions in this thesis is to investigate a patching scheme in a multicast environment for stream scheduling in the Internet. We have implemented a server that provide the clients with video over multicast channels. The video is available in several qualities so the clients can receive the best video quality based on the available resources

on their network connection. The server is implemented using the Live555 framework [24]. We have also implemented a client that is responsible for receiving and consuming the video from the multicast stream. It is also responsible for downloading patch streams for the parts of the video that are missing from the multicast stream, either because it joins an ongoing multicast stream or due to packet losses in the network. The client is implemented by adding functionality to the download module of the DAVVI media player[20], for reception of a multicast stream. The original implementation in the download module for downloading patched data over HTTP is used with some modifications.

The Internet is not a scene without any challenges or problems. Challenges like packet loss, jitter and heterogeneous Internet connections amongst the clients are present. We have investigated how these challenges affect our implementation in a real network environment. Our implementation has been tested while we imposed different challenges that are present in the Internet onto our test network.

Based on the results from our experiments, we have concluded that our implementation of the patching scheme is able to cope with the challenges that we know exist in the Internet. By the use of a quality adaptation scheme, the user is able to receive video in the best quality possible with regard to these challenges.

1.4 Outline

In chapter 2, we give a description of some terminology and concepts that are relevant to this thesis. We then examine the multicast technology as well as different periodic broadcast schemes and patching schemes. We also examine some TCP-based streaming systems, with regards to the TCP unicast stream we use for patching and the quality adaptation scheme. We give a presentation of the design of our client and server in chapter 3, and describe how our design has been realized in our implementation in chapter 4. In chapter 5, we describe our experiments and evaluate and discuss the results. We then finish our thesis by concluding and summarize our work, as well as proposing some future work in chapter 6.

Chapter 2

Streaming- and communication-techniques

Transferring multimedia content over the Internet demands a lot of network resources as well as server and client resources. The popular choice of many online video service providers is to treat every client individually using unicast without sharing of data among the clients, which means that the resource requirements of the network grow linearly with the number of clients. The techniques presented in this chapter are developed with reduction of resource usage in mind. Multicast is a communication technique developed to release server and network load with regard to network traffic. The techniques of Periodic Broadcast [4, 13, 7] and Patching [18, 7] take advantage of multicast to reduce server load. We start this chapter with presenting some terminology used in this thesis before we give an introduction to different techniques used for network communication and streaming of video.

2.1 Terminology

In this section we present relevant terminology that is useful when reading this thesis.

- **Unicast** is a communication technique, where the data is transferred from one sender to only one receiver. This is the most common communication technique today. Unicast transmission of data to many destinations at virtually the same time consumes a lot of resources since all of the different connections consume computing resources on the server and require an own separate network bandwidth for transmission.
- **Multicast** is a communication technique, where the data is transferred from one sender to many receivers. The receivers announce their interest in the data and join a multicast group. The server can send the data out on one link to all the receivers in the group without having the knowledge of how many receivers there are. The routers involved in delivering a multicast are organized in a tree structure, and whenever a packet from the server reaches a branch it is up to the routers to forward the packet in the direction of the receivers. Unlike unicast, multicast scales well to a larger number of receivers.
- **Broadcast** is a communication technique, where the data is transferred from the host to every device in the broadcast domain in the network at the same time. The broadcasting technique can potentially consume a lot of unnecessary resources as the host delivers packets to receivers that might not be interested in receiving data.
- **Stream** is a coherent flow of data transferred over a network path from a server to a client.
- **Channel** is an addressable resource used to transfer data between the server and client.
- **Client wait time** is the time the client has to wait after requesting the video before the video playback begins.
- **Consumption rate** is the rate at which a client consumes the video.
- **Quality adapted playback [20]:** A server offers the video in different qualities and let the client decide which quality to receive. This decision is being made with regard to network performance and bandwidth constraints. This concept is developed to take care of the problem with heterogeneous clients.
- **Trick play** are functions that let the client navigate within the video, like play, stop, pause, forward and rewind.

- **Video on demand** is a concept that allows the client to watch video content at any time specified by the client.
- **Peer-2-peer** is concept where all participants play equal roles. Clients act as severs and share data amongst themselves.
- **Maximum transmission unit (MTU)** is the maximum size of a packet that a network protocol can transmit.
- **Jitter** is the variation in the end-to-end delay between sender and receiver.
- **Reordering** is when packets arrive in a different order than they were sent from the server.

2.2 Protocols and standards

In this section we present some protocols and standards that we use in our work. The list presented here is only meant as a short description and each element is further explained in the context of its use in the chapters below.

- **Real-time Transport protocol (RTP) [17]:** RTP is a transport protocol suitable for real-time applications. "RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services" **[17]**. This protocol include timestamps which can be used when synchronizing the sending of packets.
- **Hypertext Transfer Protocol (HTTP) [35, 31]:** HTTP is an application layer protocol. We use this protocol's GET function to download the video segments from our video servers.
- **Real Time Stream Protocol (RTSP) [16]:** RTSP is application level protocol for control over media stream servers. It supports VCR like commands like pause, rewind and forward.
- **Transmission Control Protocol (TCP)[19] :** TCP is a transport layer protocol. It is a reliable connection oriented host-to-host protocol for delivery of data. This protocols offers guaranteed delivery with use of packet loss detection and error detection. We use this protocol in our implementation to download a patch stream from our patch servers.

- **User Datagram Protocol (UDP)[35, 30]:** UDP is a transport layer protocol. It is an unreliable connection-less protocol. It is widely used in applications where time dependent delivery is more important than accurate delivery, such as transmitting speech or video. We use this protocol for transmitting video content from our multicast server.

- **Internet Group Management Protocol (IGMP) [10]:** We use this protocol for managing the membership of the Internet protocol multicast group.

- **H.264 [33]:** We use H.264, which is a standard for video compression.

2.3 Multicast

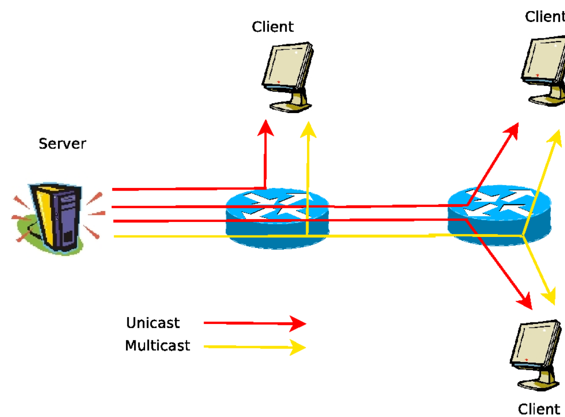


Figure 2.1: Unicast vs. multicast. By using unicast, the server has to send data to each client individually. With multicast the server has to send data out only once, regardless of how many receiving clients.

When several users want to receive the same video from an Internet service at the same time, multicast [4, 2] technology is a great technology to use. If all the users were to receive an individual unicast stream from the server, it could cause huge performance issues for both the server and the network. When a server uses multicast technology, a user is put into a group with other users depending on which video the user wants to stream. A spanning tree between the server and the receivers is constructed so that the multicast packets are received by all the users in a group. All members of a group receive the same video. The server sends out a single stream to each group. Whenever there is a branch in the tree, identical copies of a packet are made and sent to each branch, as seen in figure 2.1, in contrast to unicast, where the number of packets sent from the server is

equal to the number of receivers. By using multicast, many of the performance issues caused by unicast is dealt with. The server load is reduced because the server no longer has to manage each user individually, but only has to manage the different groups. This of course makes the system more scalable as it can handle many users without upgrading the system. The network bandwidth consumption is also reduced as the data sent out from the server is reduced to only one multicast stream per group. For transmission of data, multicast uses the UDP protocol. UDP is currently the only transport layer protocol standardized by the Internet Engineering Task Force (IETF) that supports multicast.

The disadvantage with multicast is that it is limited by the UDP protocol, which means that it is best effort and does not have any functionality for loss recovery and avoiding that packets are delivered out of sequence. If few packets are lost in a video stream, it is hardly noticed by the user, but if many packets fail to be delivered to a user, the viewing experience might not be acceptable. If a packet is lost on its way to a user, the multicast server does not simply resend that packet on the multicast stream. This would cause that every member of the multicast stream would receive the retransmitted packet, even though some members would not need it, causing unnecessary network traffic and extra server load. Packets can be delivered out of sequence when there is a change in the multicast tree, causing packets to change paths. Packets sent before the update of the tree could potentially have a longer way than packets sent after the update. This means that packets sent after the update can be received before packets sent before the update. It is up to the application to cope with packet loss, either by retransmission via unicast or trying to repair the missing data locally. It is also the application's responsibility to store any packets, received out of sequence, in the correct order.

Another issue with multicast streaming is that it is not guaranteed that all users in a group are homogeneous. They may for instance have different Internet access speeds, giving some users problems watching the video when others might not have any problems. The article "Tutorial on Multicast Video Streaming Techniques" [4] suggests some solutions to this problems. One solution is to have a single stream with source rate adaptation. With this approach, the multicast server transmits one stream and adjusts the bitrate of the stream based on feedback from clients. This approach has many drawbacks. One drawback is that it needs a scalable feedback mechanism, as the client causes overhead traffic related to this feedback. Another drawback is that the adjusted bit-rate should satisfy the client with the least resources, which penalizes all other clients. The article also proposes a solution called layered multicast streaming. With this approach, the video is encoded into a set of incremented layers. One layer, being the base layer, contains the essential video information with a basic low quality and a set of enhancement layers, that are used to improve the quality of the received video. The different layers are transmitted on different multicast channels. The client first joins the multicast channel, carrying the basic layer, to receive the basic video quality and then join other channels, sending

enhancement layers, to improve quality if enough bandwidth are available. The drawback of this approach is that it is difficult to synchronize the different multicast channels, as these must be fully synchronized for this approach to work. This solution of layered multicast streaming is better known as Receiver-driven Layered Multicast (RLM) and is presented in the article Receiver-driven Layered Multicast [25]. Single stream with active network adaptation is another solution the article [4] suggests. With this approach, the process of quality adaptation is relegated to the intermediate routers. Some routers implement agents that manipulates the video stream to adapt its bit-rate or to split the original stream into multiple streams with different bit-rates or formats using transcoding techniques. The challenges for this solution is to decide which routers in the network that should implement the agents, and the resource requirements of routers when they must forward many streams. Another large problem is that routers in a network has different owners, which makes it very difficult to implement agents on the different routers. Multiple streams with receiver switching is the last solution the article [4] presents. With this approach, the server provides a set of pre-encoded streams of the original video. These streams are coded at different rates targeted for common network access speeds. Each stream is sent out on a separate multicast channel and it is up to the client to decide which channel it wants to join. This solution requires no feedback from the client to the server. This approach for coping with heterogeneity among clients is similar to the approach we have chosen in this thesis. The effect of the issues of bandwidth heterogeneity amongst clients and packet loss is further investigated in this thesis.

When a multicast packet is sent from the server and out on the network, the multicast protocol makes duplicates of the data packet, when branches are encountered, as the packet moves through the spanning tree. There are several approaches to how this spanning tree can be generated. In general, we have three different types of multicast architectures [23]: Application Layer Multicast (ALM), Overlay Multicast (OM) and IP multicast. ALM is multicast on level 7 of the OSI model [35]. With this approach, the participating end host controls group memberships, multicast tree construction and data forwarding. It is very easy to deploy because routers in the network do not have to support multicast. However, ALM is based on the underlying network and by that, it is based upon an unicast environment meaning that it is not the best solution for solving the scaling problem in the network. ALM generates a lot of overhead control traffic for maintaining group memberships and multicast trees among the end users, and for monitoring the network by sending expensive probing messages. The amount of control traffic and is increased as the number of clients within a group grows. In addition to that, the lack of knowledge of network topology could give longer end-to-end delays and lower efficiency than IP multicast. The end-to-end delay is increased as more clients joins the group. This is because it is difficult to maintain an effective tree structure when the knowledge of the network topology is limited. Even though ALM is more effective than normal unicast, these factors make it inappropriate for us to work with in this thesis. OM is similar to

ALM operating at the application layer, where multicast can be supported by the use of additional deployed intermediate nodes in the network. These nodes are called proxies and can form a backbone overlay network. The proxies cooperate to construct a multicast tree among themselves in the network for data delivery. Proxies use unicast to transmit packets to end nodes located outside the backbone overlay. Routers in the network would not have to support multicast since it is the responsibility of the proxies to handle group management, distributing the packets through the multicast tree and to the end nodes. The drawback of this approach is that the proxies must be very carefully placed in the network to be able to construct an efficient multicast tree structure and to achieve high performance. IP multicast is router based, meaning that all routers in the network must support multicast. The multicast routers manage group memberships and construct multicast trees for delivering of data. IP multicast is able to scale to a large group of clients and truly improves network efficiency since data packets are only replicated when it comes to a branch in the tree. Compared to ALM and OM, IP multicast provides the most efficient multicast tree because all the routers are involved in the process, but at the cost of high storage requirements for routers, with regard to the state scalability issue in presence of a large number of multicast groups. The architecture of IPv4 multicast is used in this thesis and only mentioned as multicast.

Multicast is great for reducing performance issues in the context of streaming, but it also introduces some challenges [4, 2]. When users want to stream the same video it is a good idea to put them in groups and use the multicast protocol for video streaming, but what if several user do not request the video at the same time? One solution is to make video available at given times and to put all the users that request the video before the video starting time into a group. This technique is called batching. Of course, delayed start-up time does not give a very good Video on Demand service as the user may have to wait for a while before watching the video, depending on how long time it is to the video starting time. There are some techniques that can be used together with multicast that solves the problem of users having asynchronous requests. These techniques are introduced in sections below.

2.4 Periodic Broadcast

In this section, we explain the concept of periodic broadcast [7]. Periodic broadcast is a technique that could help to minimize the delayed start-up time issue with multicast streaming and support asynchronous client arrival time. The concept of periodic broadcast is to divide a video into segments. The users consume the video segments sequentially, which means that later parts of the video can be received at a later time than the first part of the video. The segments are placed in different channels where the segments are repeated

periodically. The earlier parts of the video are multicast more frequently to reduce the playback start-up latency. The users are listening to several multicast addresses at the same time so they can store future segments in a local buffer for later playback. One of the drawbacks with periodic broadcast is that it leaves no room for user interactivity, other than pause, as the user has to receive the whole video from start to end. Below, we present some periodic broadcast algorithms.

2.4.1 Pyramid Broadcasting

Pyramid Broadcasting [13, 8] divides the video into segments. The length of each segment grows exponentially the further out in the video the segment belong. The pyramid broadcasting protocol divides the available bandwidth into several channels with equal bandwidth.

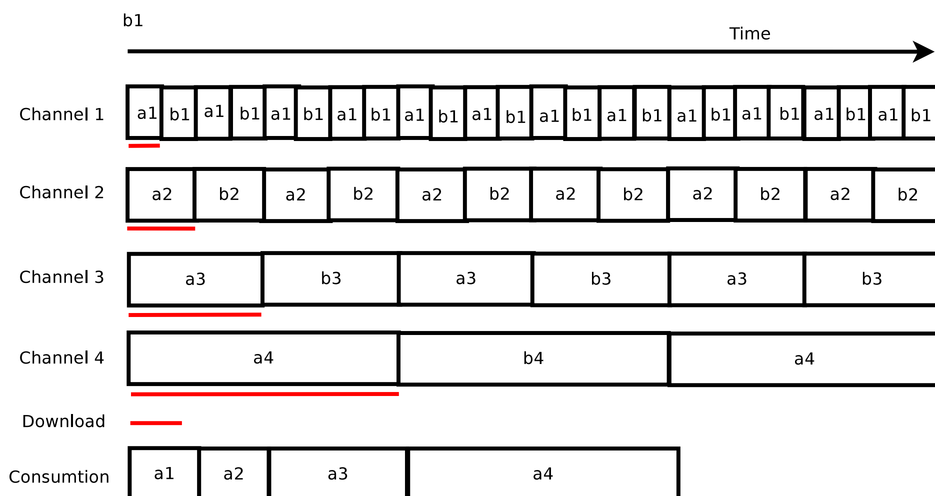


Figure 2.2: Pyramid broadcast with two videos. Client receives video a. *Adapted from [13]*

Each channel can repeat segments from different videos but only one segment per video is repeated, as seen in figure 2.2. When a user wants to stream a video, he first receives a segment from the first channel and starts playing the segment at the same time. For all the remaining segments, the user can begin to receive a new segment as soon as the playback of the previous segment has began. To be able to watch the video without any interruptions, the user must be able to start receiving a segment before the previous segment has finished playing. The drawback of this and the fact that the segment length is growing exponentially is that the user may end up with buffering over 50% of the video. The user

must also have enough bandwidth to receive more than one channel simultaneously. In the worst cast the user may end up streaming all the channels simultaneously, which happens in the case in figure 2.2. These two drawbacks of this algorithm give the user a high equipment demand with regard to available memory and available bandwidth.

2.4.2 Skyscraper Broadcasting

The Skyscraper Broadcasting [13, 8] algorithm is a little bit different from the Pyramid Broadcasting algorithm described above. The video segments are of equal length and more than one segment from the same video is repeated on a channel.

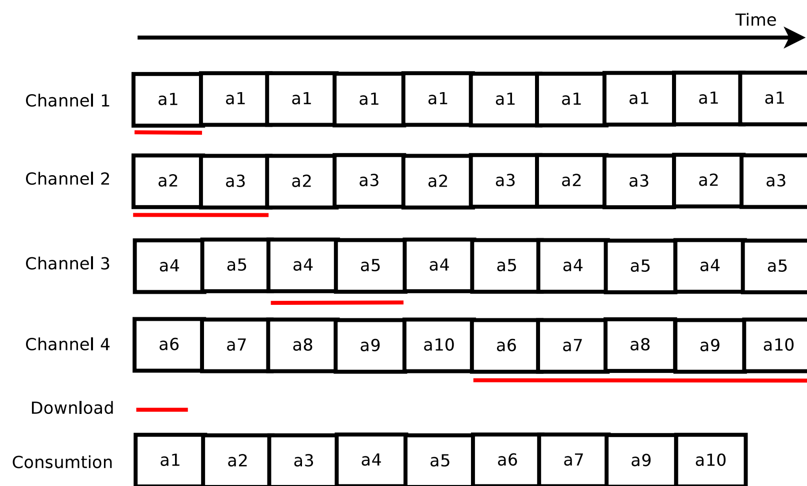


Figure 2.3: Skyscraper broadcast. *Adapted from [13]*

The number of consecutive segments that are to be played on each channel are determined by the series { 1, 2, 2, 5, 5, 12, 12, 25, 25, 52, 52, ...}, as seen in figure 2.3. As seen from the series, the first segments of the video are repeated more often that later parts giving a maximum playback start-up delay equal to the length of one segment. Compared to the Pyramid Broadcasting protocol mentioned in the section above, the Skyscraper Broadcasting protocol does not have such high demand on users equipment with regard to buffering and bandwidth. In this protocol, the bit rate is set to be equal to the playback speed. The user also receives at most 2 channels and never buffers more than 2 segments at a time. Since the segments are of equal size, the amount of data that has to be buffered on the user machine is never as high as with the pyramid broadcasting protocol.

2.4.3 Pagoda Broadcasting

The Pagoda Broadcasting protocol [13, 8] divides the video into equally sized segments like in the Skyscraper Broadcasting protocol, but how the segments are placed on different channels is a bit different. In the Skyscraper Broadcasting protocol, the segments are placed on channels in consecutive order. In the Pagoda Broadcasting protocol, the segments are not placed in consecutive order and consecutive segments appear on pairs of channels, as seen in figure 2.4.

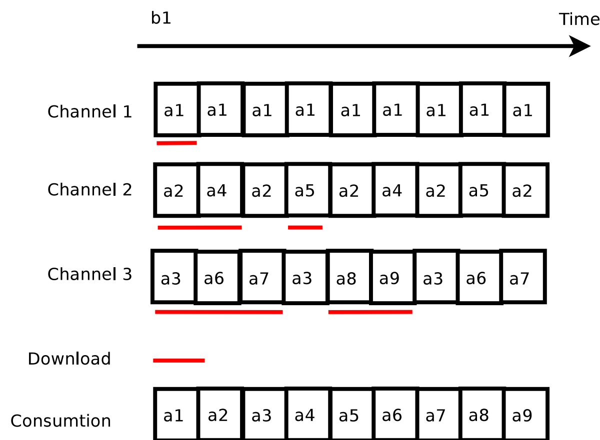


Figure 2.4: Pagoda broadcast. *Adapted from [13]*

The number of segments that is repeated on each channel in this protocol is given by the series {1, 3, 5, 15, 25, 75, 125}. The segments are placed on channels based on how often they need to be repeated. The first segment needs to be repeated every round to give a minimum startup time. The second segment is repeated at least every 2nd round and the third segment every 3rd round and so on. When the user can start receiving and consuming the first video segment it can start to receive the data from every other channel as well. The receiver receives the segment either ahead of time and store it in its buffer until it is played, or the segment is received and consumed at the same time. This protocol schedules many more segments per channel than the Skyscraper Broadcasting protocol, giving a lower bandwidth requirement for the server for each video broadcasted. On the other hand, this protocol does require that the user buffer 45% of the video.

2.4.4 Harmonic Broadcast

The Harmonic Broadcast protocol [13, 8] is also a protocol that divides the video into equally sized segments. Unlike the Pagoda and Skyscraper protocol, this protocol only

repeats one segment per channel. This protocol also lives by the concept that reception of later parts can be completed later than that of earlier parts, but not in the same way as the protocols mentioned above. With the Harmonic Broadcast protocol the user has to receive all of the channels concurrently, but not all channels are being broadcasted with the same speed. Later segments of the video are being sent at lower bit rates than earlier segments, following the harmonic series *1/n*, where *n* is the channel number, as seen in figure 2.5. This causes the server to have a low bandwidth requirement for each video because one video can be divided into hundreds of segments and for each consecutive segment the sending rate is getting lower.

Figure 2.5: Harmonic broadcast. *Adapted from [13]*

As with the other protocols mentioned, the user has to wait for a maximum time of one segment before it can start receiving the video, but a growing number of segments gives each segment a smaller size and therefore decreasing the start-up waiting time. The drawback of the Harmonic Broadcast protocol is that it sends a lot of data to the users before it is needed. In fact the client has to buffer about 37% of the video when more than 20 channels are used.

Another problem with this protocol is that a user may end up consuming a segment faster than it is received. All of the segments have the same initial start-up time. If a user requests a stream outside of this initial start-up time, the start of segment one may be in the middle of the sending of segment two. When the user then has finished receiving and consuming segment one it has only received and buffered the second half of segment two, and not the first half. Since the sending rate for segment two is lower than playback speed an error occurs. This problem can be solved by using the Delayed Harmonic Broadcasting protocol [13, 8]. By using the Delayed protocol the user is forced to wait the consumption of the first segment until the user have buffered it. This causes all the segments to

be buffered before consumption and no data arrives late. The drawback of this is that the maximum waiting time of a user is twice as long as the maximum waiting time with the original Harmonic Broadcast protocol. The article Video-on-Demand Broadcasting Protocols [8] introduces three other harmonic protocols that solve the problem of segments arriving late and use bandwidth more efficiently than Delayed Harmonic Broadcast. We now present a short description of these three protocols.

2.4.4.1 Cautious Harmonic Broadcast

In the Cautious Harmonic Broadcast [13, 8] protocol, the first segment is sent in the same way as in the original Harmonic protocol but the second and third segments differ. These two segments are both broadcasted on the second channel and at the same bit rate as the first segment, as seen in figure 2.6.

Figure 2.6: Cautious Harmonic broadcast. *Adapted from [13]*

The fourth segment and so on is being sent in the same way as in the original Harmonic protocol, following the harmonic series *1/n*, where *n* is the channel number. This means that all the segments are either received at full bit rate when needed or are already buffered by the client and ready to play. Because the three first segments are transmitted at full speed the total bandwidth requirement of this protocol is larger than with the original harmonic broadcast protocol.

2.4.4.2 Quasi-Harmonic Broadcast

The Quasi-Harmonic Broadcast [8, 15] protocol offers no additional bandwidth requirements, compared to Harmonic broadcast, but it has much higher complexity. The first

segment is transmitted repeatedly at full bandwidth. The transmission time of the first segment is regarded as one time slot. The rest of the segments are divided into sub-segments. For each segment, after the first, the segment is divided into *im-1* sub-segments for some parameter *m*. The user receives *m* fragments form each channel per time slot. These sub-segments are not sent out in order, and the first sub-segment is sent out twice as often as the other sub-segments, as seen in figure 2.7.

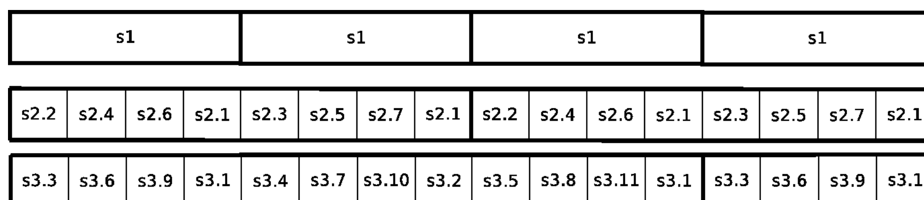


Figure 2.7: Quasi-Harmonic broadcast. *Adapted from [15]*

The point of this is that a user can consume a part of a segment and at the same time receive another part of the segment.

2.4.4.3 Polyharmonic Broadcasting protocol

The third protocol is the Polyharmonic Broadcasting protocol [8]. This protocol uses an integral parameter $m \geq 1$ and forces each user to wait *m* time before consuming the video. While the users are waiting it can still receive other segments. This allows later segments to be broadcasted with a lower bit rate than used in other harmonic protocols.

2.5 Patching algorithms

All of the periodic broadcast protocols have one thing in common. They all force the user to wait a period of time before the video playback can begin. Usually the maximum waiting time is the time it takes to send one segment. Patching is a way to solve this waiting problem, and in this section, we describe some of the patching algorithms.

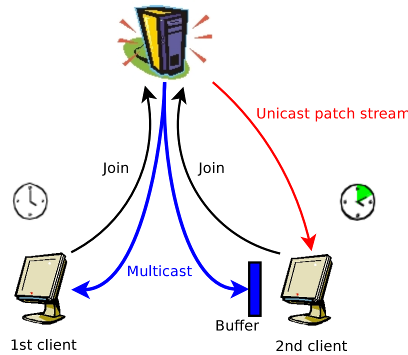


Figure 2.8: The concept of patching

The concept of patching [7], as seen in figure 2.8, is to allow user requesting a video stream to join to a multicast stream that is already broadcasting. The user has to receive two streams. One stream being the multicast stream that is stored in the user's buffer for later playback, and one unicast stream from the server that sends out the missing portions of the video from the start of the video and to the start of what has been buffered. As soon as the user's playback reaches the buffered content from the multicast stream, the unicast connection is torn down, and the user receive only the multicast stream. This is illustrated in figure 2.9.

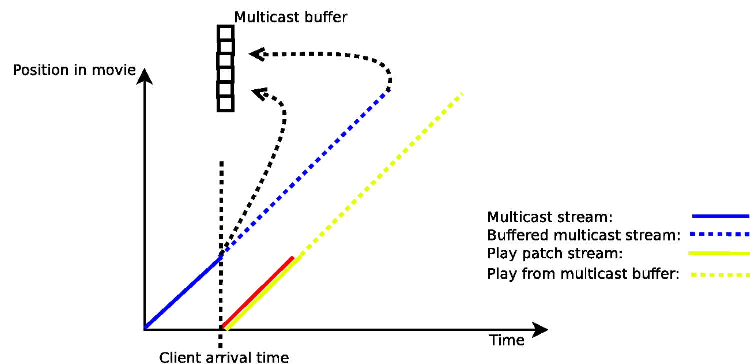


Figure 2.9: Receiving a patch stream. *Adapted from [13]*

The concept of patching truly gives the possibility of video on demand. The article "Patching: A Multicast Technique for True Video-on-Demand Services" [18] gives an introduction to some patching techniques and how they can be used together with batching in a multicast environment. This article does not present patching as a way for a single late arriving user to join a multicast stream, but instead presents how two batches of users using two multicast channels can end up using the same multicast channel.

When a server is to multicast a video, it uses a multicast channel. This channel is occupied with the video for as long as it takes to play back. This can give the server some performance problems as it limits the number of channels and the number of batches it can handle at the same time. If a batch requests a video, while another batch is already streaming the video, it must be a way to combine the two streams and not use two individual channels for the entire time of video playback. Article [18] presents a solution to this issue with the use of patching. Let's say a batch B1 has streamed a video for 3 minutes from a multicast channel C1. Then another batch B2 makes the same request for the same video. The multicast channel C2 is then activated to stream out the video for B2. Instead of having two channels streaming the same video for the entire playback time of the video, patching can be used to limit the number of channels used. The users in B2 have to listen to two channels at the same time. They receive and consume the multicast stream from C2 at the same time they buffer the stream from C1. When B2s playback reaches the start of the buffered content, C2 stops transmission of the video and both B1 and B2 listens to the same multicast stream from C1. C2 is now free and can be used to serve new requests.

2.5.1 Greedy patching and Grace patching

The article [18] presents the two strategies Greedy Patching and Grace Patching for deciding how long C2 shall stream the video when C1 is already streaming the same video. Greedy Patching uses the user's amount of free buffer space to determine how long C2 should stream. Lets say we have a 1 hour video. A batch of users arrive 10 minutes after the multicast of C1 has started and the users in the batch only have enough buffer space for 5 minutes of video. Then the users in the batch buffers the last 5 minutes of C1 and receive video from C2 up to that point. This means that for 55 minutes, two channels streams the same video. The alternative is Grace Patching. This strategy schedules a new multicast channel for the same video, if the user buffer is not large enough for the patch. When new batches then arrive later, this new channel is the last known channel to stream from and hopefully late arriving batches can join this channel. The drawback of Grace Patching is of course that it may result in many more regular multicast channels for the same video compared to Greedy Patching.

2.5.2 Periodic Buffer Reuse

The article ”Optimal Patching Schemes for Efficient Multimedia Streaming” [34] introduces two other algorithms called Periodic Buffer Reuse (PBR) and Greedy Buffer Reuse (GBR). PBR shares a restriction with the strategies mentioned above. That is that when a

new user requests a video it can only patch from the channel that last streamed the whole video. Compared to the strategies above PBR maximizes the amount of video a user can buffer from a channel by exploiting the user's buffer more effectively. When a user arrives *m* minutes after the start of a video stream, the strategies mentioned above only buffer the last *m* minutes of the video from the already existing multicast channel and create a patch. This means that the buffer may remain empty for a long period of time until the channel starts transmitting the last *m* minutes. With PBR the user receives video from the earlier transmission whenever there is available space in the buffer. Whenever the user needs to receive parts of the video from the server, the parts are received as late as possible, just before their playback times. When the user arrives M minutes into the stream with buffer space B minutes, and M > B, then the user must patch the first M minutes and at the same time buffer the next B minutes of the video from the ongoing multicast stream. At the time of M + B, the user buffer is full since the user is still playing video from the patch stream. At the time M-B, the user can start playback of the buffered content and at the same time receive additional parts of the video from the multicast stream. Of course, this leads to a new gap in the video and a new patch stream is required after B time. This goes on in a periodic fashion until the video streaming session is over.

2.5.3 Greedy Buffer Reuse

The other algorithm that article [34] presents is Greedy Buffer Reuse (GBR). GBR is an optimal patching algorithm. This algorithm allows a user to receive video from multiple ongoing transmissions to maximize the benefit of patching. GBR is a greedy algorithm as a user always receives video from ongoing multicast channels as long as there is free space in the buffer. This causes the user to reduce the amount of video it must receive from the server directly. "The GBR algorithm sequences through the frames of the video and schedules the client to receive a frame as late as possible - from a ongoing transmission (if possible), or directly from the server" [34]. There are two cases when a user must fetch the frames directly from the server. One being if no ongoing transmission includes this frame and the second being if the user does not have enough buffer space to store the frame from an earlier transmission. The key of this algorithm is that the user receives a frame as late as possible. Frames that are close to playback time do not occupy the buffer for a long time and other parts of the video can use the buffer. By using a reception schedule [34] and a channel schedule [34] for the user the system can plan what to put in the user buffer to limit the amount of video a user must get directly from the server. The GBR is said to be optimal as it minimizes the transmission bandwidth required for a group of users.

2.5.4 Lambda Patching

The article ”Tune to Lambda Patching” [14] also presents an optimized patching algorithm. This article is written as a response to the article [18] and presents two modifications of the patching technique called ”Lambda Patching” and ”Multistream Patching”. Regarding ”Lambda Patching”, the authors state that temporal distance between two multicast streams for one movie should not be determined by a client policy. Instead, it should be calculated by the server on a per video basis, since the server is aware of the average request inter-arrival time for each video. The calculation of optimal retransmission times for multicast streams is based on the measured inter-arrival time, which allows the server to tune the restart times for complete movies on a per stream basis and to tune the average number of required simultaneous server streams. We recommend reading the article [14] for further details regarding this calculation. With the other modification, ”Multistream Patching”, they demonstrate that the server load can be traded for client network bandwidth. The multicast streams are started cyclically on the server. The clients can buffer video data from multiple multicast streams at the same time, while they receive a unicast patch stream for the initial portions of a video. ”In contrast to the original technique, these cyclically started streams need not be complete video streams, but they can end when sufficient data from a running complete video stream has been received” [14]. For further details, we recommend the article [14].

2.6 TCP-based streaming

As we are going to use a TCP-based design of our patching system, and only for the patches, it is of great relevance for us to have a look at how TCP based streaming systems perform in the Internet. The use of TCP for multimedia streaming has been greatly disliked for many years and conventional wisdom is to use UDP as the transport protocol. The main reason for that is that the back-off and retransmission mechanisms in TCP can lead to high end-to-end delays, causing the media data to be delivered to the client to late. Because of this, much of the research over the last years has focused on developing UDP-based streaming protocols that provide mechanisms for TCP-friendliness and loss recovery. Despite all counter-arguments for using TCP in multimedia stream system, TCP is widely used in the commercial market. For instance, Real Media [32], Windows Media [28], Smooth Streaming [27], Move [29] and Apple HTTP Live [3] which are well known multimedia streaming systems in the commercial market, use TCP. TCP has the advantage that it is TCP-friendly and due to the reliability of TCP there is no need for loss recovery at higher levels [37]. In this section, we have a look at some related work that focuses on multimedia streaming using the TCP transport protocol. All results and thoughts presented in this section are based on the articles that are examined.

2.6.1 Multimedia Streaming via TCP: An analytic Performance Study

The article ”Multimedia Streaming via TCP: An analytic Performance study” [37], seeks to answer the following question: Under what circumstances can TCP streaming provide satisfactory performance? In order to answer this question, the authors use a baseline streaming scheme that uses TCP directly for streaming. This scheme is similar to HTTP streaming and is referred to as *direct TCP streaming*. The authors have developed discrete-time Markov models for live and stored video streaming using TCP. In live video streaming the server generates video in real time and is only able to transmit the part of the video that has been generated. This means that the transmission is constrained by the generation rate of the video. The authors refer to this as *constrained streaming*. In stored video streaming the whole video is generated before it is transmitted. The video is transmitted as fast as possible only limited by the TCP throughput in order to fully utilize the available bandwidth. They refer to this type of streaming as *unconstrained streaming*. They explore the loss rate, round trip time and timeout value in TCP and the video playback rate to find out under what conditions direct TCP streaming gives satisfactory performance.

The authors have varied parameters (i.e., loss rate, round trip time) in constrained and unconstrained streaming to study the impact of these parameters on performance and identify under which conditions TCP streaming provides a satisfactory viewing experience.

Conditions for satisfactory performance: The authors state that the viewing experience is satisfactory when the fractions of late packets is low for a short startup delay, and that studies have shown that the video quality drops when the packet loss rate exceeds 10^{-4}. As a consequence of that, they assume that the performance of direct TCP streaming is satisfactory when the fraction of late packets is below 10^{-4} for a startup delay of around 10 seconds. The achievable TCP throughput is denoted as T packets per second. The playback rate of the video is denoted as μ. T/μ represents how much the achievable TCP throughput is higher than the video playback rate. The authors state that the performance of TCP streaming improves as the T/μ increases since packets are accumulated in the client’s local buffer faster relative to the playback rate of the video. The impact of T/μ on the performance of TCP streaming is explored next, for both constrained streaming and unconstrained streaming.

Constrained streaming: The playback rate of the video, μ, is set to be 25 packets per second and the round trip time is varied such that T/μ ranges from 1.2 to 2.4. By increasing T/μ, the authors observe a diminishing performance gain. The performance is improved

dramatically as the T/μ is increased from 1.2 to 1.6 and less dramatically as it is increased from 1.6 to 2.4. This indicates that, to achieve a low fraction of late packets, the required startup delay is very long when T/μ is only slightly higher than 1 and reduces quickly as T/μ increases. It is observed that under various settings, the performance becomes satisfactory when T/μ is roughly 2. In another scenario, they had round trip times of 50, 100, 200 and 300 ms and varied the playback rate of the video such that T/μ ranges from 1.2 to 2.4. Again, a dramatic performance gain is observed when T/μ increases from 1.2 to 1.6 and less dramatic from 1.6 to 2.4. They then investigate the required startup delay such that the fraction of late packets is below 10^{-4} when $T/\mu = 2$. When the round trip time is 50 ms, the required startup delay is no more than 10 seconds under all settings. When round trip time is increased to 100 ms, they get the same result except when they experience very high loss rate and high variations in round trip time. For a long round trip time, high loss rate and high time out value, the required startup delay is in tens of seconds.

Unconstrained streaming: The authors have performed tests with the same scenarios as with constrained streaming. First they set the playback rate of the video to be 25 packets per second and vary the value of the round trip time such that T/μ ranges from 1.2 to 2.4. As with constrained streaming the results are similar. A diminishing performance gain is observed when increasing T/μ and the performance becomes satisfactory when the achievable TCP throughput is twice the video bitrate. The required startup delay is bounded within 10 seconds when T/μ increases to 2. In the next experiment they, as with constrained streaming, set the value of round trip time to 50, 100, 200 or 300 ms and vary the playback rate of the video such that T/μ ranges from 1.2 to 2. Under relatively short round trip time the required startup delay is within 10 seconds for all the settings. For a long round trip time, high loss rate and high timeout value, the required startup delay is in tens of seconds.

In this article [37], the authors have developed models for constrained and unconstrained streaming and provided guidelines as to when direct TCP streaming gives satisfactory performance. TCP generally provides good streaming performance when the achievable TCP throughput is roughly twice the media bitrate, with only a few seconds of startup delay. For large round trip times, high loss rates and timeout values, either a long startup delay or a large T/μ is needed to achieve a low fraction of late arriving packets.

2.6.2 Quality-Adaptive Media Streaming by Priority Drop

The article "Quality-Adaptive Media Streaming by Priority Drop" [22] presents a general design strategy for streaming media applications in best effort computing and networking

environments. The authors present priority-progress streaming (PPS), which is a real-time best effort streaming protocol. An implementation has been made and a prototype video streaming system that incorporates priority-drop video, priority mapping and priority-progress streaming has been released. Their system demonstrates a simple encode once, stream anywhere model, where a single video source can be streamed across a wide range of network bandwidths while maintaining real-time performance and gracefully adapting quality. Their strategy revolves around the idea of using priority data dropping, priority drop for short, as the primary means of adaptation. When using priority drop, the basic data units of the media are explicitly exposed and appropriately prioritized. By dropping data units, based on priority-order, the media quality is gracefully reduced.

The authors have developed a minimal scalable compression format, which they call Scalable MPEG (SPEG). This was implemented to test priority mapping and PPS using real video. SPEG is derived from MPEG-1 video. In MPEG video each video frame is broken down into blocks of 8x8 pixels. These blocks are converted to corresponding 8x8 blocks of coefficients using the discrete-cosine transform (DCT). Quantization, which is a strategic removal of low order bits, is then performed on these blocks, and is the basis for compression gains in MPEG. SPEG transcodes these MPEG coefficients to a set of levels, one level, being the base level, and three enhancement levels. The coefficients from each level are used to form layers. There are four layers per original MPEG frame, and these layers are the basic application level data units (ADUs) in SPEG. We are now going see how these layers is used in their priority drop scheme.

Based on the nature of the content, nature of the viewing device and the personal preferences of authors, viewers etc., we get different adaptation policies. A sports program might want to prioritize preserving fidelity of motion rather than color fidelity. A user with a PDA would not be that interested in high resolution, compared to a user with a full sized screen. By using an algorithm called priority mapping, video compression is decoupled from the final adaptation. This opens the possibility that each piece of content may be adapted in different ways for different scenarios, with far lower complexity than actually recompressing or transcoding the content during streaming. The priority mapping algorithm translates the different policy specifications into appropriate priority assignments on data units of priority-drop video. The idea is that this would make the content more re-usable. The priority mapper takes application data units (ADUs) and the quality adaptation policy as input. With this input, it generates streaming data units (SDUs) as output, which are aggregates of prioritized ADUs. Mapping windows are used to group ADUs. A mapping window is an interval in the timeline of the media stream. The ADUs in each window are prioritized separately. For every ADU within a mapping window, the mapping algorithm finds the order in which ADUs may be dropped that has the minimum impact. When all ADUs have a priority, they are grouped into SDUs, one per priority level. The SDUs are all set to have the timestamp of the first ADU in the

window. The purpose of the priority mapper is to isolate the PPS algorithm, which is introduced next, from the low level details of the video format.

The authors present an algorithm for real-time best-effort streaming called PPS. With unpredictable throughput, PPS combines data re-ordering and dropping to maintain timeliness of streaming. This algorithm is best effort as it lets the congestion control algorithm decide the appropriate sending rates. The timeliness of the stream is maintained by dropping low-priority data units at the sender when the sending rate is low. This causes the higher-priority data sent to automatically match the rate decisions of the congestion control mechanism. The PPS algorithm takes the SDUs produced by the priority mapper and uses their timestamp and priority labels to perform real-time adaptive streaming over a TCP-friendly transport. The PPS algorithm subdivides the timeline of the video into disjoint intervals called adaptation windows, which consists of one or more mapping windows. The algorithm for PPS contains three sub-components; the upstream buffer, the downstream buffer, and progress regulator. The upstream buffer admits all SDUs within the time boundaries of an adaptation window. These boundaries are determined by the progress regulator. Each time the progress regulator advances the window forward, the SDUs that have not been sent from the old window are expired and the window is populated with SDUs from within the new boundaries. The SDUs flow from the buffer in priority-order through the network to the downstream adaptation buffer. The downstream buffer collects the SDUs and reorder them into timestamp order. The progress regulator is notified whenever the SDUs arrive late because of unexpected delays in the network. The regulator can then avoid late SDUs in the future. The downstream buffer receives as many SDUs as the bandwidth of the network allows and the rest, which are of lowest priority, are dropped at the server. In this way the quality is adapted to the network conditions between the server and receiver.

The size of the adaptation window has an effect on buffer size requirements, startup delay and consistency in quality. Smaller windows give shorter startup delay because the algorithm does not allow display of a window until transmission is fully completed. Larger windows gives fewer quality changes and larger rate fluctuations can be smoothed out. To achieve both benefits from large and small windows, the PPS algorithm includes the option of changing the window size during streaming. This means that we could start with a small window to get a short startup delay and then gradually increase the window size to reduce quality changes. This is possible because the PPS algorithm can transmit the video at a faster rate than it is consumed at the receiver.

For details regarding implementation of the priority mapping algorithm and the PPS algorithm and the presentation of the media format Scalable MPEG that the authors use, we recommend studying the article [22].

For their experiment setup they have used a group of Linux based PCs acting both as

server and as end hosts in a dedicated network testbed that implements a saturated network path. They have set the delay to 50ms round-trip-time and a bandwidth limitation of approximately 25Mbps. In addition to that, the network is saturated with competing traffic. The experiments displays the effect of adaptation window size. The results point out that by gradually increasing the adaptation window size during the stream, the quality changes are more consistent than when using a fixed window size, even though the startup latency is only in the range of one second. The authors claim that their results show that priority drop is very effective. A single video can be streamed across a wide range of network bandwidths, heavily saturated with competing traffic and at the same time manage to maintain real-time performance and gracefully adapting quality. The PPS algorithm makes full use of the available bandwidth while maximizing the average video quality.

2.6.3 DAVVI

DAVVI [20], is a system for delivery of multi-quality video content in a torrent-similar way, which also provides a brand new personalized user experience. By the use of search, personalization and recommendation technologies, end users can efficiently search and retrieve highlights and combine arbitrary events in a customized manner using drag and drop.

It uses torrent-like HTTP streaming. In torrent-like HTTP streaming, the video is split up into smaller segments, where each segment is stored on web servers as a separate file. Segments belonging to the the same video can be stored in different web servers.

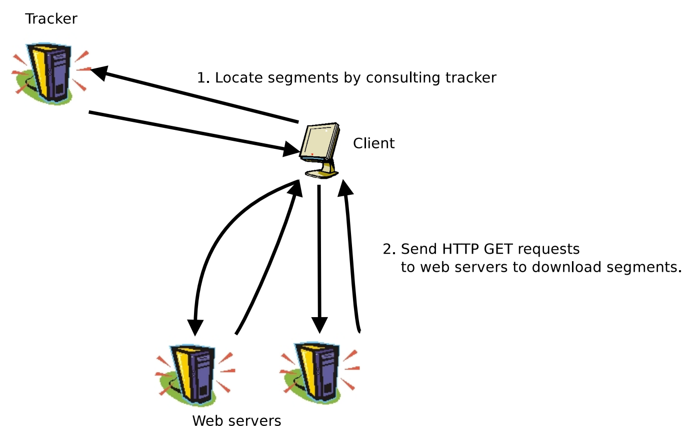


Figure 2.10: Torrent-like HTTP streaming.

As illustrated in figure 2.10, the client consults a tracker, which specifies the location of the segments that are to be downloaded. The client downloads the segments, by using

HTTP GET requests, and plays them in the order specified in the playlist. In contrast to real-time streaming, where the server controls the transmission rate of a video, the server has no control over the transmission rate of a segment in HTTP streaming. The download time of a segment from the server is limited by the network resources. Generally, a client downloads a segment and plays it, while it starts downloading of the next segment until it reaches the end of the playlist. The client downloads a certain quality based on the clients network resources. The playback time of a segment is equal for each segment, regardless of quality, but the size of a segment differs. Segments with high quality is larger than segments with low quality. Because the video are divided up into small segments, it is easy to change quality, as illustrated in figure 2.11. If a client detects that it needs to change to a lower quality, this can be done in the length of a segment.

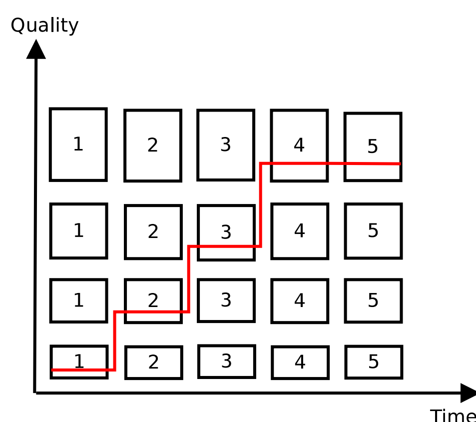


Figure 2.11: Quality adaptation in torrent-like HTTP streaming.

The DAVVI system components can be categorized into three different classes: Segmentation and dissemination, annotation and search and recommendation.

2.6.3.1 Segmentation and dissemination

The system architecture aims at serving the clients with video, that has been split up into two-second segments from multiple sources. These two-second segments are coded in several qualities so clients can achieve adaptation to available resources and reduced latency when jumping within or between streams. Each segment is a self-contained video which enables the user to retrieve arbitrary segments and continuously play them out in any order. The video format currently consists of closed group of pictures using H.264 video and MP3 audio. The segments do not contain any header information. Instead, the

video header is generated at the client side. By doing this the bit rate can be reduced with about 10% depending on rate, without causing any loss of quality.

The segments are distributed to web-servers in the Internet and a torrent-like approach over HTTP for data delivery is used to download the segments in the order of the playout. A tracker is used to hold up to date information about the individual segment's location. Before a client can start downloading a video it first has to contact this tracker to get information about where the segments are to be found.

The DAVVI media player has a very short client waiting time and is able to jump quickly between segments. The short client waiting time is achieved by downloading the first segments in a low quality. The quality are then increased for the next segments if enough resources are available.

2.6.3.2 Metadata and annotation

The authors have developed event extraction tools to analyze and annotate the videos. This is used to identify events like goals, crowd cheering, etc. Services like live text commentary of football matches are made available on the Internet by many TV broadcasters and online news papers. This unstructured commentary text is converted to annotation metadata by using a crawler that parse these pages. This makes it possible for the users to search and query for game events using a rich set of keywords like ”volley”, ”sliding tackle”, ”offside”, etc. With this system, they can then use timestamps to identify the respective video interval for an event and retrieve data some seconds before and after the annotated time.

2.6.3.3 Search and recommendation

The users can query for a broad range of events using keywords found in the live text commentary, based on the metadata aggregation, video annotation and indexing. When a user places a search, a playlist in the form of an event description, video object identifier and a time interval is returned to the user. This playlist is then sent to the video dissemination system, which retrieves the respective video segments.

2.7 Summary and discussion

In the previous section we have gone through a lot of relevant related work including multicast, periodic broadcast algorithms and patching algorithms. There has been done a

lot of research on these topics where results have been analyzed. Many of these articles have retrieved results based on calculations on paper and not from experiments in real life. The ones that we are familiar with, have performed experiments with simulations and not considered practical issues like packet loss, jitter, reordering and bandwidth limitations in the network, issues with sequential processing at server end and how much overhead problems like these can cause. Another issue one may find interesting is if any of these algorithms can be used to keep up an acceptable stream for the user when network congestion may occur.

As mentioned in section 2.3, multicast does perform better when it comes to issues like network and server resource use, but it also adds a number of challenges that have to be dealt with. The one that most users notice and keep in mind when wanting to stream a video is the start-up delay that multicast causes. By using stream scheduling techniques like periodic broadcast or patching the start-up delay can be kept to a minimum and these techniques also make the system more scalable because late arriving users can join ongoing multicast streams instead of setting up a new multicast channel to stream the whole video again. Another challenge with multicast is packet dropping. What if a single user in a multicast group does not receive a packet? It is unlikely that the server retransmits that packet on the multicast stream so all the members of the multicast group receives it one more time. Since the Internet is a best effort service and by using UDP to transmit the video it is not guaranteed that packets are delivered. This has to be solved at application level. Would it be possible for a user to directly ask the server to retransmit any packets it does not receive? This could work when the content is being buffered but would not work that well when the user is receiving and consuming packets at the same time.

As mentioned above, periodic broadcast schemes together with multicast can be used to save resources and to minimize the start-up delay that multicast streaming alone gives. In section 2.4, we have presented some periodic broadcasting algorithms and found out that most of them have both negative and positive effects compared with each other. The article ”Video-on-Demand Broadcasting Protocols” [8] presents a comparison between the different algorithms we have presented in section 2.4. The authors have compared user’s bandwidth requirements, user storage requirements and start-up delay time. The authors have not concluded which protocol that is the best, as all have their advantages and disadvantages in terms of bandwidth requirement and client buffering.

Patching together with multicast is way to reduce start-up delay and gives users the possibility to join ongoing multicast streams when they are arriving late. The articles ”Patching: A Multicast Technique for True Video-on-Demand Services” [18] and ”Optimal Patching Schemes for Efficient Multimedia Streaming” [34] present the different patching algorithms we mentioned in section 2.5. The latter article compares the algorithms presented in the first article, which it refers to as Restricted Buffer Use (RBU),

with the new algorithm, Periodic Buffer Reuse (PBR), that is said to have better usage of the user buffer space. The latter article then compared its two algorithms PBR and GBR with each other. In the comparison between RBU and PBR on client arrival rate, the PBR algorithm has a much less data transmitted per client meaning that it requires a lot less bandwidth to satisfy the users. The article [34] then compares the PBR and the GBR algorithm concluding that the GBR algorithm offers a sizable reduction in transmission overhead, lowering the server bandwidth requirements. These comparisons are only done with simulations and not in real life experiments.

We have also been looking at some related work regarding media streaming using TCP. This has been important for us since our patching functionality use TCP for data transfer. It was interesting for us to see what kind of results others have retrieved from using TCP for media streaming and if there are any limitations that TCP streaming introduces that would make it impossible for us to use it for patch data transfer. All of the three articles we presented in the related work section have given us proof that TCP actually is quite suitable for media streaming. Our patching functionality is based on the torrent-like HTTP streaming system described in the article [20], but as we see in chapter 3 and 4, we do not take all the functionality the media player has to offer into use. The video format used in the DAVVI media player is suitable for us as it makes it easy for us to retrieve the data we need from the web server system, either if it is a little data due to packet loss in the multicast stream or if we stream directly from the web server system when we have a client that arrived late to the multicast stream. We also use their scheme for quality adaptation where each segment is available at the web servers in several qualities. The media player they have implemented is used to play out the video on the client.

Thus, in order to address challenges like packet loss, quality adaptation, scalability and network resource heterogeneity amongst clients, we extend the traditional patching scheme with support for quality adaptation by jumping between channels. Moreover, we integrate this scheme with a scalable, torrent-based HTTP streaming system to deliver the patch and recovery streams. In the next chapters, this design and implementation is described.

Chapter 3

Design

In this chapter, we present the design of our system, required for the implementation of the patching algorithm and the solutions to issues we faced. We start this chapter by presenting an overview of our system and then go into the design details of our multicast server, patch server and client, respectively.

3.1 System overview

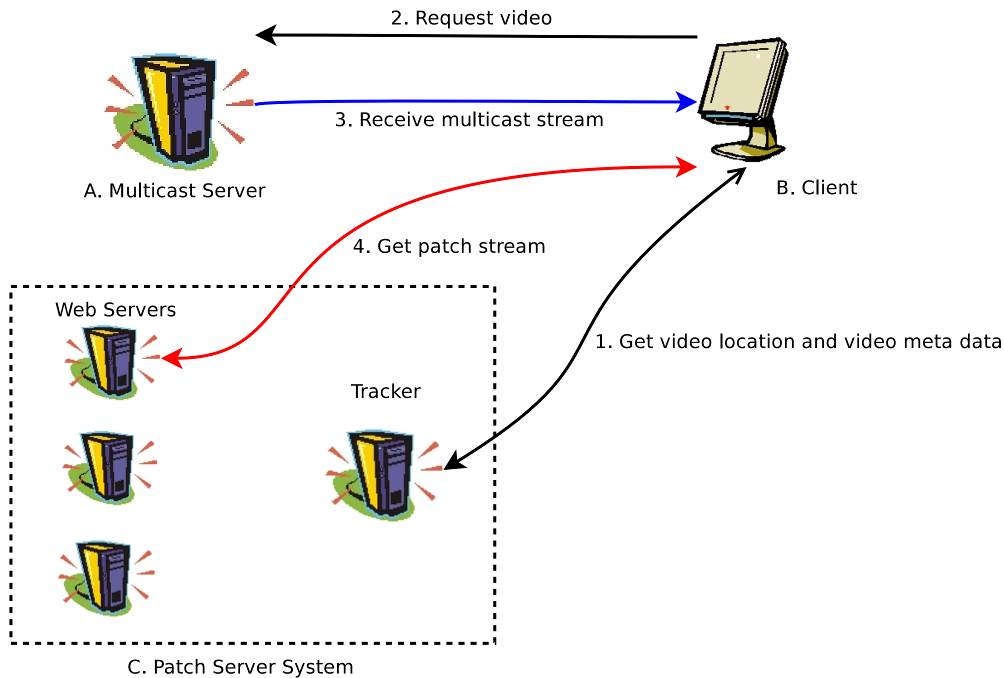


Figure 3.1: An overview of our system

In our patching scheme, we have 3 different main components as seen in figure 3.1. The multicast server, the patching server system and the client. The multicast server provides the client with a video stream over a multicast channel. This server acts on request from a client. If a client wants to stream a video, the client has to send a request to the multicast server. If the requested video is not already transmitting, the multicast server starts a new transmission of this video and sends it out on a multicast channel. In the Internet, it is not guaranteed that the network bandwidth is the same for every client. The server must therefore support the quality adapted playback concept. To be able to give the clients the best viewing experience, the server creates four different multicast channels for each video to be transmitted. Each one of these channels transmit the video in a given quality. Which one of these channels the client chooses to listen to is based on client's bandwidth limitations and processing power.

The other component is the patching server system and is based on the system presented in the article [20]. The patching server system consists of many web servers placed in various locations around the world, and a tracker. The tracker holds up-to date information about the location of the requested video segments. The tracker informs the client where

the video is to be found, and the client can then connect to the appropriate web server if a patch stream is needed. In addition to this, the tracker also provides the client with video meta data for header generation on client side. The patch server provides a client with segments if a client finds out that it misses some parts of the video from the multicast stream. A client can miss segments either because of packet loss, or that it joined an ongoing multicast stream and missed the first part of the video. If a client needs a patch, then the client consults a tracker to find out which web server that can provide the patch. The client then sends a HTTP GET request to the appropriate web server. When the web server get this request, it transmits the patch to the client. Unlike the multicast server, the web servers use unicast and serve any connecting client independently. As mentioned in section 2.6.3, the video is being sent from the server to the client without any video header information to reduce the bit rate. The web server also provide each video in four different qualities so the user can download the best quality as possible.

The last component is the client, which receives and plays video. Our client design differs from the client design in other articles about the patching scheme. The popular design choice is to let the multicast server determine if a client needs a patch stream or not. The server has to estimate how much of the multicast stream the client missed. This estimate is difficult to get accurate and the client can end up getting some of the data twice or not receive all of the data. In our solution, it is up to the client to determine which parts of the video it misses. With this approach, the client can request a patch for exactly the data it missed from the multicast stream, as the client has full control over the segments it has received from the multicast stream. Before a client can stream a video, a client setup phase is required. The client must first contact the tracker, mentioned above, to get video location information if a patch is ever needed. The client then sends a video request to the multicast server, which starts a multicast of the video unless it already is transmitting the video. To be able to receive the whole video stream, the client must first find out if it is arriving late to the multicast stream. If it is a late arriving client , it must contact the appropriate web server on the location provided by the tracker in the client setup phase, and starts a download for the part of the video stream that it misses. The client then starts the playback of the patched stream while it buffers up the ongoing multicast stream for later playback. As described later in this thesis, the client also requests patches from the patching servers if it discovers packet loss to repair the missing parts of a segment.

We move on to the design of each component. We first, present the video stream format that use before moving on to server side and client side design choices.

3.2 Video stream format

Our system is using a variant of the H.264 standard for video. We use this standard because the system that implements the servers transmitting the patch stream and the video player on client side uses this standard. This system is called DAVVI [20] and was introduced in section 2.6.3. The video to be transferred are split into segments of 2 seconds each. Each one of these 2 second segments is a self-contained video. The media format consists of closed group of pictures using the H.264 video standard with MP3 sound. To reduce the bit rate by about 10%, the media data is packed into custom made containers. For instance the MPEG transport stream contains a 4 byte header for each 184 byte of payload in every packet transmitted over the network. The custom made container used in this system does not contain any header information, so every byte received by the client is raw media data. The receiver of the video has to generate the header and apply it to the video segment before playing the segment in the player module. Both our multicast server and the patch server use this video stream format, and are able to provide the user with the same segments.

3.3 Multicast server design

As mentioned, the multicast server provides the main video stream to the clients over a multicast channel on client request. In this section, we go through the different design choices we made. We also go through issues we had to consider, on server side, to be able to transmit the video to clients in the best possible manner.

3.3.1 Channel creation

To be able to transmit our video to the client, the server first of all need to create channels. These channels are addressable resources used to send data from our multicast server to the clients. As argued in section 2.3, unicast introduces many performance issues when the number of connected clients are growing. In other words, it does not scale well without massive upgrades on the server side to be able to cope with the growing number of clients. To overcome these performance issues introduced by unicast, we make use of multicast and its advantages.

As mentioned in section 3.1, the videos come in four different qualities. To support the quality adapted playback scheme, the server therefore needs four different channels, i.e.,

one for each quality. When setting up the channels, we faced different alternatives that the multicast server could take into use. Should the server use the any-source multicast model (ASM) or the source-specific multicast model (SSM) [5]. In ASM, there can be multiple senders to one multicast address. The network is responsible for discovering all the hosts sending to that address and to route all of the data to all interested clients. ASM is suitable for applications like video conferences, where all the participants want to be aware of all other participants. In SSM, the client still has to register its interest for a multicast address, but in this model, only one specific source is sending data to this address. In addition, the SSM model has some network performance advantages over ASM. With SSM, the amount of routing information the network has to maintain is reduced compared to ASM. Since we only have one source for the multicast stream in our system, the SSM model suites us very well.

When allocating addresses to the different channels, we also faced some alternatives. One alternative is to use the same address for all the channels and only use different port numbers to distinguish them. With this approach the server has to only allocate one address per video to be sent. This solution is very reasonable to use considering the problem of the limited number of available IP addresses in the Internet. Unfortunately, routers operate on the network layer. For that reason, they can not route a packet based on a packets port number, since the port number belongs to the transport layer. This means that if the server sends the data out on channels with the same IP address, then the routers sends data from all channels to every interested client at all times during video transmit. We then use much more bandwidth than necessary because a client only listen to one stream at a time but four streams is sent to it. To assign different IP addresses to each channel is therefore the best choice for our system.

The server also has to assign port numbers to its channels. There are two alternatives we can choose. Either give the same port number to each channel or to give each channel different port numbers. The first choice limits the possibility for assigning an already occupied port number to a channel and this is the solution we have chosen. It has been reported that some versions of the Linux kernel have problems with separating multicast channels with different IP addresses but with same port numbers. Our client never registers their interest in more that one channel at a time, so this is not a problem for our implementation.

3.3.2 Channel management

The four channels the server has to create for the video transmission are created within a single thread. The server could have created four different processes since all channels could operate within their own environment and never share any memory, but if the server

were to use a separate process for each channel, it would experience a significant memory overhead and excessive context switching if a server is to transmit many different videos. The channels need to be synchronized so they start sending a segment at the same time. This is important because the client changes which channel it listens to when adapting quality playback to its network resources. If the channels were not to be synchronized at the start of sending a new segment, then the client could risk missing a lot of data if channel one were transmitting segment five and channel two were transmitting segment seven.

3.3.3 Transmission of segments

Clients that are to receive real time media streams have some requirements that are important to fulfill in regard to transmitting the segments. The main criteria is the delivery time and the delivery order of the data packets. If packets can not be delivered on time it is not necessary to deliver the packet at all, and if the packets arrive out of order the video is not displayed correctly at the client end. To be able to fulfill these criteria, our channel design must reflect this. We can either chose to transmit our data from the multicast server using transport protocols TCP [19] or UDP [30][35]. An mentioned 2.6 in, TCP is a reliable protocol which guarantees the delivery of data and that the data delivered is in correct order. If packet loss or corrupted packet is detected, TCP retransmits the faulty packets. UDP is an unreliable protocol. It has the drawback of not guaranteeing the delivery of packets or that the packets are delivered in the order they were sent. In despite of this drawback of UDP, UDP is our choice for transmitting the segments on the multicast channels. The reason, as mentioned in section 2.6, is that the guarantees that TCP gives, regarding reliable transfers with the use of ACKs for each packet sent, produce a lot of overhead which slows the overall transmission rate. The client dos not need to have a lost or corrupted packet retransmitted if the retransmitted packet arrives after the deadline for delivery. Another reason, as mentioned in section 2.3, is that UDP is currently the only transport layer protocol standardized by the Internet Engineering Task Force (IETF) that supports multicast. With UDP, we have a risk of losing packets or getting packets out of order, also known as packet reordering, but the client has to cope with this. On top of UDP, the server use RTP [17], which is a protocol that provides end-to-end delivery of real-time data. The RTP protocol offers services like defining the data to be transmitted, time stamping and sequence numbering. As we see in chapter 4, we do not directly use the services provided by the RTP protocol to ensure correct playback at client end. Instead of using RTP, we have another solution based on our own application header, which is added to the network packets before transmitting them. The reason we use the RTP protocol is because it is well integrated in the Live555 [24] library, that we use to implement the server.

3.3.4 Segment delivery time

The segment delivery time is an important issue in a media streaming system. The client is dependent on receiving the data within a certain time limit, depending on startup delay and the playback speed of the video. When determining the transmission speed of a single segment, we have to consider the server load, client buffer space and the playback time of the segment. From the server's point of view, it could either send out the data as fast as possible or as slow as possible. If it were to send out the data as fast as possible, then the channel sending the video is freed much faster, not occupying the server's attention or memory space. On the other hand, the channel occupies a lot of the server bandwidth making it hard for other channels to send at the same time. It also requires a lot of buffer space on client side. We went for client friendly approach, with regard to client buffer space. To minimize the need for client buffer space the server has to send the data out as late as possible. How late the data can be sent is limited by the playback speed of a segment, the client startup delay and how much the client chooses to buffer. A segment takes two seconds to play on client side, so the speed of sending a segment out on a channel is as close to that as possible to limit client buffering. Since segments of different qualities are of different sizes, our channels would have to support dynamic sending bit rate. It should take the same amount of time to transmit a segment regardless of its quality, as illustrated in figure 3.2. Each segment must be split into smaller packets to be sent over the network. The size of a packet must not be larger than the size of MTU, but we want the packet sizes to be as close to the MTU as possible. To be able to maintain a sending rate of 2 seconds for a segment the server has to wait some time between each packet sent. The time the server has to wait is determined by how many packets a segment consists of.

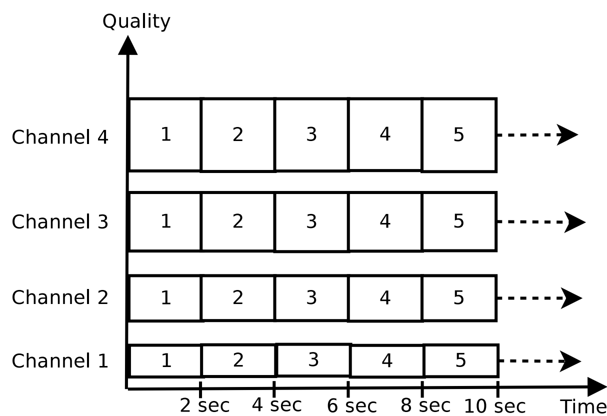


Figure 3.2: Transmission of segments in different qualities on multicast channels.

3.4 Patching server design

The system we use as a patching server is DAVVI [20] presented in section 2.6.3. In this section, we do not go into the design in details but, present the main features. For detailed information about the design we recommend the article [20], regarding the web servers.

The system is designed to provide users with multi-quality video streams and to make it easy to retrieve arbitrary parts of a video. The video stream format is the same as described in section 3.2 and in our case, comes in four different qualities.

By using web servers to download a patch stream, we reduce the load on the multicast server as the multicast server does not need to send an individual patch stream to each client and at the same time transmit its multicast streams. By using the web servers and downloading the segments with HTTP GET requests, there are no need for the web servers to keep control over which clients that are connected at all times. With HTTP GET RANGE request, the client can specify which part of the segments it is missing if it needs to download a patch due to packet loss in the multicast stream, as seen in figure 3.3. If the client misses 1500bytes of a segment, the client could just download those exact bytes and need not to download the whole segment from the patch server. It would be a massive extra load on the multicast server if all clients were to contact it for each packet they missed.

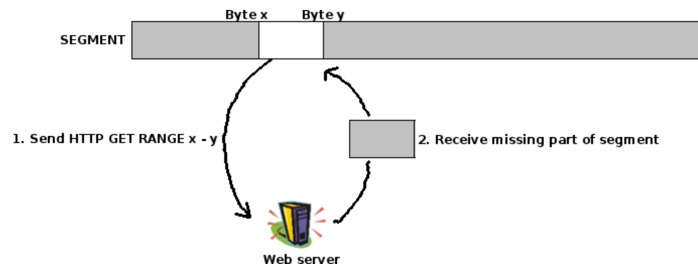


Figure 3.3: HTTP GET RANGE request.

There are great advantages in using the two second segments for patching purposes. With this concept it is very easy to detect which part of the stream that is missing and the client can download the correct segments for the patch stream without making a lot of calculations. With a two second boundary the client hits the patch stream quite accurate, regarding the parts of the video that we are missing. By using these segments, it is very easy to quickly adapt to a new quality. When the client discovers that it needs a lower quality this can be changed in two seconds, as illustrated in figure 2.11.

The patch system is called by the client whenever it is in need of a patch, either because the client is arriving late to the multicast stream or because it has encountered packet loss. These cases is further presented in the next section.

3.5 Client design

In this section we present the different design choices we made and issues we had to consider on the client side to be able to present video in the best possible manner to the user.

3.5.1 Client overview

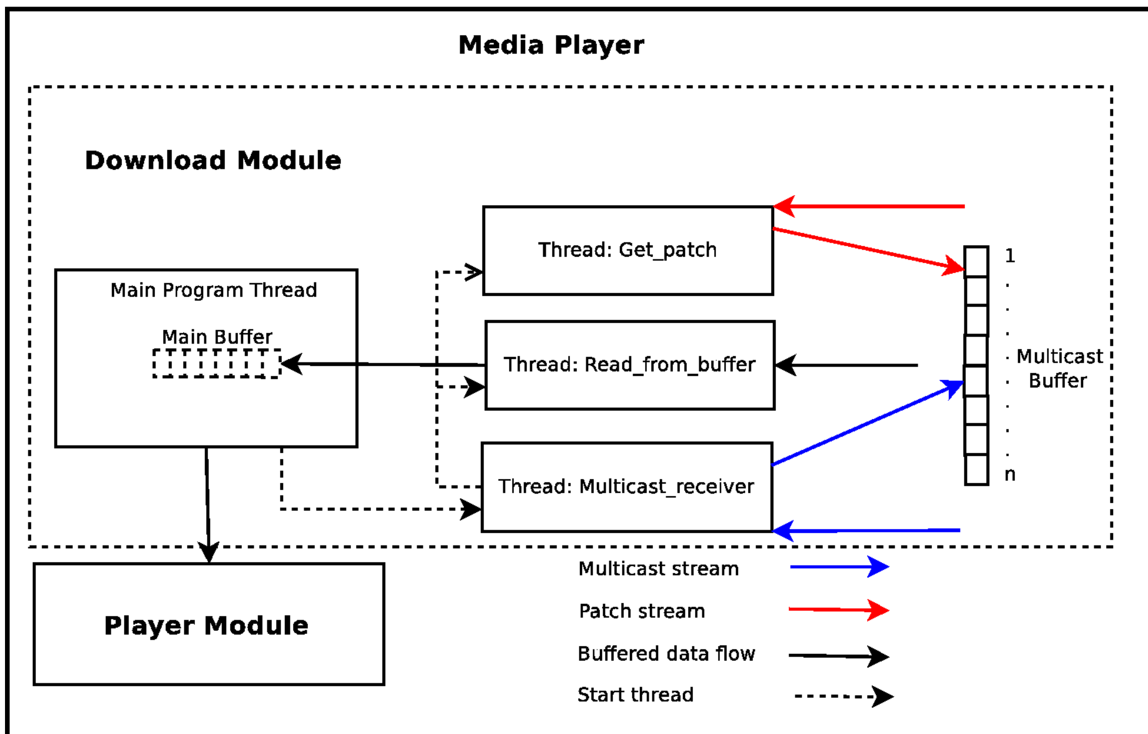


Figure 3.4: Client overview

The client in our system actually consists of two separate components, as seen in figure 3.4. One component, the download module, is responsible for downloading the video

and send it to the other module, the player module. The player module is used as described in [20], but we have extended the functionality of the download module so it is able to handle requirements like incoming multicast traffic. The download module is able to receive a multicast stream, receive a patch stream and deliver the video to the player module simultaneously. We have decided to create three different threads, in addition to the `main_program` thread, to handle the download module's responsibilities. The responsibilities of the `main_program` thread are to start the `multicast_receiver` thread, run the client setup phase, handle requests from the player module and send video to the player module whenever video data is made available by the `read_from_buffer` thread. Requests sent from the player module can be a playlist for a video to play or a request for a change of quality. The `multicast_receiver` thread is responsible for contacting the multicast server and to handle the incoming multicast stream by storing it in the correct location in the multicast buffer. This thread also starts the `get_patch` thread and the `read_from_buffer` thread. The `get_patch` thread contacts the appropriate web server and downloads the missing parts of a video. It is the responsibility of the `get_patch` thread to detect if some segments or part of segments are missing. The downloaded parts from the `get_patch` thread is stored in the correct location in the multicast buffer. The `read_from_buffer` thread is responsible for reading from the multicast buffer. The read data is passed on to the client's main buffer that is being controlled by the `main_program` thread. Whenever the `main_program` thread discovers a segment in its main buffer it sends it to the player module.

3.5.2 Client wait time

Client wait time is the time from when the client sends a request for a video, to the client is able to play the video. In the users best interest, the client wait time should be as close to zero as possible to give a best viewing experience. Our design should reflect this and try to minimize the client wait time as much as possible. In a perfect network environment where packet drop, jitter, burstiness and network congestion do not occur, we can get the client waiting time down to approximately two seconds. This is because the client always store a complete segment in its multicast buffer before sending it to the player module. However, the Internet is not a perfect network environment and we have to take some precautions, which affects the client wait time. The reason we chose to store the whole segment and not just forward a single network packet is not just because of an easy buffer implementation. It is also to camouflage jitter, burstiness and packets loss. If the client was in a very unstable network, we would have to consider to increase the number of segments the client buffers before the segments are forwarded to the player module. This, of course, increase the client wait time.

3.5.3 Quality adaptation

As mentioned in section 3.1, the video can be played back in four different qualities. The decision of which quality the download module should download is up to the player module and the `read_from_buffer` thread. The player module sets an upper quality limit based on client system performance. For instance, a handheld device generally has lower system performance than a stationary computer. This limit is sent to the download module, and it is up to the download module to fulfill the player module's request. The quality can also be limited by the client's bandwidth. In this case, it is up to the download module to set the quality limitation. When no patching is initiated, the approach for selecting the proper download quality is straightforward: Start to listen to the multicast stream providing the minimum quality and if the first segment is received within playtime, increase the quality by one until the maximum quality limit is reached. The maximum quality limit is decided by either the player module if it is limited based on system performance or the download module if it is limited by bandwidth.

The client can potentially change quality every two seconds, because each segment is two seconds long. The client starts off by receiving the lowest quality. When it is time for a segment to play, the client checks if the segment is completely stored in its buffer. If this is the case, the client increases quality for each segment arrived in time for playback, as there are no indications that the client is not able to receive a higher quality. The client drops to a lower quality whenever a deadline miss occurs, that is when a segment is not stored in the buffer in time for playback. This means that the available network resources is not good enough to transmit a segment in the current quality, within the playback deadline for a segment. After receiving 20 segments within time of playback, the client increases quality again. As we have stated, the quality can be changed every two seconds. The reason the client waits 20 segments before increasing quality again, is because we find it unlikely that the current condition of the network dramatically improves over two seconds. The 20 seconds wait time applies to the rest of the stream and only happens if the client at some point in the stream were limited by network resources.

When the client receives a patch stream and a multicast stream simultaneously, one option is to prioritize either the patching stream or the multicast stream, by keeping the current quality on one stream and minimizing the quality on the other stream, if the the client discovers that it needs a lower quality. If the client again asks for a lower quality, the quality is lowered on both streams. It is reasonable to minimize the stream that provides the shortest part of the whole video. If the client misses the start of a live multicast stream, it is difficult to know if the patch stream is shorter than the remaining multicast stream because it is unknown when the stream ends.

Another problem with prioritizing the quality on either the patch stream or the multicast

stream is if the player module requests a lower quality based on its own system performance instead of bandwidth limitation. Then, the client has to keep the high quality on one stream for no reason. Instead we went for another option if patching is initiated and the client has to listen to two streams simultaneously. Both streams starts downloading in the lowest quality and then gradually increase quality unless the client notice any problems. If the client needs a lower quality, both the multicast stream and the patch stream is downloaded in a lower quality. We believe this approach is most fair to both the patch stream and the multicast stream, as we think it gives the client a best viewing experience when receiving two streams in medium quality instead of one stream in low quality and one in high quality.

3.5.4 Scheduling decisions

To be able to receive the segments either from a multicast stream of from a patch server in time for playback, there are several scheduling decisions the client has to make. As mentioned in 3.3.4, the transmission time of a segment from the multicast server is 2 seconds, regardless of quality. All scheduling decisions the client makes are based on this transmission time. One scheduling decision that must be made, is when the client should check to see if a segment from the multicast server has been stored in its buffer. This check is done so the client can discover if it needs a patch for that segment or not. This check happens every 2 seconds. If the client checks the status of a segment less often than every 2 seconds, the check would not synchronize with the playback speed of the video, which could cause that the segment is not stored in the buffer at the time it is to be played. If the check is done more often than every 2 seconds, the client would not be able to receive the whole segment from the multicast stream before it is checked. However, a segment from the multicast server can be given more than two seconds to arrive, to compensate for jitter, by delaying the check for the first segment, as seen in figure 3.5. When jitter occurs in the network, the end-to-end delay from the multicast server to the client varies, meaning that it at times could take a segment more than 2 seconds to arrive. The figure 3.5 shows that the check for segment 1 is done 1 second after the expected arrival time of segment 1. This means that each segment has been given a window of 3 seconds to be stored in the buffer. However, a new check is done every 2 seconds, so it is synchronized with the playback speed and arrival rate of the segments.

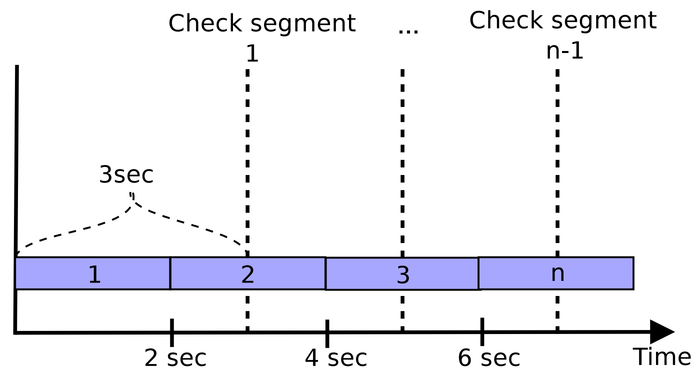


Figure 3.5: Check if multicast segments are stored in the buffer. The check is done 1 second after the expected arrival time of a segment.

If a segment has not been completely stored in the buffer at the time it is checked, the client assumes that either some parts of the segment are lost in the network, or the segment is not received at all from the multicast server. In both cases the client requests a patch for the missing segments, or the missing parts of it, from the patch server. If the client has not received any parts of the segment from the multicast server, it means that the client arrived late to the multicast stream and did not receive the first part of the video. If the client has only received parts of a segment from the multicast server, the transmission of the segment was exposed to some problems in the network, like packet loss.

To make sure that the client has enough time to download a patch for the segment, before it is to be played in the play module, it has to make another scheduling decision. The client must decide for how long it should wait before starting to play the segments, i.e., how much it should buffer. Figure 3.6 illustrate the scheduling of one segment. The segment is given 2 seconds to arrive from the multicast stream. Then the client checks if the segment has been stored in the buffer, and discovers that it is not completely stored. The client then sends a request for a patch, which is given 1 second to arrive. The download time for a segment from the patch server is limited by the current network conditions, like available bandwidth, delay and packet loss. This means that it can take, for instance, 1 second or 3 seconds to download a segment. If the patch for a segment is not downloaded within time for playback, the client experiences a deadline miss and the viewer sees a black screen for two seconds. As mentioned in 3.5.3, the deadline miss causes a drop in quality. The client plays the segment after three seconds.

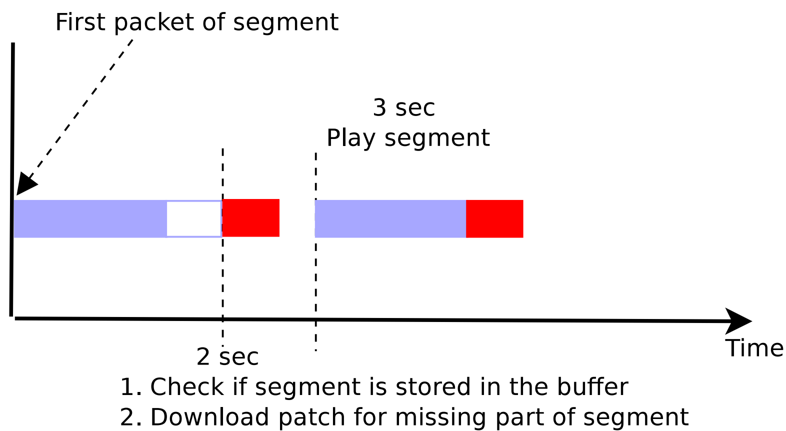


Figure 3.6: Scheduling of one segment. Client buffers three seconds and the patch is given one second to download.

If a client did not get the first 3 segments of the video from the multicast stream, it has to download these segments from the patch server. Even though the client misses 3 segments in a row, it does not download these as fast as possible. A new request for a segment is done every 2 seconds, as the client does a check to see if the segment is stored in its buffer every 2 seconds. The reason the client downloads the patch segments at the same rate as the segments are played, is because we want to minimize client buffering. Segment 6, which is received from the multicast stream, is checked if it is in the client buffer 6 seconds after it arrived, as seen in figure 3.7. This segment misses some data and the client requests a patch for the missing parts of the segment. The reason the client does not request a patch for segment 6 earlier is that it causes additional competition for the available bandwidth if the patch for segment 6 is downloaded at the same time as the patch stream and the multicast stream. The request for segment 1 is sent to the patch server after 2 seconds. This is because the client expected the segment to be received from the multicast stream and it takes two seconds before the client discovers that segment one is missing. The figure also shows that the multicast stream is received simultaneously with the patch stream. The segments from the patch server have a shorter transmission time than segments from the multicast stream. This is, as mentioned, because the download time for a segment from the patch server is only limited by the current network conditions.

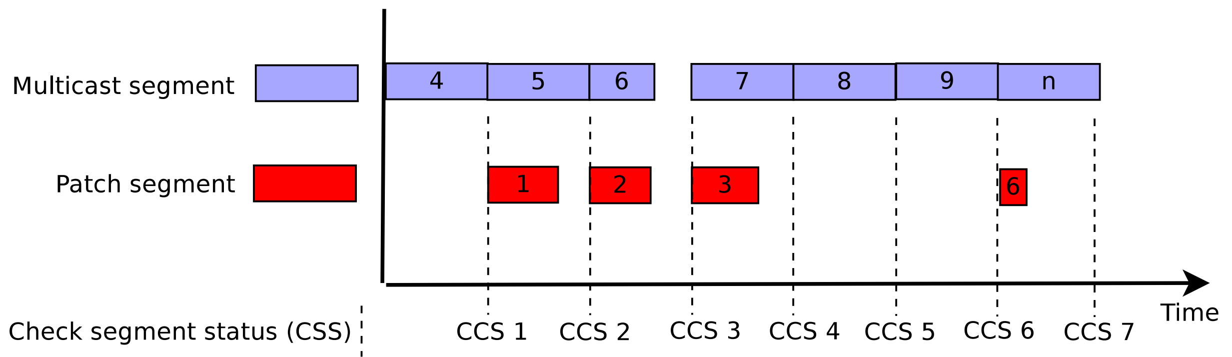


Figure 3.7: Scheduling of patch request

3.5.5 Client buffer management

The download module in our client has two buffers. The multicast buffer and the main buffer, as seen in figure 3.4. How a segment is passed between the two buffers and to the player module are illustrated in figure 3.8.

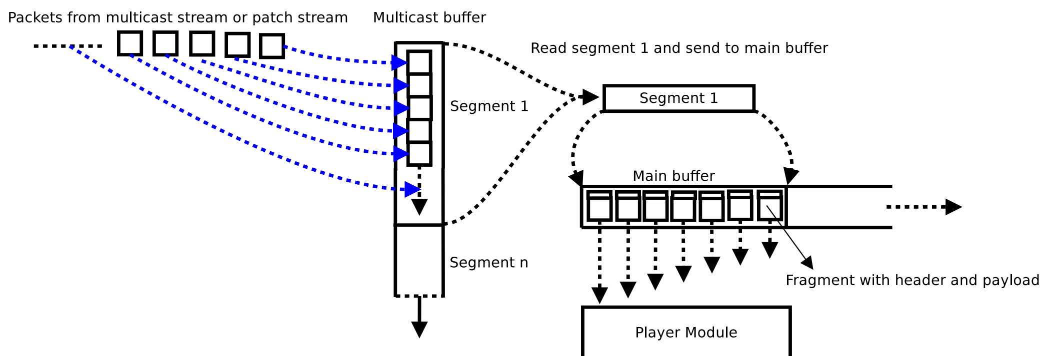


Figure 3.8: Segment flow between buffers

When receiving a multicast stream or patch stream, the packets belonging to the same segment are stored in the same part of the multicast buffer. When the `read_from_buffer` thread discovers a stored segment in the multicast buffer it reads the segment, pass it to the main buffer and remove it from the multicast buffer. The `read_from_buffer` thread tries to read a new segment every two seconds to maintain the playout speed at the player module. If a segment has not arrived at the time when it is supposed to be read from the multicast buffer then the `read_from_buffer` thread skips that segment and tries to read the next segment after two seconds. The segment that missed its deadline for playback

is dropped. The maximum number of stored segments in the buffer never exceeds the number of patched segments plus the delay set before playback. To decide where to put the incoming segment in the multicast buffer, we use segment numbering. The multicast buffer is indexed from 0 to the maximum segment number. Segment *m* is stored at index *m* in the multicast buffer. By using a byte offset specified in the packet header, the client can store the packet in the correct place within the segment. The approach of using segment numbering makes it easy for the `read_from_buffer` thread to locate the correct segment in the buffer, as it can just specify the segment number to read. As an alternative to storing whole segments in the buffer, the client could have stored the incoming packets separately and passed a packet to the main buffer as soon as it is received. Why we have chosen to store whole segments in the multicast buffer is explained in section 3.5.2.

The other buffer in the download module is the main buffer. This buffer is not designed by us, but is a part of the DAVVI implementation which we have build our code upon. The main buffer is managed by the `main_program` thread, and its responsibility is to store segments that are to be forwarded to the player module. The segments in the main buffer are not stored in the same way as in the multicast buffer. We copy a whole segment into the main buffer, where the segment is split up into fragments before being sent to the player module. Each fragment consists of a fragment header and the fragment payload. The fragment header informs the player module about which segment it received and how large it is. The fragments in the buffer is managed by the first in, first out (FIFO) algorithm. How this buffer is used to communicate with the player module, is further explained in section 3.5.9.

3.5.6 Receiving multicast video stream

To be able to receive a multicast stream, the client must first contact the multicast server, and a new video stream is multicasted if it has not already been started. We could have used the RTSP protocol for the video request but we chose to implement our own solution. The reason for that is that RTSP has a much higher complexity in its connection setup than we need. What we need is a very basic approach where the client contacts the server telling it that it wants to receive a video. This is done by sending a request for a video from the client to the server over TCP.

When receiving a multicast stream, the client must find out if it needs a patch stream. If the first segment it receives is not the first segment of the video then it is in a need for a patch. The `get_patch` thread start downloading the missing parts of the video when it discovers that something is missing and at the same time the `multicast_receiver` thread puts any incoming multicast stream packets into the multicast buffer. When receiving packets from the multicast stream, the client has to put them

together into a whole complete two-second segment before sending it to the player module. The reason is that packets can be received out of order. By waiting for a segment to arrive completely before sending it to the player module, the client can store the packets belonging to the same segment in correct order. The client can store the packets in correct order with the use of an application header. This application header is added by the multicast server to each packet sent over the multicast channel. The information in this header can be used to determine when a segment starts and when it ends and which part of the segment the client just received. How the client decide where to put the packet in the buffer is explained in section 3.5.5. The `read_from_buffer` thread later reads segments from the multicast buffer and forward them to the main buffer in the `main_program` thread, which sends them to the player module, as illustrated in figure 3.4.

When the client is receiving a multicast stream, it never listens to more than one channel at the same time. This can cause some trouble when the client wants to go from receiving one quality to another quality. In this transition phase the client leaves one multicast stream channel and joins a new one. In this phase, the client might experience some packet loss because packets is still being transmitted from the multicast server while the client change channel. If we wanted to minimize the risk for packet loss in this phase the client could have chosen to listen to two multicast streams at the same time, but this could have a negative effect if the client does not have a lot of bandwidth available. Especially, if the client was to downloading a patch stream at the same time.

The `multicast_receiver` thread must have support for dealing with jitter and packets coming out of sequence. We want a packet coming out of sequence to be included in the buffer and not detect packet loss if the client has enough time to store it in the buffer before consumption by player module.

3.5.7 Downloading patch video stream

The `get_patch` thread starts downloading segments from the patch server if the client arrive late for the multicast stream. The `get_patch` thread discovers which part of the video the client is missing and start downloading the missing segments from the patch servers. The difference in downloading a patch due to a late arrival and due to packet loss is that in the first case the client has to download whole segments from the patch servers instead of just small parts. The patch are downloaded by using HTTP GET requests over TCP. One request per segment. Since TCP offers reliable connection, the client does not need to worry about packet loss. Jitter or burstiness is not a problem either, because downloading from the patch server is a bit different that receiving a multicast stream. When receiving a multicast stream it takes two seconds to receive a two seconds long segment. When requesting a segment from the patch server, the download itself is done

as quickly as possible. There are no time limitations to reduce the send out rate from the patch server. To be able to minimize client buffering, the download module itself has to control the download speed of the patch. As mentioned in 3.5.4, this is done by using timers that only sends a request for a new segment every two seconds i.e., since this is the length of each segment. The patch function writes the received segments to the correct location in the multicast buffer. The `read_from_buffer` thread then reds the downloaded segment and forwards it to the main buffer.

In the other case, the `get_patch` thread downloads a patch, is when a packet loss from the multicast stream is detected. As with all transmissions over UDP, packet loss can occur. There are several ways to deal with this. One option is to write values of zero to the multicast buffer equal the size of the packets being dropped. This solution is very efficient performance wise but the image looks bad when played in the player module if we have a high rate of packet loss. If only one packet were dropped we might not have noticed it as a viewer. We went for another approach. Whenever the `get_patch` thread discover a packet loss, it retransmits that packet from the patch server. With HTTP GET requests, the `get_patch` thread has the possibility to download just the part the client missed by using the RANGE option, as illustrated in figure 3.3. This solution is of course more efficient, with regards to client bandwidth, than retransmitting the whole segment from the patch server when it only needs a small part of it. We want to do repairs on a segment as late as possible. As mentioned in 3.5.4, segments is checked for errors every two seconds, close to playback time. The reason for that is that we want to limit the number of requests send to the patch server at the same time. If the client is requesting a repair for a segment at the same time it downloads another patched segment, it increases the competition for the available bandwidth. An issue is that there might be limited amount of time to retransmit the missing parts, so in our experiments, we perform tests to find out how packet loss, round trip times and available bandwidth affects the download times.

We believe that using HTTP GET request over TCP works quite well, because the potential download speed of the missing parts of the video is much higher than the playback rate of the video. As stated in the article [37], a large TCP throughput relative to the video playback rate is needed to cope with large round-trip times, high loss rates and timeout values.

3.5.8 Thread synchronization

In this thread we describe how the threads are synchronized in the context of the scheduling decisions presented in section 3.5.4. It is very important that the `multicast_receiver` thread, the `get_patch` thread and the `read_from_buffer` thread are synchronized. If these threads were to operate on its own without any limits it could for example cause

that the read_from_buffer thread would try to read from the multicast buffer faster than the playout speed of two seconds. To synchronize the threads the client use timers. At the time of when the client first receives a multicast packet, a global timer starts. This timer is used to control the get_patch thread and the read_from_buffer thread. In addition, both these threads have local timers used to schedule tasks within the threads by using them in conjunction with the global timer. Both of these threads are to perform a task every two seconds. For the get_patch thread, it must check if a segment is complete or missing in total. Then, it must order a patch for that segment, if it is not completely arrived. The read_from_buffer thread is to read a segment from the multicast buffer every two seconds. With the timers the client can control when these threads should start to operate. If the client wants to buffer up two segments before playing them, it could make the read_from_buffer thread wait four seconds before it start to read from the multicast buffer. In the same way, the client can use the timers to decide when a segment must be stored in the multicast buffer before it is considered lost.

3.5.9 Download module and player module communication

The design and implementation of how the download module communicates with the player module is not ours. It is premade by the authors behind the article [20], but we have utilized these functions to be able to get our implementation to work properly with functions like quality adaptation and video requests to the multicast server. As mentioned in section 3.4, our client consists of two different components. The download module which is responsible for downloading the video from the servers and present it to the player module. The player module is responsible for applying video headers to the video and play it. These two programs have to communicate in some way and this is done using a local TCP socket. Communication between the player module and the download module is two way communication. The player module sends request to the download module, which fulfills the request and send data back to the player module. There are two different request that are to be sent from the player module to the download module. One being a request for a video and the other being a request for quality adaptation. Originally, the player module supported trick play functionality like pause and seek, but this is not supported by our implementation. The video itself is sent to the player module, whenever the main_program thread discovers something in the main buffer, by putting the data on the local TCP write socket. The player module then pick the packet up and play it.

3.6 Summary

In this chapter, we have presented the most important design features of our multicast server and client. A short presentation of the patch server system has also been given. In the next chapter, we move on to see how our design features are realized, with the presentation of our implementation.

Chapter 4

Implementation

In this chapter, we present the implementation details of our system, i.e., the multicast server and the client. The implementation details of the patch server is not presented, as this is implemented as a normal web server. Instead we present how the client interact with the patch server. We describe more deeply how the challenges and issues presented in chapter 3 are solved. First, we begin with the implementation details of our multicast server.

4.1 Multicast server implementation

In this section we present the implementation of our multicast server. We describe the Live555 Streaming media framework [24], the components our server consists of, and how we use them to solve the different issues we faced.

4.1.1 The Live555 framework

For the multicast server implementation we have chosen to use the Live555 Streaming Media framework [24]. This is an open source framework that consists of a set of C++ libraries that is suitable for embedded and/or low cost multimedia streaming applications. The libraries support many open standard protocols like RTP/RTCP, RTSP and SIP as well as streaming and receiving several popular audio and video codecs like MPEG, H263+, DV, MP3 and WAW. The design of the framework makes it easy to extend it with support for more formats. We chose to use this to implement our multicast server because it is a framework we are familiar with, and it is easy to extend it with support for the video codec we are going to use. In addition to that, it provides us with useful functionality for scheduling our video streams and sending data out on multicast channels. All information regarding the Live555 program flow and library description is taken from the Live555 website [24] and the master theses of Tonje Jystad Fredrikson[12] and Tommy Gudmundsen [15].

4.1.1.1 Live555 typical program flow

In this section we present the program flow and key concepts of this framework. This information is important to understand the more detailed description of our server implementation, that is to follow.

A typical server application using the Live555 framework use RTSP to handle incoming requests from clients. These requests are received by the server by listening on a network read handler that is implemented with a socket. The server uses an event loop `TaskScheduler::doEventLoop()` to handle the incoming requests and to manage response tasks related to the requests, for example triggering transmission of data. All tasks are put in a delay queue. This queue specifies when a task is to be executed. There needs to be active network read handlers or waiting tasks in the delay queue for the event loop to be active. A `select()` loop is used to monitor the network read handlers. The event loop works basically as follows:

```
while (1) {
        find a task that needs to be done
        (by looking on the delay queue,
                and the list of network read handlers);
        perform this task;
}
```

There are three conceptual component types in the Live555 system that we have to be aware of in order to describe how the data are passed through the server system.

Source: A source of media data. This may be a file source or a network source or an internal component in the framework.

Sink: A sink is a destination for the media data. On the server the sinks are responsible for transmitting data over the network.

Filter: Sources that receive data from other sources are called filters. When a filter is interacting with a sink, it takes on the role as a source. A framer, which is a component that provides the sink with video frames, is a typical example of a filter.

The data in an application passes through a chain of sources and sinks - e.g.,

'source1' -> 'source2' (a filter) -> 'source3' (a filter) -> 'sink'

The application starts by calling `someSinkObject->startPlaying()`, to start generating tasks that need to be done, before entering the event loop. The event loop then handles the generated tasks, put in the delayed queue. The data moves from a source and ends up in a sink which is responsible for transmitting the data out on the network. The sink requests a video frame from the framer by calling `FramedSource::getNextFrame()` on a source that is its direct neighbor. This is implemented by a pure virtual function `FramedSource::doGetNextFrame()`. This function is implemented by each 'source' module. The `FramedSource::doGetNextFrame()` implementation in each module is triggered by a call back function in the event loop whenever new data becomes available to the caller.

4.1.1.2 Live555 library description

In this section, we list a description of the main components of the Live555 library and their functionality. This list is presented in this thesis as it is described on the Live555 website [24].

The code includes the following libraries, each with its own subdirectory:

UsageEnvironment The `UsageEnvironment` and `TaskScheduler` classes are used for scheduling deferred events, for assigning handlers for asynchronous read events, and for outputting error/warning messages. Also, the `HashTable` class defines the interface to a generic hash table, used by the rest of the code. These are all abstract base classes; they must be subclassed for use in an implementation. These subclasses can exploit the particular properties of the environment in which the program will run - e.g., its GUI and/or scripting environment.

groupsock The classes in this library encapsulate network interfaces and sockets. In particular, the `Groupsock` class encapsulates a socket for sending (and/or receiveing) multicast datagrams.

liveMedia This library defines a class hierarchy - rooted in the `Medium` class - for a variety of streaming media types and codecs.

BasicUsageEnvironment This library defines one concrete implementation (i.e., subclasses) of the `UsageEnvironment` classes, for use in simple, console applications. Read events and delayed operations are handled using a `select()` loop.

4.1.2 Server components

In this section, we describe how our multicast server is implemented with the use of the Live555 framework. We provide a description of our most important server components, and their functionality. The use of these components and how they interact with each other is explained. We start with the presentation of a list of our main components.

streamServer The `streamServer` component is the main component of our application. This component is responsible for initiating all the other components. In addition to this, it also manages client requests, generates multicast IP addresses for multicast channels and to obtain information about the requested video, like finding out how many segments the video consist of and their location.

davviByteStreamMultiFileSource The `davviByteStreamMultiFileSource` component is responsible for using the information regarding the segments, provided by the streamServer, to feed one segment at a time to the `davviByteStreamFileSource` component.

davviByteStreamFileSource The `davviByteStreamFileSource` is responsible for reading a segment, provided by the `davviByteStreamMultiFileSource`, as a byte stream. It then splits the segment up into smaller frames. Each frame is given a

timestamp, indicating when it is to be sent on the multicast channel, before it is put in an internal buffer in the framework for further treatment. This class utilizes functions in the framer class `FramedSource` to fulfill its tasks.

simpleRTPSink The `simpleRTPSink` is a part of the multicast channel and is responsible for encapsulating the frames into RTP packets. It also schedules transmission of the frames by using the time stamp determined in the `davviByteStreamFileSource`. This class is not implemented by us, but is a part of the Live555 library.

4.1.3 Establishing client sessions

The server regard a client session as the time from when a client request is received, to the time of when the server has finished transmitting a video. A client session is established in the `main()` function in the streamServer application. As argued in section 4.2.3 we have chosen to use TCP to handle incoming requests from clients. By using TCP, we do not have to worry about the loss of a request packet in the network. The server listens for request from clients by using a while loop with a `select()` call. In this way, the server is able to handle requests from multiple clients. Whenever a client sends a request to the server, the server finds out if the requested video is already playing. If there is no ongoing transmission of the video, then the server starts a new transmission. If it is already playing, the server does nothing and the client has to get a patch stream for the parts of the video that it misses. Whenever the server has to start a new transmission of a video, a new thread is generated using the boost thread libraries [6]. By generating a new thread for the transmission of a video, the server can transmit the video and at the same time continue to listen for new client requests simultaneously.

4.1.4 Channel creation

The process of creating a channel and the interaction between the different components works basically as follows:

```
createChannels() {
 1. Allocate multicast addresses for each channel and a rtp port number
 2. Create UseageEnvironment:
       UsageEnvironment* env;
 3. Create TaskScheduler:
       TaskScheduler* scheduler = BasicTaskScheduler::createNew();
       env = BasicUsageEnvironment::createNew(*scheduler);
 4. For each channel. Set up channel environment.
    4.1 Create GroupSock:
       Groupsock rtpGroupsock(*env, multicastAddress, rtpPort, ...);
    4.2 Create RTP video sink:
       RTPSink* videoSink = SimpleRTPSink::createNew(*env, &rtpGroupsock, ...);
    4.3 Create video source:
       FramedSource* videoSource = DavviByteStreamMultiFileSource::createNew(*env, file_list, ...);
    4.5 Activate video sink object:
       videoSink->startPlaying(*videoSource, videoSink, ...);
 5. Start the event loop to start all channels:
       env->taskScheduler().doEventLoop();
}
```

Whenever the server finds out that it has to start transmitting a new video, it generates four new multicast channels for that video, one for each quality. All channels use the same `UsageEnvironment` and `TaskScheduler` objects. It is up to the `TaskScheduler` to synchronize the channels. We could have used a single thread for each channel but chose to use the concept of event driven programming which is recommended by the authors of the Live555 framework.

The first thing the server does when creating a channel is to allocate a multicast address and a RTP port number for that channel. In our implementation, this is hard coded by just giving a channel a static address. Ideally, the server should have dynamic addresses to make sure it does not chose an address that is already occupied, but in our test environment, this is never a problem for our proof-of-concept prototype. Our implementation is only intended for testing purposes and is not usable as a commercial product. A `GroupSock` object is generated for the channel to send multicast datagrams.

The server also needs a sink so that it is able to transmit the video. Each channel generates its own `SimpleRTPSink` object for this purpose. It is the sink that is responsible for sending a frame out on the socket object `GroupSock`.

The server also has to create a video source for its channel. This is done by generating an instance of a `DavviByteStreamMultiFileSource` object in the `FramedSource` class.

The `DavviByteStreamMultiFileSource` object needs to know which segments to read. A list of paths to all the appropriate segments is generated for each channel and passed to the `DavviByteStreamMultiFileSource` object. All paths in this list leads to segments that are in the same quality. The `DavviByteStreamMultiFileSource` object iterates through the list of paths, and for each path creates a `DavviByteStreamFileSource` object. A new `DavviByteStreamFileSource` object is initiated by the `DavviByteStreamMultiFileSource` every time a new segment is to be transmitted on a channel. When a segment has been transmitted, the `DavviByteStreamMultiFileSource:doGetNextFrame()` is called and a new instance of a `DavviByteStreamFileSource` object is initiated to process the next segment in the list. This happens every two seconds for each quality. Instances of the `DavviByteStreamFileSource` class represent the video source in our server application.

Now that everything is set up for the channels, the server can initiate the process of sending a segment by activating its video sink object function `videoSink->startPlaying` which takes the video source as a parameter, so it knows from where to get data.

Finally, the server has to provide the channels with a common event loop so the data is passed from each source to the sinks. This is done by calling the function `taskScheduler().doEventLoop()` which starts the event loop in the `BasicTaskScheduler` object.

4.1.5 Scheduling and transmitting data

The scheduling and transmission of data is not straight forward as some precautions have to be taken. To minimize buffering on client side, we have to consider how fast a segment should be transmitted. As argued in section 3.3.4, we want the transmission time for a two seconds long segment to be approximately two seconds. In this section, we explain how our system schedules and transmits the data packet to fulfill this requirement.

Our segment is transmitted by using the RTP protocol over UDP. Obviously, we have to split each segment into smaller pieces, as the network can not send a whole two-second segment in one packet. The `DavviByteStreamFileSource` class is responsible for reading a segment from file and which segment to read is specified by the `DavviByteStreamMultiFileSource` class. When reading a segment from file, the `DavviByteStreamFileSource` object reads small portions at a time, dividing the segment into smaller frames. These frames are not images, but network frames. Our RTPsink never transmits more than one frame per network packet so we want each frame to be as large as the maximum size of a network packet. The MTU of Ethernet is 1500 bytes so the

size of one frame must not exceed that limit. There are smaller MTU values, but the MTU value in our test network is not less than 1500 bytes. If the size of a frame exceeds 1500 bytes, then the frame is fragmented at IP level, which causes confusion on the client side due to the fact that the server has a custom application header that it adds to each frame sent over the network. The client would then receive some frames without this custom header. The use of our custom application header is explained in section 4.2. The frame with video data and a 16 byte custom application header added, is then encapsulated in a RTP packet with a 12 bytes RTP header. On top of that, there are headers like the UDP header and Ethernet header that also require some bytes. The Live555 framework has defined the upper limit of a frame size to be 1448 bytes. Within this size there is room for the RTP header, but our 16 byte custom application header is not taken into consideration. The `DavviByteStreamFileSource` object therefore only read maximum 1432 bytes at a time from the video segment file, giving 1432 bytes network frames. The bytes read from file are put in a buffer `fTo`, defined by the framer class `FramedSource`. This buffer is accessible by almost every part of the library and is used to pass data along from source to sink.

Now that the server has split the video segment file into smaller network frames, it have to schedule the network frames. As stated earlier, the entire segment should be sent on the channel over a two seconds long period of time. To avoid burstiness on the channel, each network frame is given a time for which it is to be sent out of the channel. How long each network frame should be delayed is given by this formula:

```
//Set fPlayTimePerFrame so the entire file takes 2 seconds to transfer
//2 seconds = 2 000 000 microseconds
int fs = GetFileSize(fileName, fFid);
float playTime = 2000000 / (fs / frameSize);
fPlayTimePerFrame = (int) playTime;
```

This formula is calculated in the `DavviByteStreamFileSource::doReadFromFile()` function. The `DavviByteStreamFileSource` class divides a segment into network frames. This formula gives us the time for how long a network frame should be delayed before it is sent out on the channel, with regard to when the previous network frame was sent. By dividing two seconds by segment file size, which again is divided by network frame size we get the time the server must delay each network frame. For example, a segment of 1MB is divided into 732 network frames. By that we get that the server must have a sending interval of 2732 microseconds, i.e., each network frame is delayed 2732 microseconds. When implementing this, we had some problems on the client side. The client was experiencing small hiccups between two segments. We discovered that the transmission time of one segment was slightly above 2 seconds. The problem was solved

by setting the transfer time to 1973000 instead of 2000000 microseconds. The time of 1973000 microseconds is based on 2000000 microseconds minus the processing delay of our application. With a time of 1973000 microseconds, we measured an average of approximately two seconds transfer time for a segment. With this formula, the sending bit rate of a channel is dynamic, as it takes two seconds to transmit a segment regardless of the quality of the segment.

Our server is run on a computer with only one CPU, making the instructions for all channels to execute in sequence. To camouflage this, the `TaskScheduler` in the framework takes care of the synchronization among the channels so that four channels send out their segments at approximately the same time based on the `playTime` that is set for each network frame.

4.2 Client side implementation

For the client application, we have extended the functionality of the DAVVI client. Originally, the DAVVI client downloads segments from a web server and plays them in its player module. Our client use this functionality to download a patch stream whenever our client is arriving late to the multicast stream, and it is used it to download missing parts of segments received from the multicast stream. The segments can be considered lost if the client experience packet loss or if the segment are not received at all. To be able to receive a multicast stream, we have to extend the functionality of the DAVVI client. We do not implement any changes to the player module in the client. All of our implementation is within the download module of the client application. We begin by listing the different functions in our implementation to get an overview of how they work. Besides these functions there are functions already provided that are responsible for communication between the player module and the download module and by getting information from the tracker, as explained in chapter 3.5.

4.2.1 Client functions

In this section we present the different main functions that we use in the implementation of our client.

connect_to_streamserver This function sends a TCP video request to the multicast server. It returns immediately after sending the request. The multicast server reply is the multicast stream itself.

join_multicast This function is used to join the multicast stream. It sends an IGMP join request to the appropriate multicast address and then return a socket that our application can listen on if the join request succeeded.

leave_multicast This function sends a DROP MEMBERSHIP message to leave the multicast stream.

multicast_receiver This function runs inside the `read_from_buffer` thread and listens to a multicast stream. It is responsible for preparing the client by calling the above functions to request a video from the multicast server as well as joining, leaving and receiving the actual multicast stream. It also start up the `get_patch` thread and the `read_from_buffer` thread.

read_from_buffer This function runs inside the `read_from_buffer` thread and is responsible for reading from the multicast buffer and send segments to the main buffer, as seen in figure 3.4. It tries to read a new segment every two seconds. If a segment is not received completely at the time it is to be read by the thread, then it is dropped.

get_patch This function runs inside the `get_patch` thread when patching is needed. It is responsible for initiating downloading of the missing segments with HTTP GET requests from a web server or HTTP GET RANGE request, if the client needs to download only a smaller part of the segment. It has been implemented by using the cURL framework [9]. The downloaded content is put into the `multicast_buffer` buffer as explained in section 3.5.7.

4.2.2 Client multicast buffer

To be able to handle an incoming multicast video stream the client must be prepared for the content it is about to receive. First of all, it needs a buffer to store the received segments in. We call this buffer the `multicast_buffer`. This is an array of char pointers where the indices in the array map directly to a segment number. For example, packets belonging to segment number ten is stored in index ten in the array. In section 4.2.3, we see how the client can store a packet at an exact location within an array index based on byte offset.

In addition to this buffer, the client has another array used to keep information about the content in the multicast_buffer. We call this the `multicast_buffer_info` array and it is a two dimensional array. This is indexed in the same way as the multicast_buffer but for each index the client can store six different integer values. Each integer value provides the

client with information about the content in the `multicast_buffer`. The integers used are as follows:

1. **Downloaded size of segment:** The first integer represents the downloaded size of a segment. This value is increased each time a packet belonging to that segment is received.

2. **Total size of segment:** The second integer represents the total size of a segment. This integer and the downloaded size of segment integer are used to detect packet loss. If the downloaded size of a segment does not reach the total size of a segment, then the client is missing some packets. The client get the total size of a segment from an application header contained withing each packet. More about this application header in section 4.2.3.

3. **Segment quality:** The third integer represents the quality of a segment. This is useful information when the client needs to have lost parts of a segment retransmitted. When retransmitting lost parts of segment the client downloads a patch from the patch server in the same quality that the segment originally was received from the multicast stream. In this way the data does not get corrupted, which could happen if data from two different qualities were to be combined in one segment.

4. **Discovered status:** The fourth integer is used to determine if the client has seen this segment before. The first time the client receives a packet belonging to a given segment, it has to allocate space in memory for that segment. When memory allocation for a segment is done the client gives this integer a value so it does not get allocated again when another packet belonging to the same segment is received.

5. **Complete status:** The fifth integer is used to determine if the client received a whole segment. Initially all segment are marked as incomplete. When downloaded size of a segment is equal to the total size of a segment then this integer is marked as complete. If a segment is incomplete at the time when the `get_patch` thread checks a segment, then it need to be retransmitted.

6. **Blacklist status:** The sixth integer is used to avoid that both the `multicast_receiver` thread and the `get_patch` thread write to the same segment in the `multicast_buffer` at the same time. This could happen if a packet from the multicast stream arrived after the deadline for which a segment must be stored complete in the `multicast_buffer`. If the client starts downloading a patch for a segment, this segment is blacklisted so that the `multicast_receiver` thread does not store any more data to that segment.

Finally, we present the third array. This is a bitmap and is used to keep track of packets belonging to a given segment that has arrived. Whenever the client receives a packet,

the packet is marked in the bitmap. When the client downloads a patch for a segment, it consults the bitmap to find out exactly which parts of the segment that is missing. In this way, the client does not have to download the whole segment when only a small part is missing.

4.2.3 Receiving multicast video stream

The `multicast_receiver` thread has the responsibility of receiving a multicast stream as well as requesting the stream from the multicast server. Before the client starts receiving a multicast stream, it has to send TCP video request for a video to the multicast stream server. This starts a new transmission from the multicast server if a transmission is not already in progress. The client then sends an IGMP join request to start listening to the multicast stream. The multicast addresses the client listens to are static addresses. Since our implementation is for testing purposes only, we have hard coded the addresses into the client. We are transmitting one video in 4 quality levels in our examples so only four multicast addresses is necessary. In a commercial product, a better solution would be to have the server send information to the client about which channels to listen to. When the routers have registered the client's interest in the multicast stream, the client starts to receive packets. These packets contain an application header in addition to the video payload. This is a 16 byte header which provides information about the packet, see table 4.1.

0 ———— 3	4 ——— 7	8 ——— 11	12 ————- 15
Segment Number	Byte Offset	Network Frame Size	Total Segment Size

Table 4.1: Application header

When a packet is received the `multicast_receiver` thread checks which segment it belongs to. If this is the first time it has seen a part of this segment, it allocates space in the memory for that segment, as explained in section 4.2.2. In addition to memory allocation the `multicast_receiver` thread also updates the `multicast_buffer_info` array with the total size of the segment which is also contained in the application header. The `multicast_receiver` thread then stores the packet into its buffer by using the `Segment Number` and `Byte Offset` in the packets application header. The `Segment Number` is used to find out which index in the `multicast_buffer` to store the packet in, and the `Byte Offset` is used to store the packet within the index in the `multicast_buffer`. The `Network Frame Size` field tells the client the size of the video payload in the packet. This is used to update information in the `multicast_buffer_info` array and is incremented for each incoming packet that belongs to that segment. Hopefully, the total network frame size, of all incoming packets belonging to a segment, is equal to the total

size of the segment. At the time when these two numbers becomes equal the segment is marked as complete. If they never gets equal, the segment remains incomplete, and the client has to request a patch for whatever is missing. If the segment needs to be patched, it is detected in the get_patch thread.

The multicast_receiver thread constantly checks to see if it should change quality. It is the player module and the read_from_buffer thread that determines if the client should receive another quality. A change in quality is requested by the player module if the hardware resources of the system is not good enough to play the video. The read_from_buffer thread request a lower quality if it detects that a segment is not stored in the multicast_buffer when the segment is to be played. In this case the segment did not arrive in time it is because of network conditions like, limited available bandwidth. The read_from_buffer thread requests a higher bandwidth if a segment is stored in the buffer multicast_buffer when it is to be played. If the multicast_receiver thread is to change quality, it first leaves the multicast stream by calling the leave_multicast function and then join a new multicast stream by calling the join_multicast function. The change of quality could happen in the middle of the reception of a segment from the multicast stream. This means that the client might miss some parts of a segment, as the client does not listen to a multicast stream between leaving one stream and joining another. The client then has to download a patch for the parts of the segment it missed.

Our implementation for receiving a multicast video stream over UDP considers packets arriving out of sequence. The client does not base its receiving procedure on the sequence number of the packets, but on the packet's segment number and byte offset. As long as all packets of a segment arrive within the playback deadline, it does not matter which order they arrive in.

4.2.4 Get patch stream

It is the responsibility of the get_patch thread to check if the client need a patch for the video. The get_patch thread works basically as follows:

```
get_patch()
  1. Wait for start.
     The longer time segments are given to arrive the longer the wait time
  2. For every two seconds
     Check if segment are complete
     If segment is not complete
       If all parts of the segment missing
          Start new thread for downloading missing segment
       If some parts of the segment are missing
          Start new thread for each missing parts of the segment
     Move to next segment
```

The client sets a deadline for when a segment must be stored in the multicast_buffer. This deadline is minimum two seconds, as it takes the server two seconds to send a segment. After this waiting period, the get_patch thread does a check every two seconds to see if a segment has arrived complete. If it has not been received at the time of this check, there are two cases: Either the client has never seen the segment before, or it has only received parts of it.

The first case happens when the client arrives late to a multicast stream and needs a patch stream. The get_patch thread then starts a new thread to download a patch for the segment. The reason the get_patch thread starts a new thread is that it might be a large round-trip time between the client and the server. This could cause that it takes longer than two seconds from get_patch thread sends the request, to the segment arrives. If the next segment is also missing, the get_patch thread might have to send a request for the segment before the last segment has arrived. By doing it this way, the client would get a constant stream when many segments are missing regardless of the round-trip time, as a new request is sent every two seconds.

The second case happens when the client are missing only some portions of the segment. By consulting the bitmap, the get_patch thread can see which parts of the segment that are missing. For each part missing, the get_patch thread start a new thread where each thread is given the actual bytes to download. These bytes is used in the HTTP GET requests RANGE option. By using multiple threads for downloading the missing parts, the get_patch thread does not have to wait for each download to be complete before asking for the next part. The request for the missing pieces are sent consecutively. When more than one packet is lost, forming a long sequence of lost coherent data, the get_patch thread specifies the byte offset of the first lost packet and the end byte offset of the last packet in the sequence, and send one request for the whole sequence, instead of one request per lost packet. The threads know which bytes they have requested, which makes it easy to put the received parts in the correct place in the buffer, as the packets is received in the correct order, due to the qualities of TCP.

When downloading a completely new segment the get_patch thread uses the current quality setting. This is because we always want to give the client the best possible quality, based on the clients observation of receiving segments within the time limit. When downloading the missing parts of the segments the get_patch thread requests the patch in the same quality as the segment originally was received in. The reason for that is that by writing data from two different qualities into the same segment causes a corrupt segment.

4.2.5 Read from multicast buffer

The operation of reading from the multicast_buffer happens in the read_from_buffer thread. This thread works basically as follows:

```
read_from_buffer()
  1. Wait for start.
     The more segment we buffer the longer the wait time
  2. For every two seconds
     Check if segment are complete
     If segment is not complete
        Reduce quality if not minimum
        Drop segment.
     If segment are complete
        Copy segment into main buffer
        Increase quality if not maximum
     Move to next segment
```

In the read_from_buffer thread, the client can control how many segment it wants to have in its multicast_buffer before sending them to the player module by using timers. In a perfect network, it would be reasonable to send a segment to the player module as quickly as possible after it has been stored in the multicast_buffer. However, the Internet is not a perfect network so the client might have to buffer up a number of segments before trying to read them. By doing so the client camouflages network jitter and gives the transmission of patched segments time to be completed. As mentioned, the packets can be delivered out of sequence and a packet from a new segment can be received before the last segment is complete. If the client buffer up a number of segments, this also helps camouflaging this network effect by giving the segment some extra time to be received. How much the client should buffer depends on the amount of jitter the network gives and how much out of order the packets can arrive.

Every two seconds the read_from_buffer thread does a check if a segment is complete, i.e. the client can play the segment. Every time the read_from_buffer thread has read a segment successfully, it does a check to see if it should request an increase in quality. As mentioned in section 3.5.3, the player module provides us with a maximum quality limit. The read_from_buffer thread never increases the quality beyond this limit. The quality can also be changed without notification from the player module. The client starts receiving in lowest quality and then gradually increase the quality as segments are received. For every segment that arrive within playtime, the read_from_buffer thread increases the quality by listening on a multicast stream providing us with better quality video, as there are no indications of a limited bandwidth. An exception is if the read_from_buffer thread has previously reduced quality due to playtime deadline misses, which indicates that the client has a bandwidth problem. In this case, the read_from_buffer thread only increases the quality for every 20 segments that are received before the playtime deadline.

The reason for the interval of 20 segments is that the client tries to avoid to download a better quality than it is able to. It is not likely that the network conditions have dramatically improved within the length of one segment. After 20 segments the load on the network might be less, or the client might have stopped downloading a patch stream at the same time as it receives a multicast stream. If it turns out that the client is only receiving a multicast stream, then it should be able to increase the quality on that stream. If the client ever experience a playback deadline miss the `read_from_buffer` thread decreases the quality by one for each deadline miss, unless the client has reached lowest quality, and drop the segment. By decreasing the quality, the network load is reduced and the client might be able to receive segments within the time limit again. The segment that is dropped due to deadline miss is never played and the thread moves to the next segment in line.

4.3 Implementation challenges using curl

When we started to run tests, we discovered a problem with our implementation. This problem was related to requests for patched data when the client experienced packet loss. Even though the client makes one request for coherent lost packets, the client experience problems when many packets are lost and few of them are coherent. What happens is that the client is running out of resources when generating to many threads. The boost thread library gives an error message telling us that the client has run out of resources and then the program crashes. To solve this, the client has a limit of 300 threads at a time. As we see in one of our experiments, this is not a big problem unless the client experience extreme packet loss conditions. In these conditions the requests for lost data are delayed because the client must wait for the number of threads to be lower than 300 before a new thread can be generated. The reason the client is using threads is because the download function in the curl framework [9] does not return until the requested data has been downloaded and that slows down any new requests for data. A better solution would be to not use the Curl framework for downloading data and instead use the Linux function Wget and use a select loop to handle incoming data. By doing this, the client would not need to use threads at all to download patched data.

4.4 Summary

In this chapter, we have given a detailed description of how our implementation of the multicast server and client works according to the design in chapter 3. The implementation choices we have made are done with considerations on the challenges the Internet

gives us, like packet loss, jitter and packets arriving out of sequence. In the next chapter, we perform experiments on our implementation to find out how it works in different scenarios.

Chapter 5

Experiments

In this chapter we perform experiments on our implementation of the DAVVI system extended with the patching scheme in an Internet environment. The purpose of these experiments is to see how our implementation cope with the challenges that come along with the Internet, like packet loss, jitter, bandwidth limitations, long round trip times and reordering. We investigate how our application can adapt to these challenges by using client side buffering and quality adaptation.

5.1 Test environment

In this section we present our test environment. We have a look at how our testbed network is setup, the media files used in the experiment, and what kind of tools and techniques we have used to analyze our results.

5.1.1 Testbed

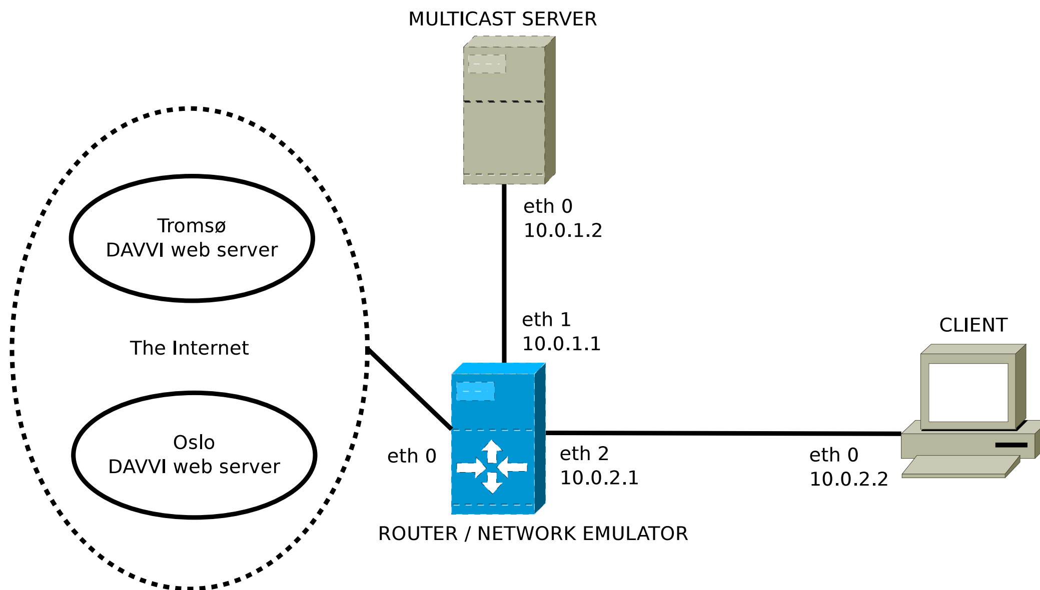


Figure 5.1: Overview of testbed

Due to the fact that multicast is not switched on in the Internet, we have to set up our own network to simulate an Internet environment. Our testbed network consists of three computers, as seen in figure 5.1. The multicast server is responsible for sending a multicast stream out on the network. The client is responsible for receiving a video stream either from the multicast server or the patch server system. The patch server system is web based and located in the Internet. To connect the multicast server, the client and the patch servers, we use a computer that works as a router and a network emulator. All network traffic between the Internet, multicast server and the client goes through this router. By running network emulations on this computer, by the use of netem [11, 21] and tc [11, 21], we can introduce packet loss, jitter, reordering, delays and bandwidth limitations to the network. It is also responsible for building a multicast tree to forward packets from the multicast server to the client. As seen in the figure 5.1, the network is divided

into separate IP subnets for the client and the server computers. The server and the client are connected to the router using different interfaces, eth1 and eth2 respectively. To set up multicast forwarding and multicast tree building, the router is running XORP [36], which is an open source routing platform. We have configured XORP with IGMPv2 for multicast membership service and PIM-SM [1] for multicast tree generation.

5.1.2 Media description

We use collections of two-second video segments to test our streaming system in different network scenarios. The collections, four in total, are a clip from a football match presented in four different qualities. Depending on the test we are running we choose an appropriate number of segments to play. Due to differences in qualities the segments have different bitrates. The average bitrate of the video from lowest to highest quality, including our application header and RTP header in the network packets, are as follows: 738Kbps, 1301 Kbps, 2171Kbps, 3194Kbps.

5.1.3 Tools and techniques

We have implemented our multicast server and client to provide us with statistics of when segments are sent and when they are received in the context of their playback time. The client is also able to provide us statistics regarding packet loss. We use these statistics to make graphs using Gnuplot. We also use Wireshark to inspect the network log.

5.1.4 Test parameters

To create different network scenarios that can occur in the Internet, we provide the network emulator with some parameters that we alter throughout the tests. Values for the different parameters is presented within the description of each experiment. The following parameters is used:

- **Delay:** This is the delay the network emulator adds to the links between the multicast server, the client and the Internet connection. By increasing this parameter, round trip time between the components is increased. The delay is set in both directions of a link so the round trip time is twice the delay.

- **Packet loss**: This parameter decides how much packet loss that is inflicted in the network. The value of packet loss is given as a percentage of the number of packets transmitted over a link.
- **Jitter**: This parameter decides how much jitter that is inflicted in the network. This is given as a percentage of the delay parameter. The delay vary by 20% with 20% jitter.
- **Bandwidth:** This parameter indicates the maximum bandwidth available. By default, this value is 100Mbps on our local test network. For our Internet connection, we have a measured bandwidth of 80Mbps . When specifying this bandwidth parameter, the value is common for both the link to the Internet and the local network unless the bandwidth specified is larger than the bandwidth of our Internet connection. We assume that the client have enough bandwidth to receive a stream in the lowest quality, i.e., 738Kbps, at all times.

We also have some parameters that we can alter in our application to control how much the client should buffer before starting playback of the segments and when the client should start to download a patch. In addition to this, we can decide at which segment the client should join a multicast stream. These parameters are used:

- **Multicast deadline:** This parameter is the deadline for when a multicast segment must be stored in the client's buffer. If a multicast segment is not stored in the buffer within this time, the client considers the segment incomplete and start downloading a patch. The minimum value for this parameter is two seconds as it takes minimum two seconds to receive a segment from the multicast stream. As mentioned in section 3.5.8, this value is based on the first segment that are to be played and a new segment is checked every two seconds after the first check. This means that when the client needs a patch stream for the first 10 segments, the 11th segment is received from the multicast stream and has basically 22 seconds to arrive before it is checked, if the multicast deadline is set to two seconds.
- **Playback time:** This parameter decides the client's deadline for playback, i.e., how many seconds the client should buffer. A segment that is to be played must be in the buffer before this deadline or else the client gets a playback deadline miss.
- **Client arrival time:** This parameter decides at which segment the client joins a multicast stream. If this parameter is set to 100 then the client joins the multicast stream at segment 100 and needs a patch stream for the first 100 segments. To achieve this the multicast server never transmit the first 100 segments. The client receives the first 100 segments from the patch server at the same time as it receives

the next 100 segments from the multicast server, as seen in figure 5.2. In this figure the multicast server transmits the first 100 segments to illustrate that the multicast server starts transmitting at segment 0 in real life.

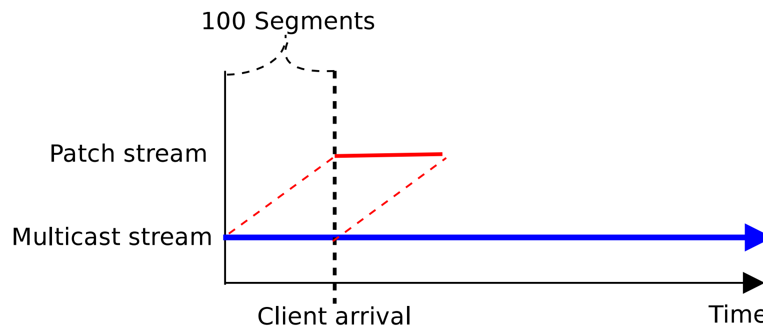


Figure 5.2: Client receiving a patch stream and a multicast stream

5.2 Experiments

In this section we perform experiments to investigate how the patching scheme copes with the different challenges that Internet provides. For each experiment we analyze and discuss the results within the section of the experiment.

5.2.1 Proof of concept

Before we start testing how our implementation manages the different challenges in the network, we perform an experiment to see if our implementation actually works. This experiment is performed in a perfect network environment where we do not add any delay or inflict the network with packet loss, jitter or reordering of packets. We have a look at the sending rate of the multicast streams as well as how they are received. The client to joins the multicast stream at segment 100 to test our patching functionality. The client wait time for this test is three seconds. If we knew that the client would not need a patch stream for this transmission we could have set the wait time close to two seconds. Our test parameters for this test are shown in table 5.1.

Delay	0
Packet loss	0
Jitter	0
Bandwidth	Default
Multicast deadline	2000 ms
Playback time	3000 ms
Total number of segments	500
Client arrival time	100

Table 5.1: Proof of concept: Test parameters

As we can see, the first segment has been given 2000 milliseconds to arrive, which is the transmission time of a segment. If it is not received within this time limit, it means that it is not completely received, and the client needs a patch for that segment. We have given this patch 1000 milliseconds to download (3000 milliseconds - 2000 milliseconds). The reason we have set such high download time for the patch is that the patch servers are located in the Internet where challenges like packet loss, jitter and reordering can affect the download time of the patch.

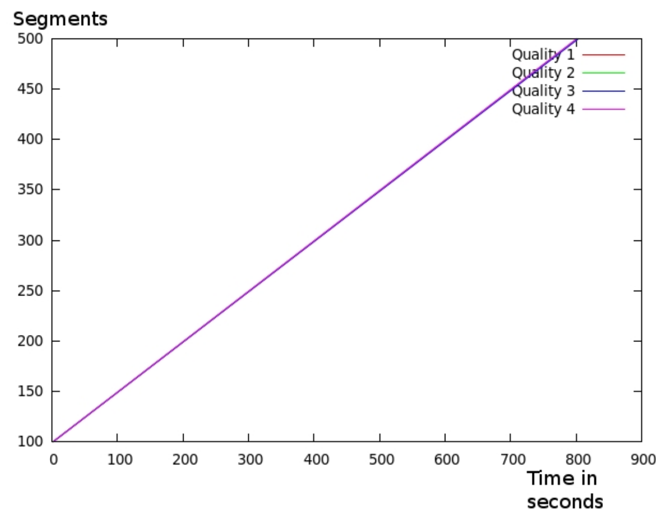


Figure 5.3: Multicast server sending rate

The server starts sending at segment 100 because we wanted the client to join the multicast stream at segment 100. Since we only have one client, the multicast server therefore

started sending at segment 100 when the client requested a stream. The figure 5.3 give a view of the sending rate of the stream from the server. As we can see from the figure, the channels are well synchronized and it takes approximately two seconds to send a segment out on to the network. The server sends 400 segments with a total transmission time of 800 seconds

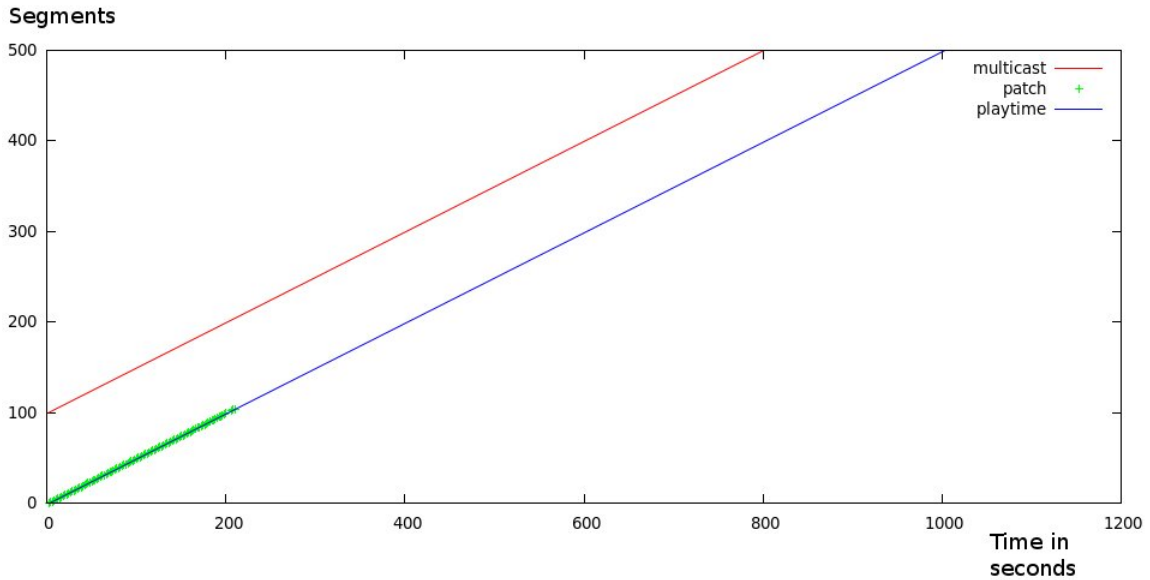


Figure 5.4: Client receiving segments

The figure 5.4 shows the time of when segments are received and whether they are received from the multicast stream or from the patch stream. The figure proves that our implementation of our patching scheme works. As seen in the figure, the client is able to store the segments from the multicast stream in its buffer while receiving a patch stream. When the patch stream ends, the client starts playing segments from it's buffer. The figure 5.4 corresponds to the figure 2.9 which describes the patching concept.

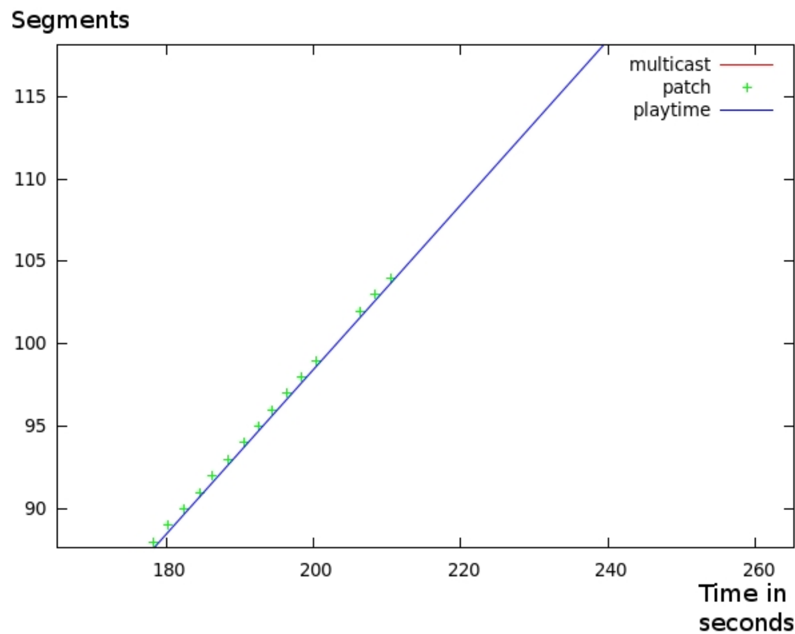


Figure 5.5: Section of client receiving segments

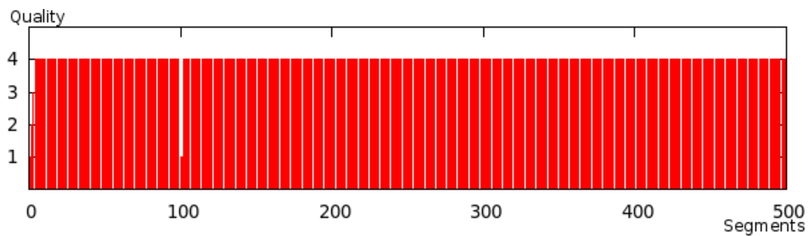


Figure 5.6: Playback quality

Figure 5.5 gives us a closer view of a section of the graph in figure 5.4. This section of the graph gives us a view of the last segments the client downloads from the patch servers. As we can see in the figure there is a little gap between the time of receiving segment 99 and 102. The segments 100 and 101 are as expected received from the multicast stream, but the segments 102, 103 and 104 are patched due to packet loss during an increase of quality. The increase of quality is illustrated in figure 5.6, where we can see that the quality is increased from segment 103 to segment 104. The client downloads a patch for segment 102 in quality 1 because that is the quality it originally received from the multicast stream. Segment 103 is patched in quality 4 because the client did not receive any packets from the multicast stream as the client increased the quality gradually from 1 to 4, and quality 4 is the clients recommended quality at the time of downloading the

patch. We expected the client to receive some packets of segment 103 from the multicast stream during the increase of quality. However, we are experimenting in a real network and the IGMP membership request messages could be lost. Segment 104 is patched in quality 4 because the client missed the first part of segment 104 from the multicast stream. The qualities are illustrated with values from 1 to 4, where quality 4 is the highest quality. If quality 0 is shown, it means that the segment was not received in time for playback.

The first 100 segments are received from the patch server at the same time as the next 100 segments are received from the multicast server. We can see that the segments downloaded from the patch server are received in the same quality as the multicast segments received at the same time. As mentioned in chapter 3.5.3, the client starts receiving segments in the lowest quality and gradually increases the quality up to the maximum limit if all goes well. The graphs shows us that the client is able to receive all segments before playtime which allows it to receive the best quality. When the client figured out that segments 0, 1 and 2 were received within playtime, the quality was increased in both the patch stream and the multicast stream.

Summary As we can see from the results above we have managed to implement a quality adapted system for receiving multicast video. If the client misses the start of the multicast stream, it sends requests to the patch server for the parts of the video that are missing. The segments that were received from the multicast stream and the patch stream at the same time, were received in the same quality. We have proven that our implementation works, and we move on to perform experiments on our implementation regarding delay, packet loss, jitter and reordering.

5.2.2 Bandwidth

The Internet is not a heterogeneous environment, so different clients have different network performance specifications. In this test we have a look at how our application handles different bandwidth settings when listening to a multicast stream while getting a patch stream from the patch servers. We have a look at how our implementation use quality adaptation to make sure that the client can receive segments before playtime. Our parameters for this test are shown in table 5.2:

Delay	50
Packet loss	0
Jitter	0
Reordering	0
Bandwidth	8Mbps -> 1Mbps
Multicast deadline	2000 ms
Playback time	10000 ms
Total number of segments	300
Client arrival time	100

Table 5.2: Bandwidth: Test parameters

We have added a delay of 50 milliseconds to our network connection because delay is normal in the Internet. Adding this delay to our network gives us 50 milliseconds delay to our multicast server and adding 50 milliseconds to the delay to the patch server. The measured round trip time from our lab at Simula to the patch servers located in Oslo and Tromsø are approximately 1 millisecond and 23 milliseconds. However, the patch server in Tromsø is never used in any of our experiments. The round trip time to the patch server is approximately 100 milliseconds. We choose a round trip time of 100 milliseconds because it is well within the test values used in the experiments in the article [37], and we feel it is a realistic value in the Internet.

The average bitrate of a segment of highest quality is 3194 Kbps. When downloading a patch stream and a multicast stream simultaneously at highest quality, we get a bitrate requirement for full quality of total 6388kbps. Given that these calculations are based on average values, we believe that a bandwidth of 8 Mbps is enough to handle two incoming streams at highest quality simultaneously. This is a bandwidth that is available to many households today.

We run three tests with a limited bandwidth of 8Mbps, 6Mbps, 4Mbps, 2Mbps and 1Mbps, respectively. This values are based on the different qualities combined bit rate when receiving a stream from the multicast server and the patch server at the same time. The multicast deadline is still set to 2000 milliseconds because we do not expect transmission time of a segment from the multicast stream to be longer than 2000 milliseconds, but we have increased the playback time to 10000 milliseconds compared to our previous test. The reason we have increased the playback time to 10000 milliseconds is that we do not know how the limited bandwidth affects our client and we want the client to give patched segments plenty of time to arrive to limit playback deadline misses. Moreover, as stated

in [37], 10 seconds client wait time is acceptable. The client joins the multicast stream at segment 100, i.e., the client must download the segment 0 - 99 from the patch server. We have chosen to stream 300 segments, as we believe this is enough to investigate how the client copes with a limited available bandwidth.

The results from our test runs are presented with three graphs for each run. We show a graph for when segments are received in contrast to playback deadline, a graph for playback quality and a graph illustrating the missing portions of a segment the client received from the multicast stream. The graphs illustrating when segments are received does not show when segments from the multicast stream arrived, except for figure 5.7 which gives us a view of the reception of all segments where the client has an available bandwidth of 2Mbps. This is because the segments from the multicast stream potentially arrive 200 seconds before playback. By including the reception of segments from the multicast stream, the graphs would not scale well. We have some abbreviations, shown in table 5.3, to simplify the figure 5.7:

msr	multicast segment received
psr	patch segment received
ppsr	partially patched segment received
p	playtime

Table 5.3: Abbreviations for figure 5.7

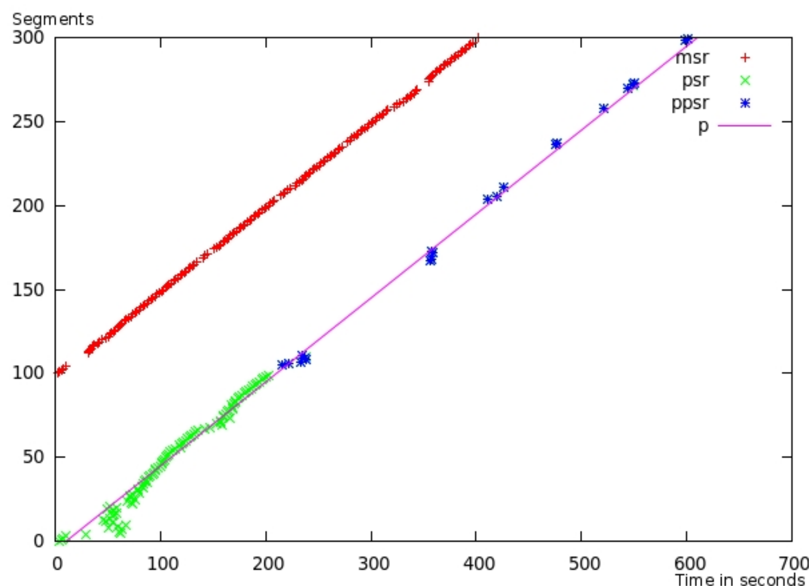
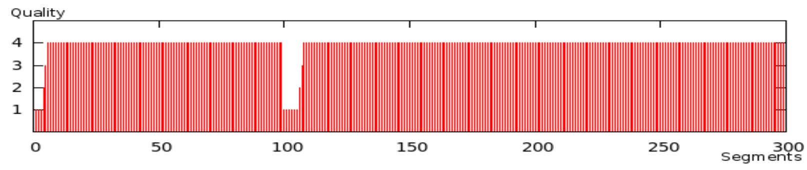


Figure 5.7: Client receiving segments with bandwidth 2Mbps

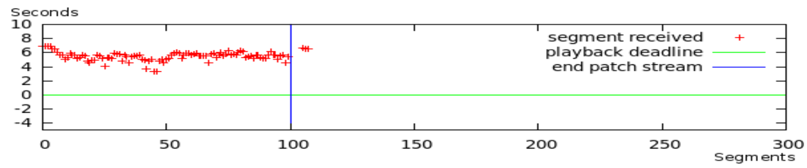
The figure 5.7 shows when the segments are received at the client with an available bandwidth of 2 Mbps. For a detailed view of quality adaptation, data loss and when segments are received before playtime with this bandwidth setting, we refer to figure 5.11. At the start of the stream, the client increased quality as it was able to receive segments in time for playback. When it increased the quality, the patched segments were not received within time for playback, and the client experienced deadline misses. These deadline misses caused the client to request a lower quality, and the client was then able to receive a lower quality within time for playback after a while. However, the deadline misses had serious effect on the download time of the segments that followed, even though quality was reduced. The reason for this is that the high quality segments that first caused a deadline miss occupied the network for a long time, which means that they were still in transfer when the requests for lower quality segments were sent. The competition for the available bandwidth became large and caused a low quality segment to have a longer transmission time than it would have if it were the only segment sent over the network at that time. This competition for the available bandwidth also had another effect. As we can see in the figures 5.7 and 5.11(b) there are great fluctuations in the arrival time of the segments from the patch stream in the beginning of the stream. This is because packets were dropped in the network, and TCP retransmits each packet dropped, causing a longer transmission time. The packets were dropped by our router because the amount of data sent from the patch server and multicast server was more than the link between our router and client managed to forward to the client at that time.

When the router dropped packets, it also had an effect on the multicast stream received at the same time. The holes in the multicast stream means that a segment from the multicast stream was not completely received. As seen in the figure 5.7, these holes are patched at a later time, closer to playback. The holes are not repaired earlier due to reasons explained in section 3.5.7.

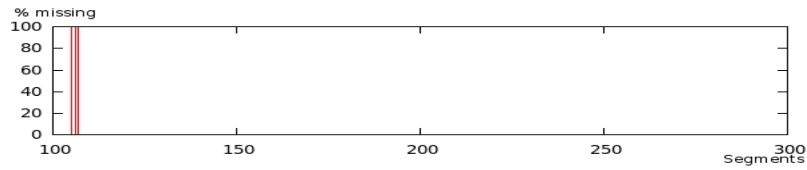
As seen in figure 5.11(a), the client received the multicast stream and the patch stream at the same time, it was not able to receive a higher quality than quality 2. When the client only received segments from the multicast stream it was able to receive a stream in quality 3.



(a) Playback quality

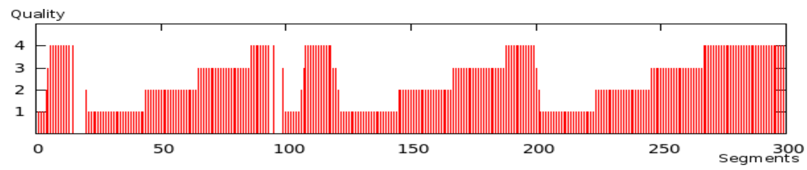


(b) Patch delivery time in seconds before playout

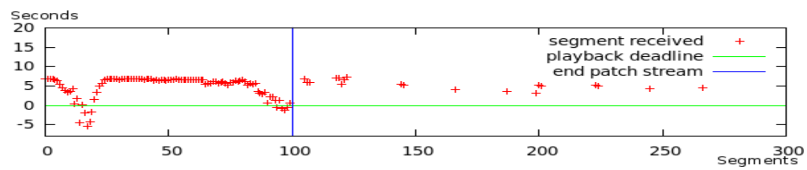


(c) Percent of segment data missing from the multicast stream

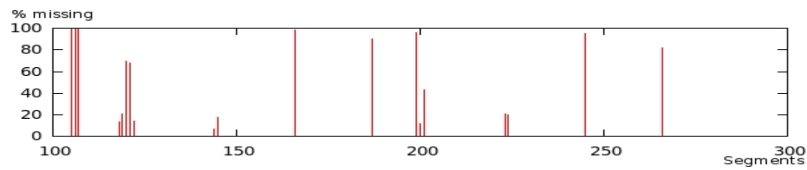
Figure 5.8: Results with bandwidth 8Mbps



(a) Playback quality

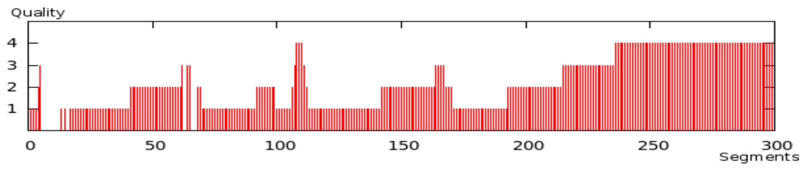


(b) Patch delivery time in seconds before playout

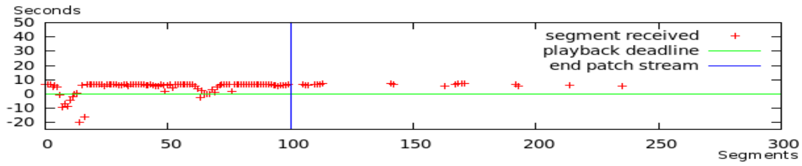


(c) Percent of segment data missing from the multicast stream

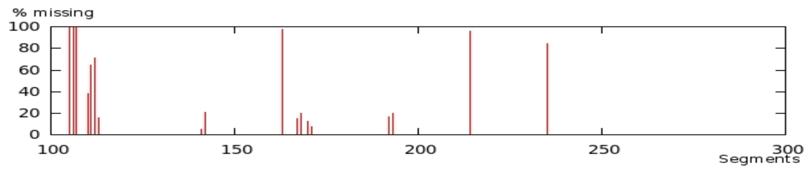
Figure 5.9: Results with bandwidth 6Mbps



(a) Playback quality

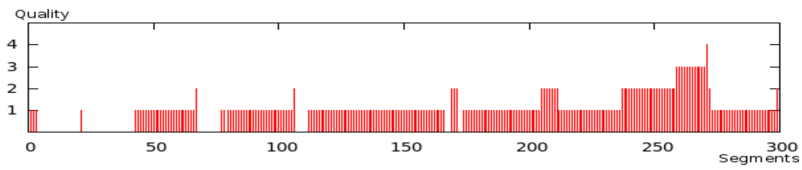


(b) Patch delivery time in seconds before playout

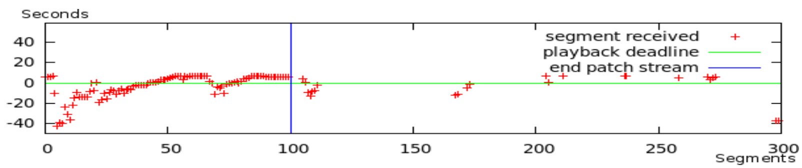


(c) Percent of segment data missing from the multicast stream

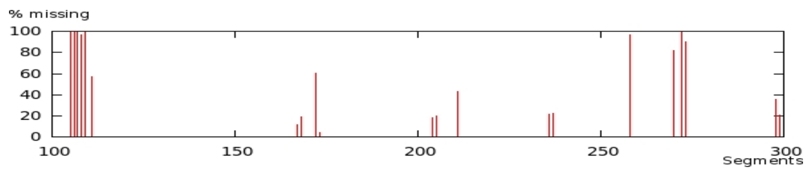
Figure 5.10: Results with bandwidth 4Mbps



(a) Playback quality

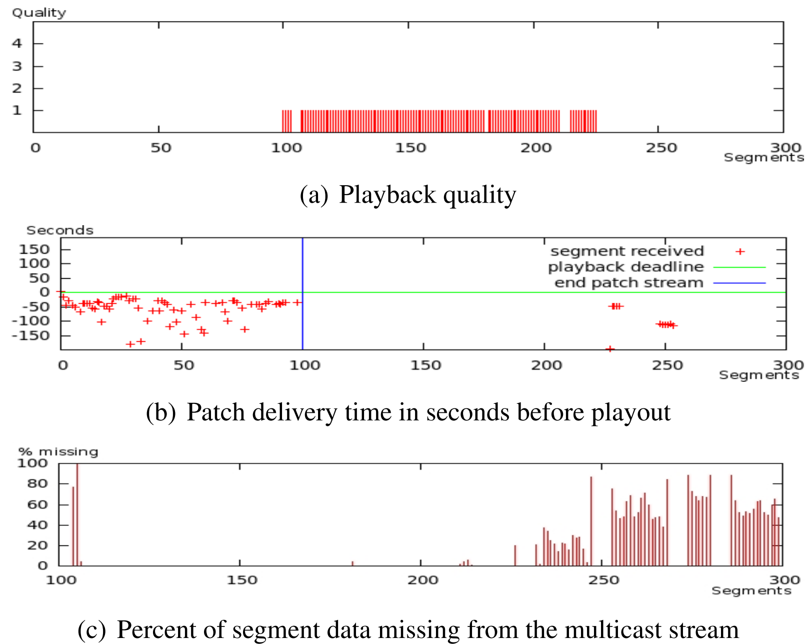


(b) Patch delivery time in seconds before playout



(c) Percent of segment data missing from the multicast stream

Figure 5.11: Results with bandwidth 2Mbps



(a) Playback quality

(b) Patch delivery time in seconds before playout

(c) Percent of segment data missing from the multicast stream

Figure 5.12: Results with bandwidth 1Mbps

As seen in the graphs in figure 5.8, the client were able to receive segments in max quality when downloading a patch stream simultaneously with receiving a multicast stream, when having 8Mbps available bandwidth. Figure 5.8(b) gives us a view of the arrival time of patched and partially patched segments. As we can see in the figure, all segments arrived within time of playback. As expected, the transmission time of low quality segments is less than the transmission time of higher quality segments.

Figure 5.8(a) shows us the quality of each segment. The segments that are received from the multicast stream and the patch stream at the same time are in the same quality. Notice that the first 100 segments are from the patch stream and the next 100 segments are from the multicast stream, meaning that, for instance, segment 150 is received from the multicast stream at the same time as segment 50 is received from the patch stream.

The figure 5.8(c) shows us the amount of lost data from the multicast stream per segment and therefore only displays the segments from segment 100 to 300. We can see that the segments that were not completely received from the multicast stream, corresponds to when the client changed quality in figure 5.8(a). This is because at a point in time, the client never listens to a multicast stream when changing from one multicast channel to another. The rest of the segments from the multicast stream were received without any packet loss.

When we reduced the available bandwidth to 6Mbps the client was not able to receive the highest quality when downloading a patch stream and simultaneously receiving a multicast stream, as seen in the graphs in figure 5.9. The client was able to receive both streams in highest quality for a little while, but the playback time caught up with the download speed and the client experienced deadline misses. This concurs with the theoretical calculations. Downloading both streams at the same time with a combined bitrate of 6388Kbps with an available bandwidth of 6Mbps does not work. However, in theory, the client is able to receive the streams in the second highest quality with a combined bitrate of 4342Kbps, which seems to be correct. From segment 200 the client only received segments from the multicast stream, and it was able to receive segments in the highest quality. When the client finished receiving the patch stream from the patch server, it is able to receive a multicast stream at the highest quality. The client experienced some packet loss in the multicast stream in is test as well. This is because quality changes, and that packets was dropped by the router when the client tried to download a to high quality.

When we lowered the available bandwidth to 4Mbps, seen in the graphs in figure 5.10, we experienced the same results as with 6Mbps bandwidth. The only difference is that the client was not able to receive as high quality as with 6Mbps when downloading a patch simultaneously with the multicast stream.

We finally did a test with bandwidth 1Mbps. As seen in the graphs in figure 5.12, this does not work well. A patch stream and a multicast stream received at the same time require more bandwidth than the client has available. In scenarios like this it would be reasonable for the client to choose to only download from the patch server and not receive any segments from the multicast server at all. This is the only option the client has if the whole video is to be received. If we compare the two graphs 5.12(b) and 5.12(c), we can see that many of the segments that miss data are not received in the graph 5.12(b). This is due to the problem mentioned in chapter 4.3. In this case, the client needs more than 300 threads to download the missing parts, causing that the generation of some threads are forced to wait. Some of the segments actually did not arrive at all. We believe that there might be a problem with the curl framework or our use of curl when the request times out, making some of the threads stall. When the threads stall, they steal a lot of client system resources, making it difficult to receive the multicast stream. We can see in graph 5.12(c) that a lot of data from the multicast stream is not received when the problems of thread generation occurred at segment 230.

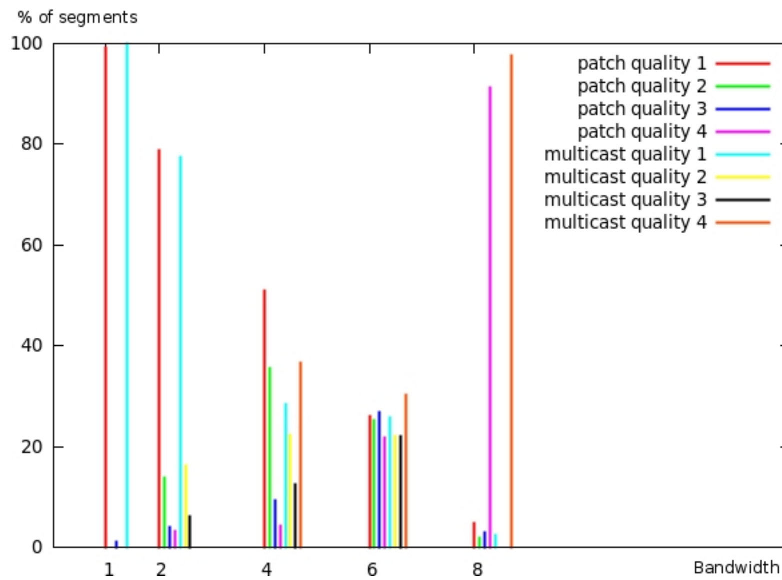


Figure 5.13: The % of received segments in different qualities

Figure 5.13 gives us an overview of how many percent of each segments that are downloaded in a certain quality at a certain bandwidth. As seen in the graph, we are able to receive much higher quality when we have 8Mbps bandwidth than 2Mbps. We can also see that our solution for quality adaptation causes a lot of quality changes when the available bandwidth is limited to 4 Mbps and 6 Mbps.

We saw that with an available bandwidth of 2 Mbps, it had a dramatic effect when the client tried to receive a quality that was to high for that bandwidth. To avoid this effect, we propose that the client have knowledge about the available bandwidth. This can be done by always downloading the first segment from the patch server, regardless of whether the client is arriving late to the multicast stream or not. The client can then measure the time it takes to download one segment to calculate the available bandwidth. With this knowledge the client can avoid trying to download a higher quality than what is theoretically possible and it would avoid the problems of overloading the network causing the router to drop packets. However, if a segment is not received from the patch server in time for playback, the client should abort the download of the segment. The segment is played if it arrives too late anyway. By aborting the download for a segment that missed its deadline, we would limit the competition for the available bandwidth. The information the client has about the network also leads to more consistent quality, as the client would experience fewer deadline misses.

The client also experienced some deadline misses when some of the partially patched multicast segments arrived to late for playback. This is because the multicast segments

were originally tried to download in a higher quality than the client bandwidth can handle. The repair for the missing parts of the segment is downloaded in the same quality as the parts received from the multicast stream. As we can see in figure 5.11(c), the missing parts of the segments are at times above 60% of the total size of the segments, many times close to 100%. This is a lot of data to download if the quality of that data is too high. We could argue that this would not happen if we implemented the solution to the quality problem mentioned above, but it could still happen even though the client is not arriving late to the multicast stream. If the client has measured its bandwidth and finds out that it has just enough to download the highest quality and then experiences massive packet loss on one of the segments, this could be a problem. The client would have to download the missing parts of the segment in the same quality as the part of the segment already received, being the highest quality. This issue also requires a solution as the viewer will not be pleased with a lot of deadline misses. As a solution to this issue the client has to evaluate if it is cheaper to download a completely new segment in a lower quality than to download the missing parts of the segment in the original quality. In the experiment with an available bandwidth of 2 Mbps, the client would probably not have experienced deadline misses on the segments received from the multicast stream if it was to download completely new segments from the patch servers in a lower quality, as the combined bandwidth requirements for the patch and the multicast stream would be less than 2Mbps.

When a client has experienced a deadline miss, the client waits a minimum of 20 segments before it tries to increase quality again. As we have seen in these tests, this causes the client to receive a lower quality than it is capable of for a long period of time. It is clear that a 20 segments waiting period is two long, and the algorithm for quality adaptation should be implemented with a shorter waiting period.

Summary In this experiment we have seen that by limiting the available bandwidth the client is in need of quality adaptation to be able to receive the video within time of playback. However, some issues has been discovered while performing this experiment. We have suggested that the client should have some knowledge about the available network bandwidth to avoid downloading a quality that is theoretically impossible. When downloading segments from the patch server that are of higher quality than the client is capable of receiving within playback, we have seen that the transmission of these segments increases the download time of segments that are to follow. We have proposed that client knowledge of network resources limits this problem, as it never downloads a quality that is higher than what is theoretically possible. We also suggested that the transmission of a patch segment should be aborted if the client experience a deadline miss of that segment. This would remove this segment from the competition for the available bandwidth and the following segments from the patch server would have a faster transmission time. If the client downloads a repair for a segment in a higher quality than the client is able to

receive, we argue that the client should consider downloading a completely new segment in a lower quality from the patch server instead of the repairing the multicasted segment. We have also seen that our algorithm for quality change are not able to give the client the best possible quality at all times. The waiting period of 20 segments before the client increase quality after a deadline miss is not very effective as it takes a long time before the client fully utilizes the available bandwidth again. The client should be implemented with a shorter waiting period and give the client knowledge over the available network bandwidth to increase the quality more quickly by increasing to the maximum quality that is theoretically possible. The knowledge of network bandwidth would ,as mentioned, prevent the client of downloading a quality that is theoretically impossible.

5.2.3 Packet loss

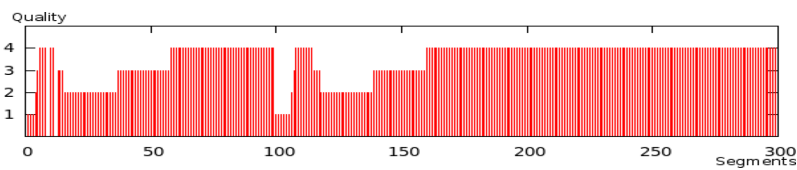
Packet loss is a common obstacle in the Internet and it is a challenge for video streaming. When high rates of packet losses occur in the network it does not give the viewer an acceptable view of the video unless countermeasures are taken. In this experiment, we have a look at the effect packet loss has on our implementation, and if the idea of downloading a patch for the packets that are lost in the multicast stream is able to provide the client with video in time for playback. The following test parameters is used, as shown in table 5.4:

Delay	50
Packet loss	0,2% -> 5%
Jitter	0
Bandwidth	8Mbps
Multicast deadline	2000 ms
Playback time	10000 ms
Total number of segments	300
Client arrival time	100

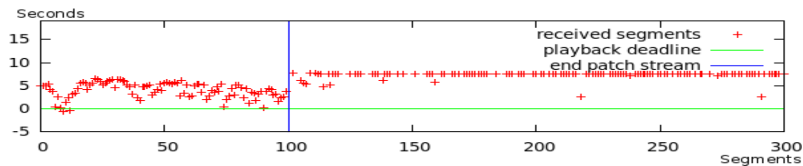
Table 5.4: Packet loss: Test parameters

Again we add a delay of 50 milliseconds to the network in both directions. Our experiments is exposed to packet losses of 0,2%, 0,5%, 1%, 3% and 5% in both directions of the network paths, as we see this as reasonable values of packet loss in the Internet. We set the maximum available bandwidth to 8Mbps. This is a bandwidth that is present in

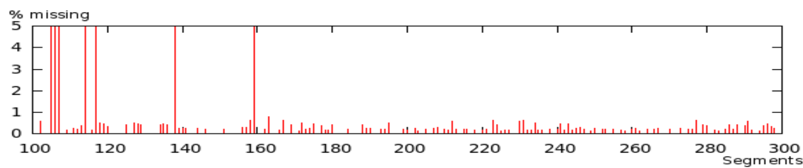
many households today and as shown in the previous experiment, it is enough to receive a stream in highest quality from the multicast server and the patch server at the same time. The multicast deadline, playback time, total number of segments and client arrival time are the same as for the last experiment. The graphs that are to be presented shows the arrival time of a patched segment, quality adaptation and the percentage of segment data we have to download from the patch servers when packet loss in the multicast stream are detected. For the graph of segment arrival time, we have limited the view by excluding the arrival time of complete segments from the multicast stream as they arrive approximately 200 seconds before playback time. The graph therefore only shows when patched and partially patched segments have arrived.



(a) Playback quality

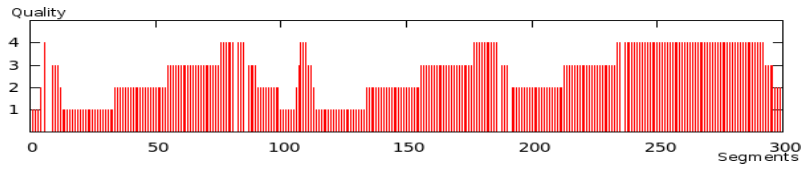


(b) Patch delivery time in seconds before playout

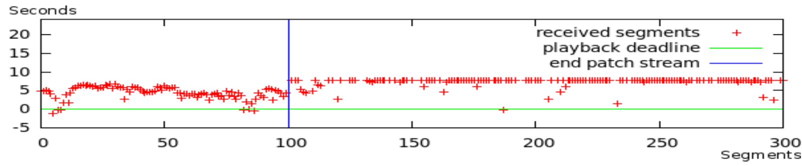


(c) Percent of segment data missing from the multicast stream

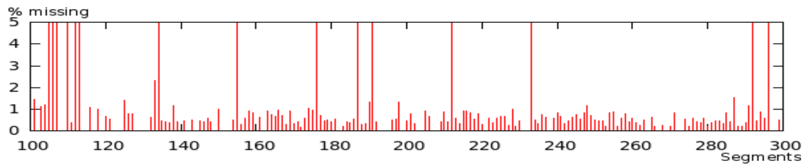
Figure 5.14: Results with packet loss 0.2%



(a) Playback quality

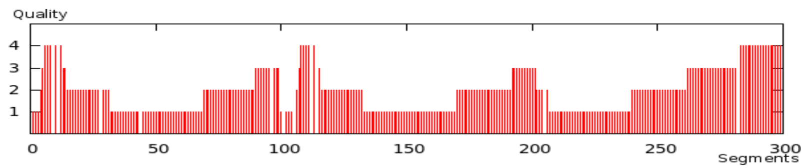


(b) Patch delivery time in seconds before playout

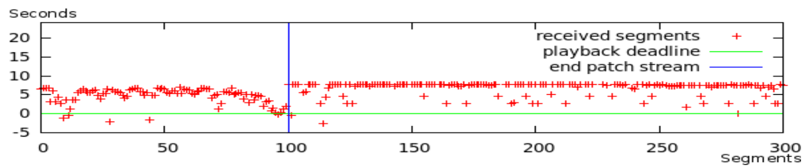


(c) Percent of segment data missing from the multicast stream

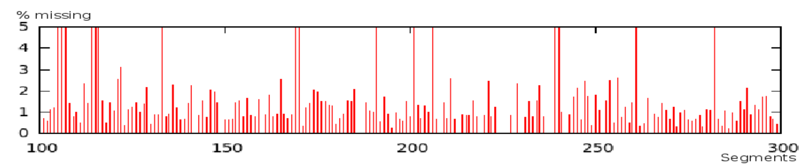
Figure 5.15: Results with packet loss 0.5%



(a) Playback quality

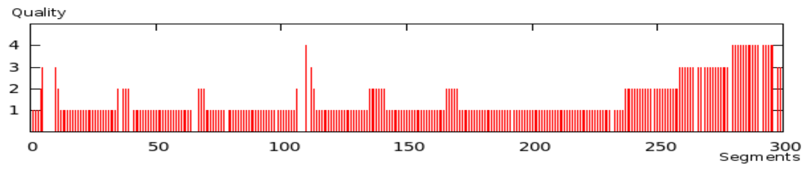


(b) Patch delivery time in seconds before playout

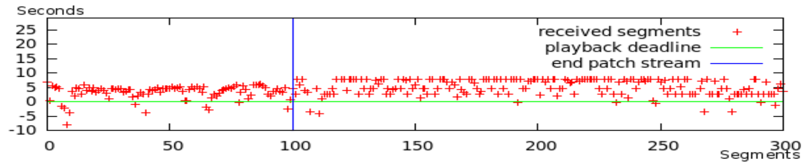


(c) Percent of segment data missing from the multicast stream

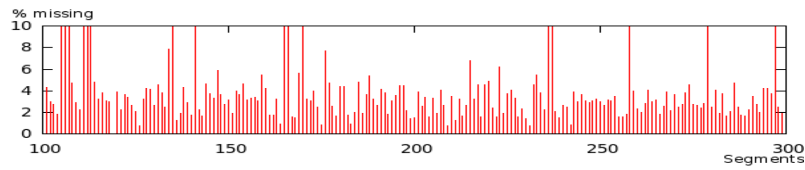
Figure 5.16: Results with packet loss 1%



(a) Playback quality

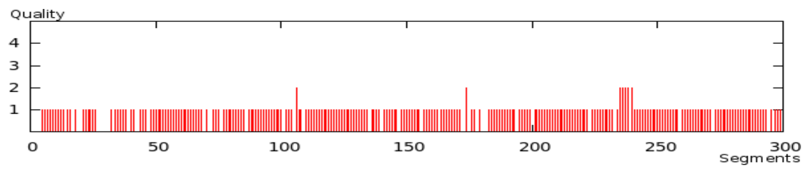


(b) Patch delivery time in seconds before playout

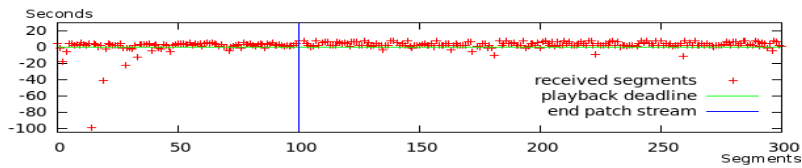


(c) Percent of segment data missing from the multicast stream

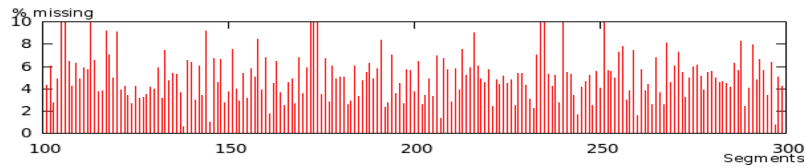
Figure 5.17: Results with packet loss 3%



(a) Playback quality



(b) Patch delivery time in seconds before playout



(c) Percent of segment data missing from the multicast stream

Figure 5.18: Results with packet loss 5%

In this test we, have seen that by increasing the percentage of packet loss in the network, it has a dramatic effect on the reception of video segments. We first exposed the network with a packet loss rate of 0,2%. The results are shown in figure 5.14. Even at this low rate of packet loss the client did experience 2 deadline misses. The segments that missed their deadline were in the highest quality. The quality was then decreased, and the client was able to receive all segments within playback time. As we can see, in graph 5.14(b), there are great fluctuations in the arrival time of the segments from the patch stream, but when patching is used to repair segments from the multicast stream, the arrival time is more consistent. As seen in graph 5.14(c), the missing data in the segments from the multicast stream is about 0,2% average if we exclude the ratings above 1%. Ratings above 1% are the effects of a change of quality.

We then increased the packet loss rate to 0,5%. As seen in the graphs in figure 5.15, the client experienced much of the same as with packet loss rate of 0,2%, but it did experience a deadline miss when repairing a segment from the multicast stream[1]. This is because parts of this segment were lost due to an increase in quality which means that almost the whole segment had to be patched. With normal packet loss all segment were received within deadline. As we can see in figure 5.15(c), the missing parts of the multicasted segments are average approximately 0,5% if we exclude the ratings above 5%. The ratings above 5% are the effects of a change of quality. It works well to download the missing parts of the segments from the patch servers and all are received within playback time.

When increasing the packet loss rate to 1% we experienced much of the same as in the experiment with packet loss of 0,5%. The major difference is that the client was not able to receive as high quality. As we can see in graph 5.16(b), the client also had some deadline misses when downloading a patch for an incomplete multicast segment. The client had to download a patch for 97% of segment 114 and 80% of segment 282. But for segment 101, the client only had to patch 0,7% of the segment, and it still experienced a deadline miss. The missing parts of these segments are downloaded from the patch server in the same quality as the data received from the multicast stream for these segments. As we discussed in section 5.2.2, when the client is missing a massive part of the segment, the client should consider if it is cheaper to download a completely new segment from the patch server in a lower quality, than to request a partial patch in the original quality. In figure 5.16(c), we can see that the missing parts of the multicasted segments are in average 1% if we exclude the results above 4% that are due to quality changes.

We also performed experiments with packet loss rates of 3% and 5% as seen in the figures 5.17 and 5.18. As the packet loss rates were increased, the client experienced a growing number of deadline misses, even at the lowest quality level. At these rates of packet loss, it is a great challenge to give the client an acceptable view of the video.

[1]For segment quality is 0 although no deadline miss is shown in figure 5.15. This is due to a bug with our threads using curl. We ignore it here as it only happened to very few segments in all tests.

We have seen in these tests that the effect of packet loss in the network can be dramatic. However patching is a good technique to repair segments from the multicast stream up to approximately 3% packet loss. When the rates of packet loss come above 3 percent it becomes a real challenge to give the client an acceptable view of the video, even when the client only had to repair segments from the multicast stream. By using quality adaptation the client can tolerate some packet loss but there are clearly limits in what quality adaptation can compensate for. Another solution to cope with large amounts of packet losses is to increase client wait time and buffer more segments before consuming them. However, this is only effective as long as the client is able to receive a segment in average every two seconds or less. If the average gap between the arrival of the segments is above two seconds then it does not matter how many segments the client buffers before consuming them. In this case the client loses in the long run. Another important issue is that even with low rates of packet loss there is always a chance that a segment can take a very long time to download, as the same packet can potentially be lost over and over again. This was observed with a packet loss rate of 1% were segment 101 missed the deadline by 0,5 seconds even though we only had to download 0,7% of the segment from the patch server. Another problem with packet loss is that many segments can be in the process of downloading at the same time. This is shown in the fluctuations in the arrival time of the segments. When a segment experiences a lot of packet losses during download, TCP retransmits the lost packets and that can cause a longer download time than two seconds. This means that it is not given that a segment is completely downloaded before the download of the next segment begins, causing that many segments compete for the same resources in the network at the same time.

Summary Packet loss is a great challenge for video streaming in the Internet, and it is difficult to avoid deadline misses as the rates of packet losses are increased. However, we have argued that using quality adaptation and giving the client enough time to download the patched segments, it can camouflage packet loss up to approximately 3% loss rate. We have also seen that packet loss increases the transmission time of a segment from the patch server, because TCP retransmits every packet dropped, and the request time for the packet that is to be retransmitted is equal to the delay between the client and the server. If a lot of packets are dropped, the transmission of one segment can cause a longer transmission time of another segment, as one segment might not be completely downloaded before the download of the next segment starts.

5.2.4 Jitter and reordering

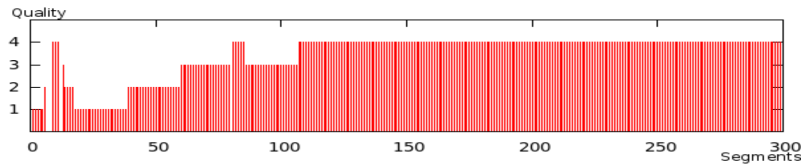
Jitter is the variation of delay in the network and can occur in the Internet as a consequence of congestion in the network or that there is a change of network path. As a consequence

of jitter, reordering can occur. This can happen when packets choose different routes or when some packets are lost and get retransmitted. In this experiment we investigate what effects jitter and reordering have on video streaming over the Internet. Initially, we wanted to look at jitter and reordering in two different experiments, but Netem, from version 1.1, reorders packets when jitter is enabled. In the documentation of netem, there is a solution to how to guarantee that the packets remained in correct order but we did not get that to work. The test parameters used are shown in table 5.5:

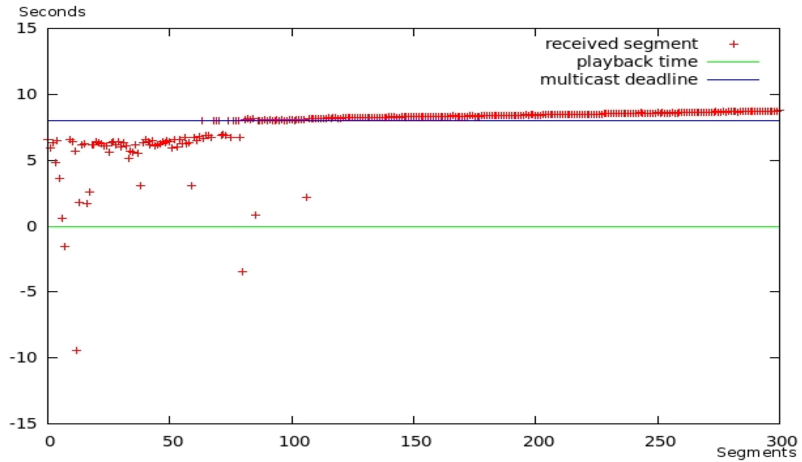
Delay	50ms -> 185ms
Packet loss	0
Jitter	2,5 -> 50%
Bandwidth	8Mbps
Multicast deadline	2000 ms
Playback time	10000 ms
Total number of segments	300
Client arrival time	0

Table 5.5: Jitter: Test parameters

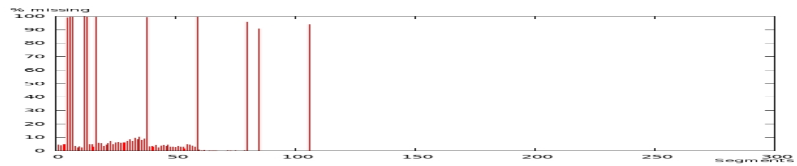
In this experiment we use delays from 50ms to 185ms in both directions. We do not inflict the network with any packet loss as we already have experimented with the effect of packet loss. The jitter percentage is set from 2,5% to 50%, meaning that the delay varies by the given %. The delay and jitter values are based on values retrieved by sending ping requests to different locations in the Internet. The highest round trip time we measured was to the Mongolian newspaper, The Mongol Messenger[26], where we measured a round trip time of 370 milliseconds. However, the jitter values we are using in our experiments exceed any jitter values we retrieved from ping requests, where the highest jitter value were 5 %. The reason we have chosen such large jitter values is because we expect that jitter and reordering have minimal effect on our scheme. The bandwidth limitation, multicast deadline, playback time and total number of segments are the same as for the previous test. The client arrival time is set to zero, meaning that the client receives all segments from the multicast stream and does not need a patch stream in the beginning. This is because the multicast deadline is essential in this test. As the client buffers 10 seconds of video before consuming, we do not focus on playback deadline misses. What we are interested to have a look at in this test is the unnecessary patching the client has to do when jitter causes the multicast segments not to arrive before the multicast deadline.



(a) Playback quality



(b) Segment arrival time in seconds before playout



(c) Percent of multicasted segment data missing from the multicast stream

Figure 5.19: Results with 185 milliseconds delay and 50% jitter

In this experiment, we have had a look at the effect network jitter has on our implementation. We did test runs with a delay of 50 milliseconds and jitter of 3%, 10% and 20% without noticing any effect. We then increased the delay to 185 milliseconds with the jitter of 10% to 40 % but did not see any effect. When we increased the jitter percentage to 50%, we noticed some effect. Theoretically we should have noticed some effect with lower percentage values, but the reason we do not is our implementation. Our implementation is controlled by timers. In theory the client should do a check on the multicast buffer every two seconds to see if a segment from the multicast stream has arrived. However the clocks are not 100% accurate because our implementation runs on a single CPU and it might take a few milliseconds more than two seconds before the segment is checked. This little delay is just enough to give a segment from the multicast stream time to be received before the client does a check on the segment, which gives us some milliseconds

extra buffering. However this gives us an indication that within reasonable delay and jitter values the client does not notice any effect when it has this little buffering. In theory the client notices the effect of jitter when it has to request a patch for segments that did not arrive within the multicast deadline. This would be unnecessary patching, as the multicast segment arrives eventually, unless it is dropped due to packet loss. The effect of unnecessary patching would be larger if the client experienced packet loss in the network. As seen in the previous experiment it could take a long time to download even small parts of data. With unnecessary patching the risk of this increases.

In figure 5.19, we can see the results we retrieved when we added a delay of 185 milliseconds and jitter of 50% to the network. As we can see, the missing portions of the segments are about 5% average, if we exclude the loss caused by a change of quality. The client only experiences incomplete segments in the beginning of the stream and after some time it does not miss any portions of the segments from the multicast stream. The reason for this is that the average interval between delivery of the multicast segments is just below 2 seconds. This means that our calculation of the processing delay in section 4.1.5, is not accurate. A time period of 600 seconds results in about one second of unwanted buffering. This has a minimal effect for the client as this little unwanted buffering does not cause any problems of available memory. We could argue that it has an effect if the video stream is long enough, but with the available memory resources in modern computers it is unlikely that this is a problem if the video is of reasonable length. For a live broadcast, which potentially never ends, the problem is nonexistent as the video never is transmitted faster than it is recorded.

Summary By this test, we can say that buffering is needed to compensate for jitter. The more jitter that's in the network the more we should buffer. However, with reasonable jitter values jitter is not a problem for our implementation. Jitter is more of a challenge for interactive streaming like video conferences and audio chat. In these scenarios, the packets must be delivered within the time limits or else it has a noticeable effect for the user. We have also discovered that our technique of storing data packets based on byte offset managed to cope with the challenge of reordering of packets. When we stream a video in a scenario like this, we are able to see the whole video in the player module without any errors. All received packets are stored in the correct order within the correct segments. Another interesting result we retrieved from this experiment is that the download time of a patch is very long for some segments even though we did not expose the network to any packet loss. This leads us to our final experiment.

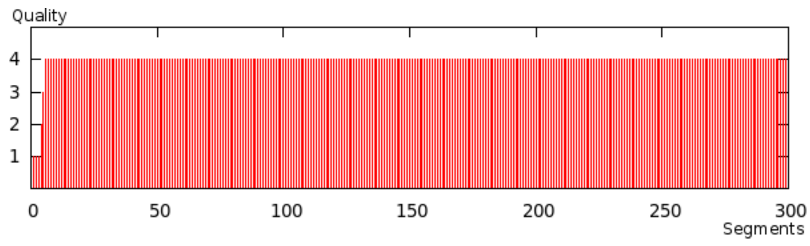
5.2.5 Delay

The Internet is a heterogeneous environment and the delay may vary. This has nothing to say for our multicast stream as this is transmitted over UDP and we therefore only have one way communication. Delay therefore only affect our patch stream as this is TCP, which is two way communication. In this experiment we investigate if delay has some effect on our patch stream. We perform a test where the client download all the video content from the patch servers and not receive any segments from the multicast channels. The following test parameters is used, as shown in table: 5.6.

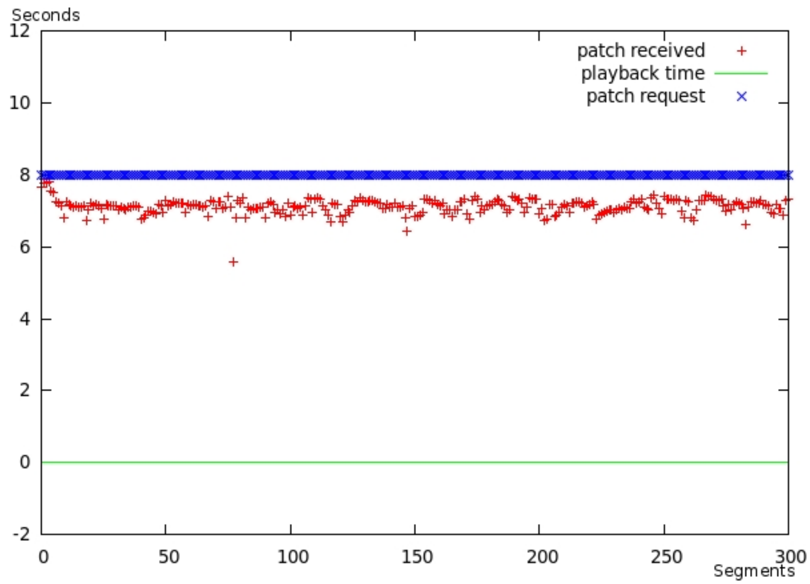
Delay	0 ms, 50ms and 185ms
Packet loss	0
Jitter	0%
Bandwidth	8Mbps
Patch time	2000 ms
Playback time	10000 ms
Total number of segments	300
Client arrival time	0

Table 5.6: Delay: Test parameters

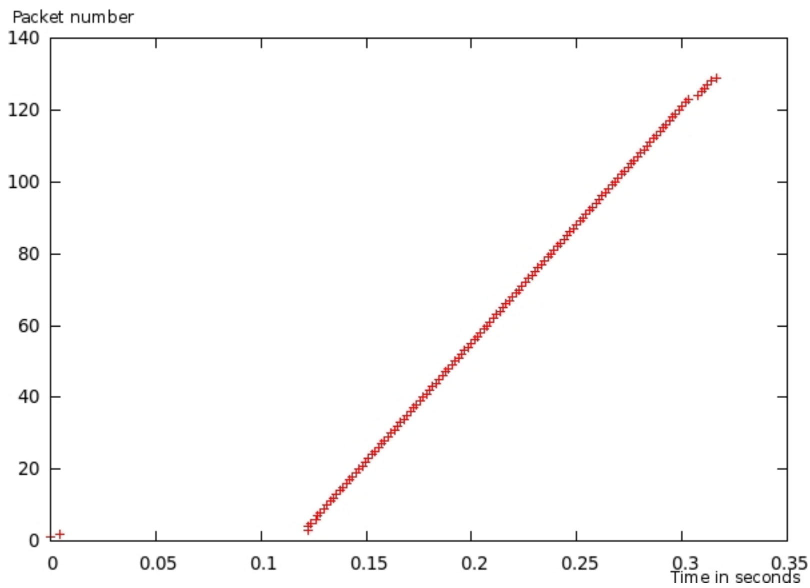
We run three test with a added delay of 0 milliseconds, 50 milliseconds and 185 milliseconds. A maximum delay of 185 milliseconds is used as this is the worst delay we have discovered using ping requests. We do not inflict the network with any packet loss or jitter. We have set the client arrival time to 0 so that every segment is downloaded from the patch servers.



(a) Playback quality

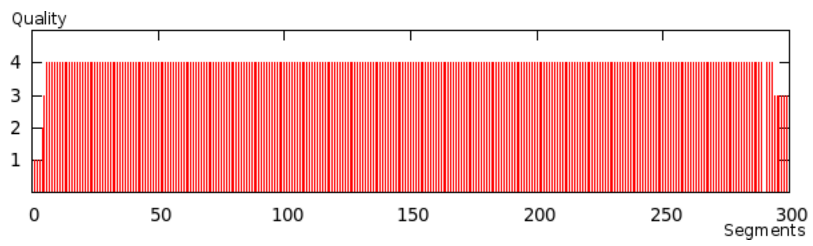


(b) Segments received over TCP in seconds before playout

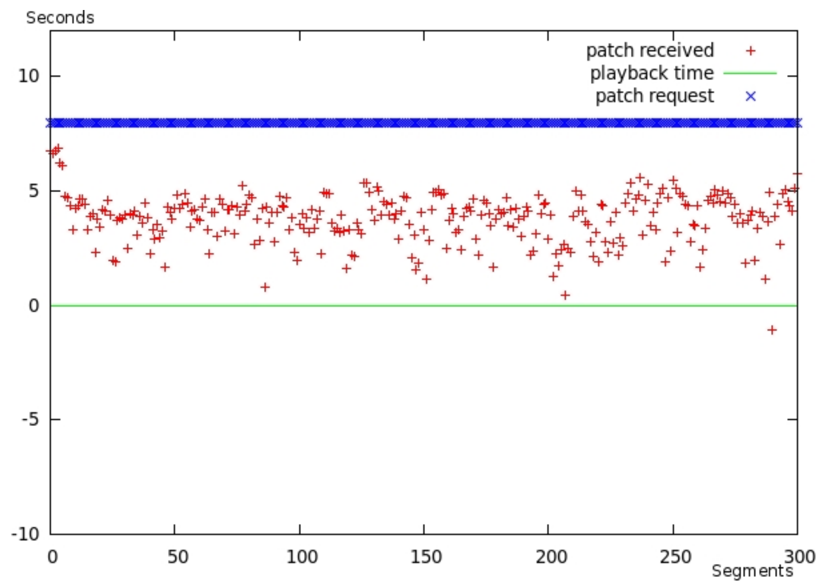


(c) Reception of one segment in lowest quality over TCP

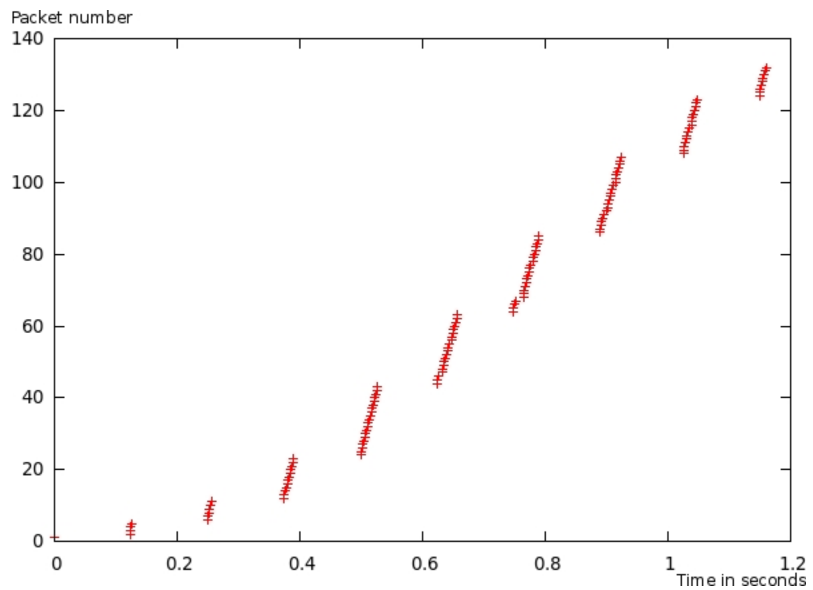
Figure 5.20: Results with 0 milliseconds added delay



(a) Playback quality

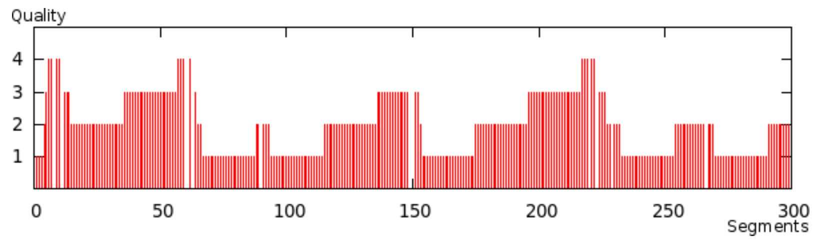


(b) Segments received over TCP in seconds before playout

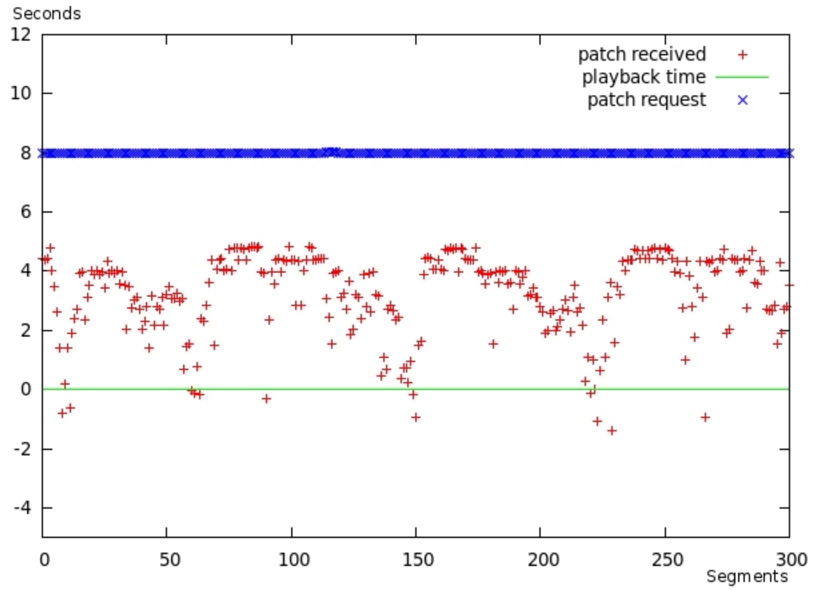


(c) Reception of one segment in lowest quality over TCP

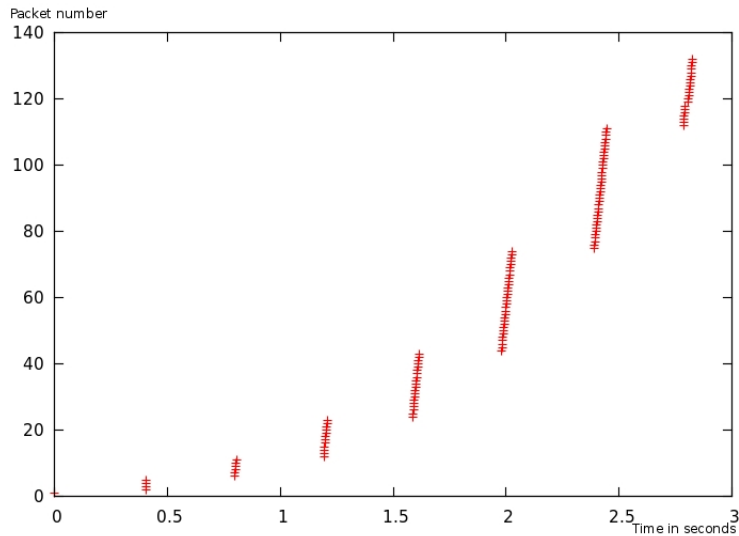
Figure 5.21: Results with 50 milliseconds added delay



(a) Playback quality



(b) Segments received over TCP in seconds before playout



(c) Reception of one segment in lowest quality over TCP

Figure 5.22: Results with 185 milliseconds added delay

As we can see from our result the delay has dramatic results when it comes to delivery time of one segment over TCP. When we increase the delay we can see that it takes longer time from a request for a segment to the download of a segment is complete. The fluctuations in arrival time also vary a lot when delay is increased. There are two reasons for that. The first is packet loss. It takes a longer time to retransmit a packet if the delay is large. The client is downloading the segments from the Internet so packet loss can occur. The second and main reason for the increase in download time of a segment is the slow start strategy of the TCP protocol. TCP uses this strategy to avoid sending more data than the network is capable of transmitting. This is a feature that works well for transmitting large files but when it comes to smaller files, like our video segments, it can have a negative effect of not fully utilizing the available bandwidth, if the delay is large.

Figures 5.20(c), 5.21(c) and 5.22(c) shows how the slow start strategy affects the transmission time of one segment, with round trip times of 0 milliseconds, 100 milliseconds and 370 milliseconds added to the network. The slow start strategy starts off by sending out few packets from the server. How many packets sent is determined by the size of a congestion window. For each round trip time the congestion window is increased if the previous packets all arrived without problems and decreased if, for instance, a packet loss was detected. The maximum size of a congestion window is determined by the bandwidth delay product. The bandwidth delay product (BDP), BDP = bandwidth * round trip time, specifies the maximum amount of data on the network at any time. For our first test it is quite clear that TCP maximize the bandwidth and TCP quickly get a maximum congestion window which gives the client a smooth stream. For our tests with 50 milliseconds and 185 milliseconds added delay, we do not make full use of the bandwidth. It takes the slow start strategy too long time to get a maximum congestion window, so the transmission ends before a maximum congestion window is reached. The BDP for our test with 50 milliseconds delay is 5000 bytes or approximately 33 network packets of 1500 bytes at a time on the network, and 246 packet for our test with 185 milliseconds added delay. We can see that we do not reach those numbers in our two last tests. In our second test, we discovered a packet loss which decreased the congestion window before it was increased again, causing additional time before the congestion window can reach maximum size.

This would not be a problem if the client had a single TCP connection for all segments downloaded from the patch server. The slow start strategy would have a longer time to fully utilize the bandwidth, as the client download all segments within the same TCP connection. This would also improve our implementation in terms of packet loss, as it would be faster to download missing parts of a segment. However, in our implementation, the client makes a new TCP connection to the server for each segment it has to download from the patch servers. The curl library limits us from implementing a solution with a single shared TCP connection for all requests to the patch server. As mentioned in section 4.3, the download function in the curl library does not return until the download of one

request has finished. This limits the client from sending a new request over the same TCP connection before the download of the last request has been completed. If we did not use the curl library in our implementation, the client could have pipeline the requests and received all segments from the patch server over the same TCP connection.

If the client only had one TCP connection for the entire patch stream. Then another problem would be introduced if the client was to receive a multicast stream at the same time. After a while, the congestion window would be so large that it gets a higher sending rate than the multicast stream. In this case the client could risk losing a lot of packets from the multicast stream as the network is filled up with patched segments from our patch servers. To avoid this we should introduce some kind of client side scheduling for downloading a segment from the patch server. The client could request smaller parts of a segment at a time, to get a total download time of approximately two seconds for a segment from the patch server. Then the rate of packets from the patch server would not be that large and it would not become such a large competitor to the multicast stream for network resources.

Summary In this experiment, we have seen the effect delay has on the transmission of patched segments. The client are not able to fully utilize the available bandwidth because of the slow start strategy of TCP. We have proposed a shared TCP socket for all requests to the patch servers as a solution to this problem. This would give TCP longer time to reach a maximum contention window, which would decrease transmission time of a segment. The limitations of the curl library limits us from implementing a solution based on a shared TCP sockets for all requests to the patch servers. We have also discussed the need of scheduling for downloading a segment from the patch server, if the client has one shared TCP socket for all requests to the patch servers.

5.3 Summary

We have in this chapter performed a series of experiments to see how our implementation of a patching scheme would behave in a real network. First we performed an experiment that proved that our implementation of the patching scheme worked. The client was able to receive a multicast stream and a patch stream at the same time. The client started to play segments from its buffer as soon as the patch stream ended.

We moved on to experiment with different bandwidth limitations. The quality adaptation technique enabled the client to receive segments in different qualities under different

bandwidth settings. Whenever the client experienced a deadline miss, it decreased quality and was able to receive segments within the time of playback. The client was later able to increase the quality again, as the patch stream ended and it was only listening to the multicast stream. However, we did see some limitations with our quality adaptation algorithm, as it gave the client long periods of time between each time the client tried to increase quality after a deadline miss. We also suggested that the client should have some knowledge about the available network resources so it never tries to receive segments that is theoretically impossible to receive. It was also suggested that the client should stop downloading a patch for a segment if the segment has missed its deadline. There is no need for waisting network resources on a segment that is not going to be played.

We did experiments on a scenario with packet loss to investigate how this challenge affected our implementation. In addition to giving the client 8 seconds to download patch segments, the client used quality adaptation to be able to receive the segments within the time limit. However, when we increased the rates of packet loss we discovered that it potentially takes a long time to download the missing parts of a segment, no matter which quality the client downloaded. The overall download time of a segment in a lossy environment gets lower when the client reduces quality, as lower quality segments requires less bandwidth. By sending requests for only the missing parts of the multicast segments to the patch servers, the client was able to cope with quite large packet loss rates. However, packet loss is a great challenge for video streaming in the Internet, and it is difficult to avoid deadline misses when the rates of packet losses are increased.

We also discovered that jitter and reordering is not a great challenge for our implementation. By applying a small amount of buffering, the client is able to cope with jitter within reasonable values. Our technique of storing data packets based on byte offset also managed to cope with the challenge of reordering of packets.

At last we performed an experiment with different delay settings. The client only downloaded segments from the patch servers, as we wanted to investigate how delay affects TCP. We discovered that the slow start feature of TCP affects the download time of a segment in addition to the round trip time, as TCP does not fully utilize the available bandwidth when the delay is increased. The segment files are of a size that makes TCP unable to maximize its congestion window, as the transmission of the segment ends before the maximum size of the window is reached. We suggested that using one shared TCP socket for every request to the patch server would solve this problem, as TCP would be given a longer time to maximize it's congestion window.

From the results retrieved from our experiments, we have argued that our implementation of the patching scheme, with suggested improvements and solutions, is able to cope with the challenges of heterogeneous users, packet loss, jitter, reordering and delay, within scenarios that are reasonable to experience in the Internet.

Chapter 6

Conclusion

In this thesis, we have designed and implemented an application for streaming media in a patching scheme, where video in several qualities are made available to the client. Our client discovers which parts of the video it did not receive from the multicast stream and pulls the patch from a patch server. We have performed experiments to investigate how this scheme behaves in a real network. The theory within this field are discussed based on simulations. We therefore saw the need for doing experiments on this topic in a real network to see how it would behave in the Internet and how it copes with the challenges the Internet introduces. Here, we summarize our work and have a look at our most important contributions to this thesis. Finally, we discuss what should be further investigated and presents this as future work.

6.1 Summary and contribution

We have argued that the unicast scheme in today's Internet does not scale well when the number of clients is growing. As it becomes more and more popular to stream media over the Internet, the challenge of avoiding network congestion is getting larger due to the increase in network traffic. Multicast is not made available in the Internet today, but we see this scheme as likely to be introduced in the future as the problem of network traffic is getting difficult to cope with. By using multicast, stream scheduling schemes can be used to reduce network traffic and server load. We have had a look at the properties of different periodic broadcasting schemes and patching schemes. After evaluating the pros and cons of these schemes, we chose to design and implement a patching scheme. This decision is based on the fact that the patching scheme is more friendly to client buffer,

has a shorter client startup time, and it does not occupy as many multicast IP addresses compared with a periodic broadcast scheme. We have also familiarized ourselves with protocols and concepts relevant for this field, that has helped us making decisions during the implementation of our application.

The main contribution to this thesis is our design and implementation of a patching scheme with quality adaptation in a multicast environment and the experiments performed on our implementation. As mentioned in section 1.2, the concept of using a torrent-like HTTP approach for patching and quality adaptation in a multicast scheme has not been implemented or investigated in any other article that we are familiar with. We have used the Live555 library [24] for the implementation of our server and extended the DAVVI media player [20] with functionality for receiving a multicast stream in combination with a patch stream from the web servers. Based on our experiments we have argued that our implementation, with suggested improvements and solutions, copes well with the challenges present in the Internet, within reasonable scenarios. We proved that our implementation of the patching scheme was able to provide the client with segments it missed from the multicast stream.

We have seen that quality adaptation is a good technique when serving video to clients, when clients have different network resources available. However, we discovered that our algorithm for quality adaptation might not be able to provide the client with the best possible quality at all time. This is because the client has a long waiting period between each time it increases quality after a playback deadline miss. We also suggested that the client should have some knowledge about it's own network resources to maximize the potential of quality adaptation.

We have also seen that our implementation of patching is a great technique for repairing damaged segments from the multicast stream in a packet loss scenario as well as requesting whole segments from the patch server. The transmission time of a segment from the patch server to the client increased as we increased packet loss rates. To compensate for the increase in transmission time, the client used quality adaptation to request a patch stream in a lower quality. As the quality was decreased for the patch stream, it was also decreased for the multicast stream. By doing so the client was able to receive whole segments and repairs for segments within time for playback. However, avoiding playback deadline misses became difficult without increasing client buffering, as the rate of packet loss became above 3%.

The challenges of jitter and reordering of data packets are not very large for our implementation and if jitter were to reach very high rates, it can be dealt with by increasing client buffering. We argued that jitter and reordering is more of a challenge for interactive streaming like video conferences and audio chat.

We discovered that the slow start feature of TCP limits us from utilizing the full bandwidth as our video segments are not large enough for TCP to reach a maximum contention window before the transmission of one segment is complete. This increases the transmission time of a segment with our implementation, since the client makes a new TCP connection for each request to the patch servers. We discussed the possibility of using only one shared socket for all requests to the patch server and the effect this might have on our reception of segments.

By comparing our results to the articles about stream scheduling schemes, we can see that the challenges present in the Internet do affect these schemes. The articles have presented designs and implementations of these schemes, and performed simulations to see that the concept works, and what resource requirements these schemes require from the server, the client and the network. We are not familiar with any simulations that presents how these schemes are affected by challenges in the network. The authors of these articles have not considered the challenges we have experimented with in this thesis. Articles have proposed solutions to some of these challenges, but we have not seen any experiments in a real network with these solutions. However, we have seen from the results of our experiments that precautions on the client side are essential to overcome the challenges in the Internet.

Based on the experiments performed on our implementation in this thesis, we conclude that a patching scheme is feasible within today's infrastructure in the Internet, if the ISPs switch on multicast, as it limits server load, network traffic and is, to some extent, able to cope with the challenges present in the Internet. However, the technique of quality adaptation and client buffering is essential to camouflage the challenges of client bandwidth heterogeneity and packet loss. Jitter is, as mentioned, not a great challenge for this scheme. Neither is normal values of delay, if the suggested solutions for the problem with TCP slow start is implemented.

6.2 Future work

We have in this theses seen how patching can be used to serve clients joining an ongoing multicast stream, as well as repairing damaged segments due to packet loss. It would be interesting to investigate the possibilities this patching scheme can give a client when it comes to trick play functionality, i.e., interactivity like jumping back and forth, seeking, etc. The client could utilize multicast patch streams to be able to seek within the video. There are several possibilities on how this can be done. For instance, the server could generate a new multicast stream if a client wanted to seek forward in the video, giving the other clients that are receiving the original multicast stream, two multicast streams

to receive. The clients receive one stream for instant consumption and one stream to be buffered for later playback. If a lot of clients wants to seek forward in the movie then a client would potentially get many multicast streams to receive at the same time.

We experimented on how the effect of delay were on a TCP stream, and we concluded that delay has a massive effect due to the slow start strategy of TCP. We also discussed a solution for only using one shared TCP socket for all requests to the patch servers, as this would give TCP a longer time to reach a maximum size of the congestion window and fully utilize the bandwidth. The effect this might have on the multicast stream is not investigated, and we made an assumption that the transmission of the patched data would occupy the network causing packets from the multicast stream to be dropped. We then proposed the need for a scheduling mechanism on client side to limit the download rate of the patch stream so that downloading a segment would take approximately two seconds and not steal all the available bandwidth from the multicast stream. It would be interesting to investigate different scheduling algorithms for this purpose and perform experiments in the Internet to see the real effects.

Another interesting aspect of the patching scheme is how it can be combined with a harmonic broadcasting scheme. The harmonic broadcasting scheme are able to cope with clients making asynchronous requests for a video, but it would be interesting to see how we could use patching to repair the video, in case of packet loss and how that would affect the bandwidth.

In this thesis we have implemented an algorithm for quality adaptation. In our experiments we discovered that this algorithm was at times not able to fully utilize the client bandwidth. We proposed a shorter waiting time for each time the client tries to increase the quality, as well as giving the client knowledge about its network connection. It would be interesting to further develop the quality adaptation algorithm, in terms of these improvements, and investigate how this affects the reception of a video stream in the Internet.

Bibliography

[1] K. Almeroth. The evolution of multicast: from the mbone to interdomain multicast to internet2 deployment. *Network, IEEE*, 14(1):pages 10–20, jan/feb 2000.

[2] K. Almeroth and M. Ammar. The use of multicast delivery to provide a scalable and interactive video-on-demand service. *Selected Areas in Communications, IEEE Journal on*, 14(6):pages 1110 –1122, aug 1996.

[3] Apple. Apple http live. http://tools.ietf.org/html/draft-pantos-http-live-streaming-03, Accessed: April 2010.

[4] H. Bettahar. Tutorial on multicast video streaming techniques. *3rd International Conference: Sciences of Electronic, Technologies of Information and Telecommunications*, March 27-31, 2005.

[5] S. Bhattacharyya. Rfc: 3569 - an overview of source-specific multicast (ssm), July 2003.

[6] boost. boost c++ libraries. http://www.boost.org, Accessed: Jan, 2010.

[7] M. K. Bradshaw, B. Wang, S. Sen, L. Gao, J. Kurose, P. Shenoy, and D. Towsley. Periodic broadcast and patching sercices - implementation, measurement, and analysis in an internet streaming video testbed. In *In Proceedings of the 9th ACM int. confecence on Multimedia*, pages 280–290, September, 2001.

[8] S. W. Carter, D. D. E. Long, and J.-F. Paris. Video-on-demand broadcasting protocols. In *Multimedia communications: directions and innovations*, pages 179–189, Orlando, FL, USA, 2001. Academic Press, Inc.

[9] curl.haxx.se. curl. http://curl.haxx.se, Accessed: Feb, 2010.

[10] W. Fenner. Rfc: 2236 - internet group management protocol, version 2, 1997.

[11] T. L. Foundation. Netem. http://www.linuxfoundation.org/collaborate/ workgroups/networking/netem, Accessed: March, 2010.

[12] T. J. Fredrikson. Scheduling of data streams over a multicast protocol. *Master thesis, Simula Research Laboratory and Universitetet i Oslo Institutt for Informatikk*, December, 2008.

[13] C. Griwodz and P. Halvorsen. Inf5071: Performance in distributed systems, distribution part1. http://www.uio.no/studier/emner/matnat/ifi/INF5071/h08/undervisningsmateriale/06-distribution-1.pdf, Accessed: May, 2009.

[14] C. Griwodz, M. Liepert, M. Zink, and R. Steinmetz. Tune to lambda patching. *SIGMETRICS Perform. Eval. Rev.*, 27(4):pages 20–26, 2000.

[15] T. Gudmundsen. Periodic broadcasting protocol - implementation and measurements. *Master thesis, Simula Research Laboratory and Universitetet i Oslo Institutt for Informatikk*, May, 2009.

[16] A. R. H. Schulzrinne and R. Lanphier. Rfc: 2326 - real time streaming protocol (rtsp), 1998.

[17] R. F. V. J. H. Schulzrinne, S. Casner. Rfc: 3550 - rtp: A transport protocol for real-time applications, July 2003.

[18] K. A. Hua, Y. Cai, and S. Sheu. Patching: a multicast technique for true video-on-demand services. In *MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia*, pages 191–200, 1998.

[19] U. o. S. C. Information Sciences Institute. Rfc: 793 - transmission control protocol, Sep 1981.

[20] D. Johansen, H. Johansen, T. Aarflot, J. Hurley, A. Kvalnes, C. Gurrin, S. Zav, B. Olstad, E. Aaberg, T. Endestad, H. Riiser, C. Griwidz, and P. Halvorsen. Davvi: a prototype for the next generation multimedia entertainment platform. In *MM '09: Proceedings of the seventeen ACM international conference on Multimedia*, pages 989–990, 2009.

[21] A. Keller. Manual tc packet filtering and netem. *ETH Zurich*, July 2006.

[22] C. Krasic, J. Walpole, and W.-c. Feng. Quality-adaptive media streaming by priority drop. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 112 –121, 2003.

[23] L. Lao, J.-H. Cui, M. Gerla, and D. Maggiorini. A comparative study of multicast protocols: top, bottom, or in the middle? In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 4, pages 2809–2814, 2005.

[24] L. N. I. (LIVE555.COM). Live555 streaming media. http://live555.com, Accessed: May 2009.

[25] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 117–130. ACM, 1996.

[26] T. M. Messenger. The mongol messenger. http://www.mongolmessenger.mn, Accessed: April 2010.

[27] Microsoft. Smooth streaming. http://www.smoothhd.com, Accessed: April 2010.

[28] Microsoft. Windows media. http://www.microsoft.com/windows/windowsmedia/default.mspx, Accessed: April 2010.

[29] M. Networks. Move. http://www.movenetworks.com, Accessed: April 2010.

[30] J. Postel. Rfc: 768 - user datagram protocol, Aug 1980.

[31] J. M. H. F. L. M. P. L. R. Fielding, J. Gettys and T. Berners-Lee. Rfc: 2616 - hypertext transfer protocol – http/1.1, June 1999.

[32] Real. Real media. http://eu.real.com/realplayer/, Accessed: April 2010.

[33] T. S. M. W. S. Wenger, M.M. Hannuksela and D. Singer. Rfc: 3984 - rtp payload format for h.264 video, Feb 2005.

[34] J. R. Subhabrata Sen, Lixin Gao and D. Towsley. Optimal patching schemes for efficient multimedia streaming. *International Conference on NOSSDAV*, June, 1999.

[35] A. S. Tanenbaum. *Computer Networks*. Pearson Education International, fourth edition edition, Aug 2002.

[36] T. X. Team. Xorp. http://xorp.org/, Accessed: March 2010.

[37] B. Wang, J. Kurose, P. Shenoy, and D. Towsley. Multimedia streaming via tcp: An analytic performance study. *ACM Trans. Multimedia Comput. Commun. Appl.*, 4(2):pages 1–22, 2008.