UNIVERSITY OF OSLO
Department of Informatics

Scheduling of Data Streams over a Multicast Protocol

Masteroppgave

Tonje Jystad Fredrikson

December 1, 2008

Scheduling of Data Streams over a Multicast Protocol

Tonje Jystad Fredrikson

December 1, 2008

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my gratitude to my advisers, Carsten Griwodz, Pål Halvorsen and Håvard Espeland, for their guidance, valuable feedback, and helpful attitude.

I would also like to thank Håkon Stensland for his continued encouragement throughout my studies, leading up to this thesis. Along with Paul Beskow he has been a great help in providing feedback on my writing. I also wan to thank Alexander Eichorn for helpful discussion, and Pengpeng Ni for help with PSNR and SSIM.

I would also like to thank all the guys in the lab. I have truly appreciated the motivating work environment, good conversations, and our many laughs.

I would also like to thank my parents for continued support, and my friends for encouragements along the way.

Tonje Jystad Fredrikson

Fornebu, December 1, 2008

Chapter 1

Introduction

1.1 Background

Networked multimedia refers to the transmission of video, audio, and other interactive content like online games. A common denominator is that end users are given immediate access to the stored media upon request. Growing popularity for networked multimedia has been seen over the last years, and as a result we see new media services arise. Examples span from video streaming on `www.youtube.com`, games like World of Warcraft [4], to Internet based TV and radio as seen on `www.nrk.no/nett-tv/` and `www.radio1.no`. In this thesis we will focus on a subset of networked multimedia, namely Video-on-Demand (VoD).

Transmission of video represents a range of challenges as large amounts of data are delivered over the network. End users have high demands with regard to delivery time and sometimes expect the system to respond instantaneously. On the server side the processing performance and ability to meet client requests is vital. The network needs to be able to handle the transmissions, whether the challenge is bandwidth, delay, or routing. The client needs enough bandwidth to download the video, as well as processing power to receive, decode and display the streams. Depending on the transmission model, the client will possibly need buffering capabilities, something that will be discussed in later chapters. There has been active research on trading off resource use at servers, clients, and on intermediate systems such as proxies, and in the network itself. This work has, however, been largely theoretical due to shortcomings of the deployed infrastructure, many of which are no longer an obstacle.

As the network infrastructure has improved, we now find it interesting to revisit the research areas for transmission of VoD. Some focus on how to differentiate the delivery to the client based on the end users needs. Another area of research focuses on mechanisms for scheduling the media streams that are transmitted. Using scheduling techniques the VoD providers seek to avoid consuming more network resources than necessary. Stream scheduling mechanisms attempt to deliver content rapidly following an individual user's request so that the system appears to answer immediately. They also seek to provide individual service, while handling requests from as many users as possible and still reduce the resource consumption on servers and in the distribution system and network.

Some techniques assume broadcast infrastructures like TV broadcast over antenna. Using these ideas on the Internet is difficult, because TV broadcast guarantees bandwidth and delay, while the Internet does not. Other techniques use Internet multicast, but multicast has not been supported by Internet Service Providers (ISPs). Good payment models in conjunction with multicast is one problem area, as multicast addresses can be listened to by anyone in the autonomous network. Potential for Distributed Denial of Service attacks is another challenge. Communication across different ISPs was yet another challenge, as some do not agree in policies and what services should be accessible for their customers. The increasing demand on processing in the network could result in a need for more complex routers and higher costs for the ISPs. The number of end users with enough bandwidth to actually make good use of VoD transmissions has only increased rapidly in the later years. Prior to the upgrade of the network infrastructure it was impossible for ISPs and researchers to actually implement broadcast and multicast outside of testbeds, such as it is today.

Existing service providers (such as www.sf-anytime.com, fxt.no) presently offer video-on-demand, and they are now noticing that the lack of multicast restricts the number of customers that they can serve concurrently. ISPs are therefore upgrading their networks to support multicast, and open it at least for their own on-demand applications. By using multicast in combination with scheduling techniques using several channels, as well as buffering at the client side, the server and network resources can be significantly reduced. There is a trade off between the reduced use of resources on the server side and on the requirements on the client side. The end users will in most cases have to wait for a short time before the movie can actually be watched. In addition, the client will meet a higher requirement for bandwidth, as well as need for buffering; both challenges that with today's technology have feasible solutions.

Consequently, ISPs are currently interested in stream scheduling techniques. As mentioned, most of them are purely theoretical and make assumptions about quality of service in the underlying network. Such assumptions include constant transmission, latency, delivery rates and bandwidth, as well as a lossless environment. In this thesis, we revisit one of the techniques, Cautious Harmonic Broadcasting, used together with Real Time Streaming Protocol and Real-time Transport Protocol, to test and evaluate these in our environment.

1.2 Problem statement

Stream scheduling techniques show a great potential for reducing server load and improve resource utilization for distribution of highly popular media files. In the 90s, when the first of these techniques were proposed, it was not feasible to put them into practice because of various reasons, including those mentioned in the background (see 1.1). Consequently the techniques were therefore evaluated analytically or by using simulations. However, as the computing technology has improved the situation today is different.

In this thesis, we therefore seek to develop an implementation based on the concepts of periodic broadcasting. A periodic broadcast server divides a video object into multiple disjoint segments, and broadcasts these segments over a set of multicast addresses. Harmonic Broadcasting is a scheduling technique that makes use of periodic broadcasting, and it has shown promising results in simulations. Focusing on Harmonic Broadcasting, we will put periodic broadcasting into practice using an existing streaming framework, namely live555 [3]. Live555 is a streaming library currently used in streaming applications like VLC (VideoLAN Project [5]) and MPlayer (MPlayer team [6]). We will look at design issues for how the scheduling technique can be implemented. Finally, we will evaluate the performance of the implementation to determine how well it works in an emulated system, as well as expose challenges and suggest possible solutions.

1.3 Main contributions

In this thesis we have investigated a selection of promising scheduling techniques for periodic broadcasting. As ISPs are upgrading their networks to support multicast, they see the opportunity to use less resources to offer their customers quality VoD by making use of efficient scheduling techniques. Of the techniques studied, we found the Cautious Harmonic Broadcasting (CHB) algorithm to be most promising.

CHB is designed to schedule video streams over a multicast network. Reduced transmission bandwidth is one of the benefits of using this algorithm. Our most important contribution was designing a CHB streaming server and client, and implementing a prototype of each. Our server prototype has successfully put the CHB algorithm into practice.

In our implementation we used the live555 framework, enabling future integration with open source media players like VLC and MPlayer. In our assessment of the prototypes we have performed a series of tests to evaluate the performance. We suggest that conflicting use of variable and constant bit rates can be one factor that can interfere with the CHB scheduling. We also observed that the temporally aligned scheduling scheme in CHB can result in duplicate or lost video frames.

From our work we can conclude that it is feasible with today's technology to put a CHB streaming server into operation for a number of subscribers.

1.4 Structure

This thesis is organized as follows; An introduction to terminology and concepts within scheduling techniques is presented in chapter 2. We also give a background on different scheduling theories, including Harmonic Broadcasting. There are existing implementations to a couple of the scheduling techniques. In chapter 3, we give a summary of articles that describe some of them, and discuss their results. To make use of the underlying network, scheduling techniques must make use of transmission protocols. Chapter 4, gives an introduction to common streaming protocols, as well as a common standard for video compression. In chapter 5, we look at design challenges for implementing a periodic broadcasting algorithm, the Harmonic Broadcasting protocol. We then move on to chapter 6, where we explain the actual implementation. This includes an introduction to the live555 streaming library that has been used. We continue to evaluate the performance of the implementation in chapter 7, and discuss our findings further in chapter 8. The discussion leads up to our conclusion in chapter 9 with suggestion for further work.

Chapter 2

Scheduling techniques

In this chapter, we will take a look at some scheduling techniques for VoD. VoD consumes lot of the resources in the network. Even a small video with resolution of 512 x 288 pixels will require a bandwidth of 1100 kbps from end to end. With technologies like High Definition (HD) that has higher resolution than normal TV, the demands for bandwidth can seem likely to increase. The following techniques seek to optimize the resources and bring them down to a minimum to lower the requirements to the underlying network, the client, and the server, as well as costs.

The Harmonic Broadcasting technique will be examined in more detail as it is the one we consider most promising with regards to use of resources, particularly with regard to bandwidth. Before we look at the different scheduling techniques, we will have a short introduction to terminology and concepts that are recurring throughout the chapter. We will also briefly look at what some of the costs related to these techniques are.

2.1 Terminology and concepts

2.1.1 Terminology

Unicast is when a packet is sent from a source to a single client.

Multicast When sending is directed only to the clients who have specifically requested membership in a group, we call it multicast, as there can be multiple recipients of the packets transmitted. Packets are only routed through the span tree created by group members.

Broadcast Sending packets to all clients in a network is referred to as broadcasting. It is, however, up to the client to choose to listen and receive the packets or not.

Stream: Continuous transmission of data.

Channel: A stream of data on the VoD server (Carter et. al 2001 [7]). The resources needed for continuous delivery of a single stream. (Dan et. al, 1993 [8]).

Segment: A movie can be divided into several non-overlapping parts called segments.

Consumption rate: The rate at which a client consumes the media (for example watches a video).

Wait time The time gap between a client requests a video until it is ready for consumption is called 'wait time'.

Trick Play: Functions like Play, Pause, Stop, Forward and Rewind are referred to as Trick Play functions.

2.1.2 Concepts

In "Prospects for Interactive Video on Demand," Little and Venkatesh [9] classify interactive services into several categories based on their level of interactivity. They are:

Broadcast has no VoD. Like with television the end user has no control over the session.

Pay-per-view exists today, and requires the client to sign up and pay for a specific session. Finding good ways to ensure billing is a current challenge.

Quasi VoD (Q-VoD) requires the users to be grouped. Users can then perform some Trick Play functions by switching groups.

Near VoD (N-VoD) Some Trick Play functions like forward and reverse are available to the end user. The capability is provided by multiple channels with the same program skewed in time.

True VoD gives the user full control over the session presentation, including Trick Play functions.

2.1.3 Costs related to scheduling techniques for Video-on-Demand

Recurring in this chapter will be references to various costs related to scheduling techniques for Video-on-Demand. For a better understanding we will give a brief introduction to some of these. Costs in our context refers to not only monetary value. It includes resources used as well as complexity in implementations and other factors that influence the quality and success of an offered VoD service.

Wait time Wait time can determine whether a client decides to use a service or not. If the waiting time exceeds what the client is willing to wait, the number of users of the service will decrease, and lessen the value of offering the service.

Delay Delay from when a packet is sent until it is received at the client side will add to a users wait time. Constant delay for all packets will not affect the quality of the received video. However, delay occurring at random points can affect the viewing experience if the packets do not reach the client in time. It will have the same effect as packet loss. Network congestion is once cause of such delay, interference in wireless networks another one. Rebooting of a router is yet another cause.

Packet loss Video is very sensitive to packet loss. The loss of vital frames in the video can cause flickering when viewing the movie, and in worst case the loss of either sound and/or images.

Buffering Some scheduling techniques will transmit data prior to client consumption. This requires local storage to buffer the data until the time of consumption. The main buffering can take place either at the server, a proxy server or a client. The buffered data can be either in memory, or written to disk. Monetary costs will depend on type of storage medium in use. Buffering can on the other hand prevent negative effects from delay.

Scalability A scheduling technique that is said to scale well can transmit video to an increasing number of users, while having the costs on the server and network increase at a much lower rate. Good scalability will make the costs per client decrease as more clients use the service.

Bandwidth Video itself requires a substantial amount of bandwidth in comparison to other services like downloading web pages or just audio. The cables carrying the data signals need to have enough bandwidth to transmit the video from end to end. Unicast will transmit one separate stream to each client. For highly popular videos, this will cause high stress on the network as well as the streaming server. Broadcast will transmit to everybody, and hence take up unnecessary bandwidth to those who do not wish to receive the data in question.

Network costs The network itself depends on routers to forward packages to the end user. With increasing amount of data these routers need to be able to handle an increasing amount of data.

Application complexity When implementing a scheduling technique, the complexity will directly affect the costs for the development of application. The more complex the algorithm, the longer it takes for a programmer to code, and the costlier is the maintenance and upgrading of the application.

In the following sections we will look at three main mechanisms: Delayed On-Demand Delivery, Prescheduled Delivery, and Client-Side Caching. We will describe these and give examples of scheduling techniques that represent each of them, and discuss their performance.

2.2 Delayed On Demand Delivery

Delayed on Demand is based on true on demand delivery. However, it is delayed because it will not transmit the data until several clients have requested the same file. This enables more efficient delivery than when sending unicast to each individual client.

2.2.1 Batching

When multiple clients requests for the same movie arrive within a short time frame, they can be batched together and serviced by the same multicast stream [8]. The wait time between each single stream, the bathcing window, will directly affect the server capacity. There is a trade-off between decreasing server capacity and increasing wait time. Batching has the benefits of being able to service popular media different than less popular media. The technique is N-VoD as Trick Play is to some extent possible by changing the stream a client listens to. For example, to rewind, or pause a client can jump to the next stream that has been activated. If no new stream is available, a solution if to create a new batch. To fast forward, the client can likewise switch to the preceding stream. However, the availability of Trick Play does rely on the underlying batching policies.

The bathcing can be done purely by number of clients requesting the same video, by a predefined schedule, or a maximum wait time before multicasting. Drawbacks include the wait time, as well as limited saving with regard to bandwidth in comparison with other scheduling techniques. Careful selection of which policy to use will be important for any service provider using batching. Three main objectives are to reduce the average wait time for the clients, avoid clients canceling their requests, and to be fair to all requests; irrespective of popularity.

2.2.2 Adaptive Piggybacking (Stream Merging)

Adaptive piggybacking seeks to reduce wait time for servicing new requests for VoD, and is used in conjunction with batching. By changing the consumption rate for video streams of the same video, the streams can later be merged. [10]. By reducing the consumption rate of a movie for an initial stream, and increasing it for the stream following, the two streams can be merged into one when they eventually reach the same video frame. After the merging the consumption rate can be set to normal. Golubchik et. al [10] established that a deviation of at most 5% from the normal consumption rate will not be perceived by the viewer. They concluded that even small variations in the delivery rate can be enough to decrease the bandwidth significantly.

Adaptive Piggybacking can be categorized as a Q-VoD technique as the clients do not have full control, but can have access to limited Trick Play by changing group and stream. It has a great benefit over pure batching in that the wait time is reduced. It is also possible to serve a client immediately if desired, at the cost of increased bandwidth. Variation in the delivery rate can also make already merged streams merge with new streams added at a later point, decreasing the bandwidth even more. It still requires several streams, at almost basically full bandwidth, containing the same content, being transmitted over the network.

2.3 Prescheduled Delivery

Presided delivery mechanisms will already be running the video at the time of the client's request. For example, a video will be transmitted repeatedly to a multicast address, where clients need to start listening when the video has completed an iteration and reached back to start. The client will perceive the service as on demand, whereas in truth it has been scheduled in advance. Prescheduled deliveries will generally require segmentation of the video, and buffering and reordering on the client side.

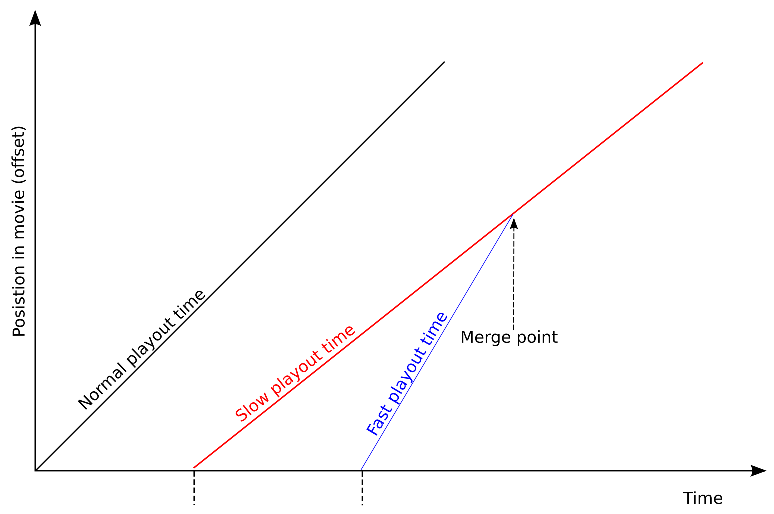

Figure 2.1: *Adaptive Piggybacking*

2.3.1 Staggered Broadcasting

Staggered Broadcasting is a technique that is based on transmitting a video on several channels by staggering the starting time for the video evenly across the channels [11]. Each channel transmits the video at the required bandwidth, so the needed bandwidth on the server side increases linearly as a product of number of channels. The difference in starting time between each channel is the phase-offset. The waiting time for a client will then be equal to or less than the phase-offset time. In other words, a provider may adjust the phase-offset according to popularity of a video to provide a better (though perhaps unfair) service. A popular movie can have small segments, so the phase offset is smaller, possible starting points more frequent. This model provides limited Trick Play functions where forward can be done by jumping to a channel one phase-offset earlier, or pause and rewind by jumping to a later one. The drawback of staggered broadcasting is the high need for bandwidth as the phase-offset gets smaller. It is, however, one of the techniques that has been put into real life use. It can be implemented both for broadcasting and multicasting [11]. The benefit of multicasting over broadcasting is that the stream is only transmitted via routers with client members of the multicast group. In other words, with no members, mainly server resources add to the cost given that the network provides multicast routing.

2.3.2 Pyramid Broadcasting

Unlike Staggered broadcasting, where the segments of the video are same length, it is possible to use variable length segments. This idea was introduced by by Viswanathan

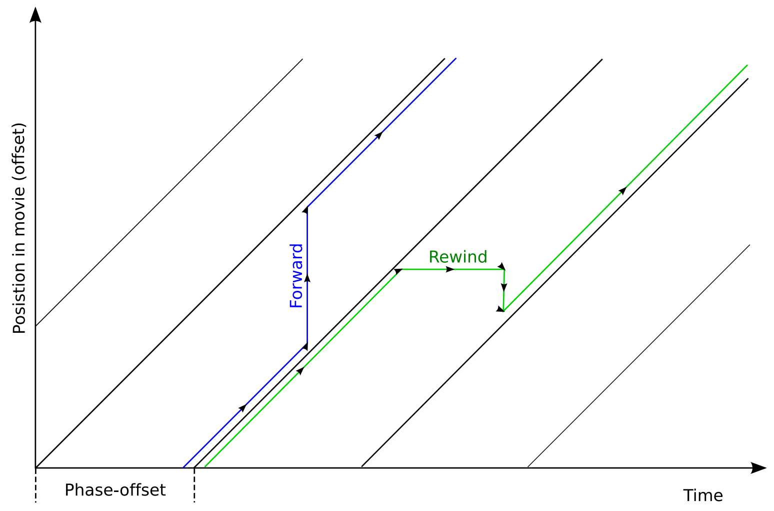

Figure 2.2: *Staggered Broadcasting*

and Imielinski in 1996 [12], and is called Pyramid broadcasting. Pyramid broadcasting multiplexes a video on the channels in a similar way as Staggered broadcasting. However, the movie segments are broken into segments of increasing sizes (instead of equal). The name Pyramid was derived from the notion that stacking the segments on top of each other (with the first, and shortest segment on top) the segments would take the shape of a pyramid. The smallest segment (the first part of the movie) is broadcast most frequently. While the first segment is consumed, the second one can be downloaded. As a result, Pyramid scheduling requires the end user to buffer the data upon reception. Pyramid Broadcasting requires that the number of high bandwidth channels

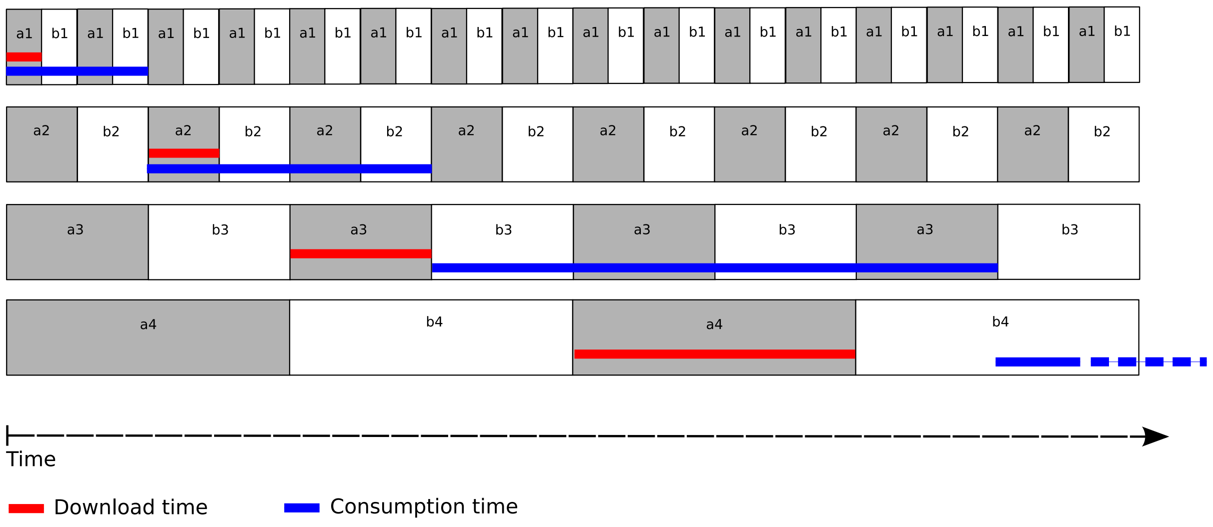

Figure 2.3: *Pyramid Broadcasting for movie* ***a*** *and* **b**, *where the numbers indicate segment number. The red lines indicate the time of download of the indicated segment for movie* ***a***, *whereas the blue line shows the client's consumption time of the same segment (also from movie* ***a****).*

is predefined and constant for a given server. If a video is divided into n segments, then there will also be n equal-bandwidth channels. As a result, the movie segment can be broadcast at a higher bandwidth than the movie is actually consumed. Where staggered broadcasting only uses a channel for one movie segment, pyramid allows for segments from multiple movies to be broadcast interleaved using the same channels. See figure 2.3 for a graphical model of the play out of 2 movies using 4 channels.

Drawbacks with Pyramid Broadcasting are that the clients need to buffer more than 50% of the desired video, in addition to receiving all channels concurrently in a worst case scenario. In figure 2.3, this can be seen as all of segment a_4 is buffered prior to consumption. The bandwidth is a direct multiple of number of segments, and will only work for clients in high bandwidth networks.

2.3.3 Skyscraper Broadcasting

Where Pyramid broadcasting took its name from the shape of how the segments would look stacked, Skyscraper broadcasting takes its name from how equally sized segments would be placed over several channels in a steep, skyscraper-like shape. In "Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan Video-on-Demand Systems" Hua and Sheu lay out a scheduling scheme where a movie is divided into n equally sized segments, and the following pattern determines how many segments to be allocated on each channel:

```
1, 2, 2, 5, 5, 12, 12, 25, 25, 52, 52, ...
```

Each channels has the same bandwidth, and must be equal to the consumption rate of the video. The client receives at most two channels at a time while buffering at most two segments.

2.3.4 Harmonic Broadcasting

Harmonic Broadcasting was introduced by Juhn and Tseng [13]. Their theory breaks a video into n equally-sized segments S, and dedicates n streams for broadcasting the different segments of the video. Each stream i repeatedly shows segment S_i with bandwidth $\frac{b}{i}$, where b is the consumption rate of the video. However, in "Efficient broadcasting protocols for video on demand" Paris, Carter and Long [2] prove that this protocol will not always deliver all data on time. It requires that the client waits

Segments	**1**	**2**	**2**	**5**	**5**	**12**	**12**
						27	39
						26	38
Segment						25	37
						24	36
numbers						23	35
						22	34
in						21	33
				10	15	20	32
channel				9	14	19	31
				8	13	18	30
		3	5	7	12	17	29
	1	2	4	6	11	16	28
Channel	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Table 2.1: *The table shows how the segments in Skyscraper broadcasting are distributed to their respective channels in the a shape of an increasing steep 'skyscraper'. This table only shows up to 12 segment for one channel. The next channel with 25 segment, would underline the steep increase in number of channels per segment.*

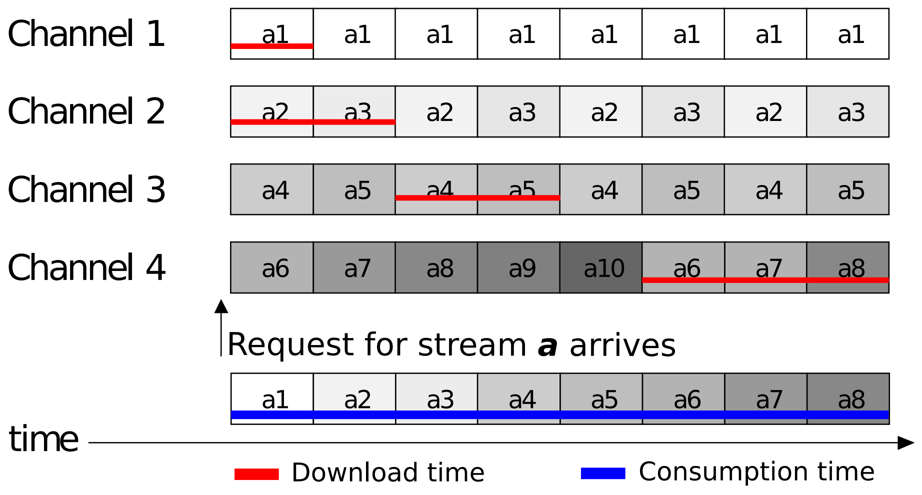

Figure 2.4: *Skyscraper Broadcasting for movie **a**. where the numbers indicate the segment numbers. The red line indicates the time of download, whereas the time of consumption at the client end is indicated by the bottom row. (Figure adapted from Griwodz and Halvorsen, 2008 [1])*

for $(n-1)d/n$ units of time. As a result, Paris et al [2] have proposed two variations to the protocol, described below, that address this issue. These protocols ensure that all frames will always arrive on time and the client will never have to wait any extra time after the beginning of an instance of segment S_i.

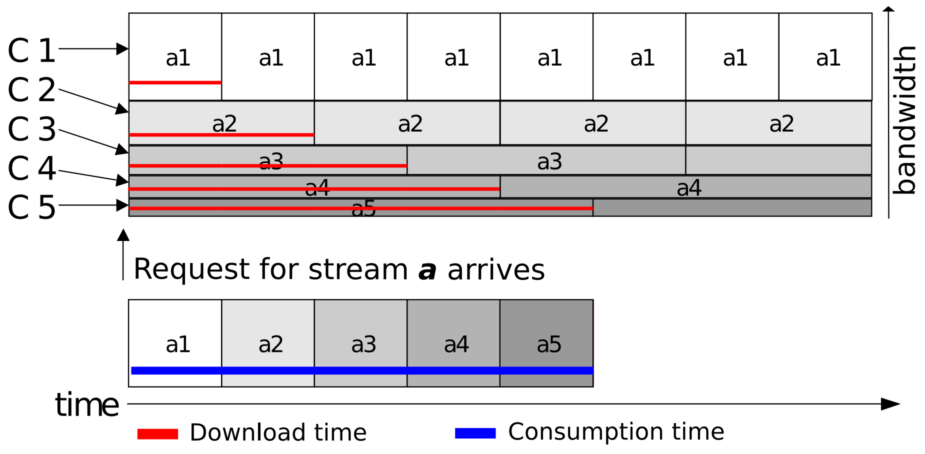

Figure 2.5: *Harmonic Broadcasting. (Figure adapted from Griwodz and Halvorsen, 2008 [1])*

Cautious Harmonic Broadcasting (CHB)

Cautious Harmonic Broadcasting solves the problem by sending the first 3 segments at full bandwidth (alternating S_2 and S_3), and then transmitting S_4 to S_n at decreasing bandwidths ($b_i = \frac{b}{i}$ for $i = 3, ..., n - 1$). As a consequence the first three segments must all be delivered on time as they are broadcast with full bandwidth.

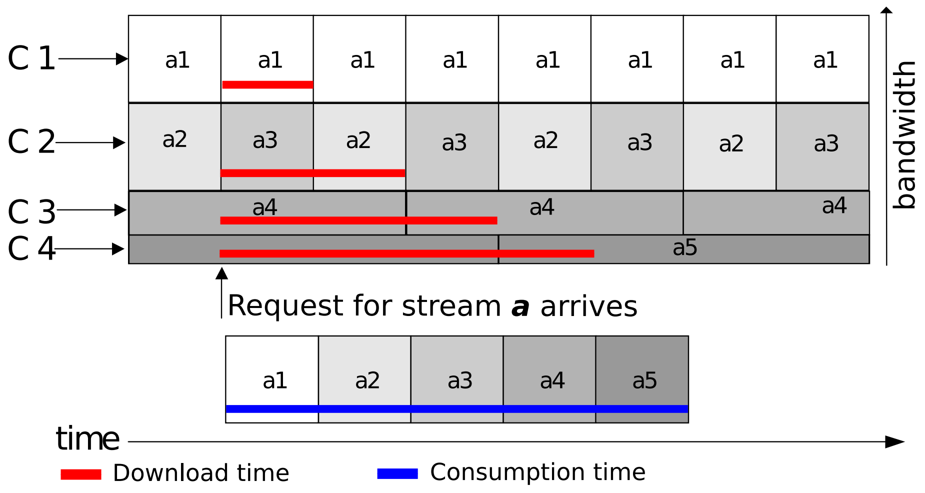

Figure 2.6: *Cautious Harmonic Broadcasting Algorithm, transmitting movie **a** and its corresponding segment number. (Figure adapted from Griwodz and Halvorsen, 2008 [1])*

Quasi-Harmonic Broadcasting (QHB)

If we allow the client to consume data from a segment while it is still receiving data for that segment, we can improve upon the CHB protocol.

Like Harmonic Broadcasting, CHB broadcasts the first segment at full bandwidth. For each segment i, for $1 < i \leq n$, i is broken into $im - 1$ fragments for some parameter m. and the client will receive m fragments from each channel per time slot. If each time slot is divided into m equally sized subslots, then the client will receive one fragment during each subslot. The fragments are transmitted in the following order:

- In channel i, the last subslot of each time slot is used to transmit the first $i - 1$ fragments of S_i.
- The rest of the subslots transmit the other $i(m - 1)$ fragments such that the k^{th} subslot of slot j is used to transmit fragment $(ik + j - 1) \bmod i(m - 1) + i$ (see figure 2.7).

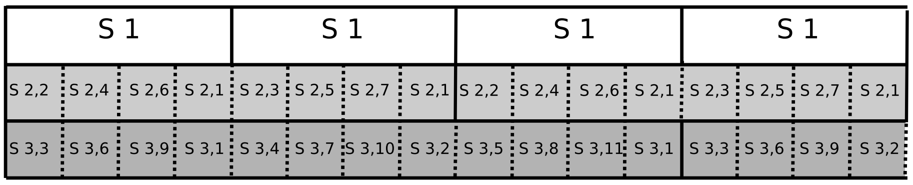

Figure 2.7: *Quasi-Harmonic Broadcasting Algorithm, transmitting segments S, where the notation $S_{2,4}$ refers to segment 2 and its sub-segment 4.*

2.4 Client Side Caching

Client Side Caching is, like the name indicates, based on buffering at the client, while offering the service on demand. As a consequence they require a considerable amount of client resources. We will describe one, and very briefly introduce a second.

2.4.1 Patching (Stream Tapping)

Consider a movie that is multicast on a channel. A client who wants to watch the movie requests the movie n minutes after the initiation. Instead of having to wait for the next iteration of the movie, the server can transmit the n minutes of the movie by unicast, and at the same time let the client listen to the multicast channel (see figure 2.8). By continuously buffering the next $n$minutes on the multicast channel the client is able to receive the movie as True VoD. The n minutes of the movie that was unicast to the client is referred to as a 'patch' for the missing portion of the movie at the time the client requests to watch. The length of the patch will be dependent on the size between the initial and following multicast stream, as seen in figure 2.9 In "Patching:

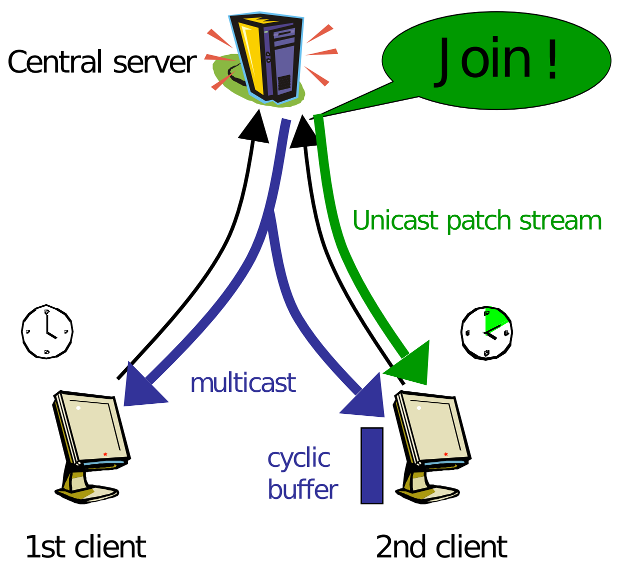

Figure 2.8: *Patching. A client buffers buffers the multicast stream transmitted during the time the client receives the patch as unicast. Figure adapted from Griwodz and Halvorsen [1]*

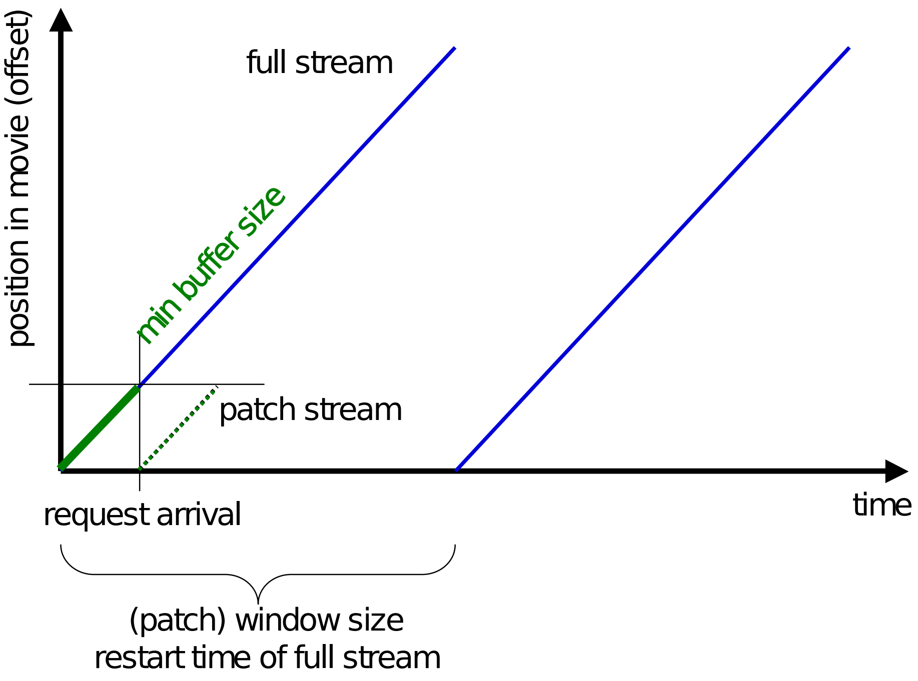

Figure 2.9: *After a multicast stream is initiated, arriving client requests are served by unicast streams patching the already transmitted part of the multicast stream. The same part of the multicast stream is buffered at the need for buffering on the client side equals the size of the individual patch. Figure adapted from Griwodz and Halvorsen [1]*

A Multicast Technique for True Video-on Demand Services" Hua et. al describes how patching techniques can be used in conjunction with bathcing. In this scenario, a server can be set up with a number of logical channels. As clients request the movie they are added into a client waiting queue, W_1 list used to batch clients together for the next available channel C_1 to multicast the movie. For clients who request the movie after the first batch has started to receive the video a new wait queue, W_2 is created. When the next available channel, C_2, is free the clients from W_2 are being multicast (instead of unicast) the missing patch. By having all the clients from batch W_2 listen to both C_1 and C_2 for the duration of the patch, channel C_2 can be made available for new clients at the time the patch has filled the missing portion. In other words, a batch of later arriving clients get a multicast patch stream, and buffers at the same time portion of the previous multicast stream until the patch is completely transmitted. As a consequence, more channels are available, and more clients can be served.

The latter patching technique will offer clients a Near VoD service. The use of batching in combination with multicast can reduce the server load significantly. The server bandwidth will rely linearly on the number of batches. The requirements on the client side will be a maximum bandwidth of two times the playback rate of the movie, in addition to the local temporary buffer equal to the size of the batch (in the case of being in a patch batch of clients).

2.4.2 Hierarchical Multicast Stream Merging (HMSM)

By combining the techniques known from Piggybacking, Patching and a version of Skyscraper (Dynamic Skyscraper), Eager et. al has proposed the HMSM technique [14]. The key idea is to let each transmission be based on multicast. Clients must receive the video data faster then the playback rate. It makes use of multiple streams, as in Skyscraper, accelerated streams, as in Piggybacking, and merging large multicast groups as in Patching. As a consequence the clients receives a True VoD service, and resources are reduced on the server side. However, the complexity of this technique can be hard to combine with user interactivity. Client buffers may also become large.

2.5 Discussion and Summary

We have looked at different algorithms that can be used to schedule media streams. What they all have in common is that they are assume that videos are encoded with a constant bit rate. However, as we will see in the next chapter, widely used codecs use variable bit rate, which can cause challenges not considered by the described scheduling techniques.

Staggered broadcasting results in the server bandwidth to increase linearly as a result of the number of channels in use. In "Efficient broadcasting protocols for video on demand" Paris, Carter and Long [2] compare the bandwidth per video when streamed with either Pyramid, Skyscraper or Harmonic (including Cautious Harmonic), see fig 2.10. Of these the Cautious Harmonic performs better both with regards to waiting time at the client side, as well as the server bandwidth when measured as a multiple of the playback rate. In chapter 3, we will describe some existing implementations, where patching is said to result in a network bottleneck when exceeding 5 client requests per minute at the transmission of a 3 mbps video. As a result we believe that harmonic

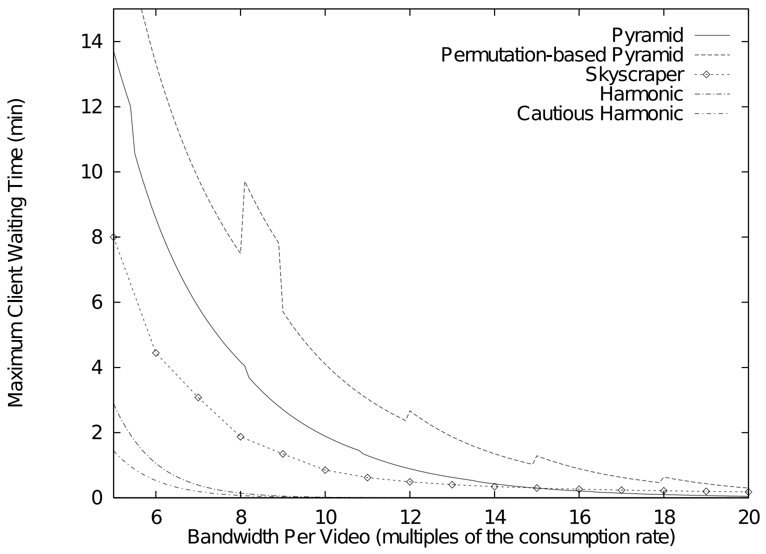

Figure 2.10: *The comparison of wait time and bandwidth when using the Pyramid, Skyscraper and Harmonic based scheduling algorithms. Figure taken from Paris et. al, 1998 [2]*

is the most efficient. Cautious Harmonic Broadcast efficiently limits the use of server bandwidth and can at the same time have a low wait time from the video request is sent to client can actually start consuming the media file. Quasi-harmonic broadcasting requires somewhat less bandwidth, but at the cost of a more complex implementation. Consequently, we have chosen to implement Cautious Harmonic Broadcasting for our experiments.

We will now take a closer look at implementation of three different scheduling techniques.

Chapter 3

Existing Implementations

Scheduling techniques like the ones discussed in chapter 2 have mainly been studied theoretically, run as mathematical simulations. As already mentioned, shortcomings in the deployed infrastructure in the 1990s made testing in real life scenarios difficult. However, a few implementations do exist. We will take a closer look at the implementation of three different techniques examined in two separate articles. The first two techniques are implemented in a real life scenario, whereas the third one is mainly simulated to perform measurements on the performance of the algorithm. Through our discussion of these we will look for relevant aspects to keep in mind when developing our own application.

3.1 Periodic Broadcast and Patching Services

In "Periodic broadcast and patching services" [15] an Internet streaming test bed is implemented, measured and analyzed. The goal was to expose and develop solutions to underlying system issues that arise when both periodic broadcast and patching algorithms are put into practice, one being caching implications. Below is a description of the scheduling techniques used in the streaming testbed.

Periodic broadcast: Clients play videos sequentially. By dividing a video into segments, and broadcasting these over a set of channels, clients listen on the different channels simultaneously, buffering later segments of the movie. Earlier segments of the video are transmitted more frequently. The study used the Greedy Disk-conserving Broadcasting (GDB) scheduling scheme. GDB seeks to conserve client disk space by letting a client receive data as late as possible [16]. Doing so, less buffering space is needed at the client as more data is received close to the time of consumption. The scheme uses equal bandwidth channels to transmit the video.

Patching: A new client listens to an existing multicast stream transmitting later segments of the video, which is then buffered. The server only needs to separately transmit the earlier frames that were missed by the new client. Patching differs from periodic broadcasting in that video is transmitted on demand, not continuously. The study used the patching as described in [17]. This algorithm makes use of multicast streams, also for the patch stream.

3.1.1 Testbed

We will now briefly describe the streaming server and client architecture used in the study. The server was set up with a Server Control Engine to handle interaction between the server and its clients, including separate threads for listening and for scheduling. It was also set up with a Server Data Engine, using separate disk threads to retrieve data from disk to main memory, and network threads to transmit data from main memory to the network. The mentioned threads would operate in rounds.

The client was equipped with a data engine that receives data and reorders out of order data, and is separate from the decoder software (i.e, video player).
RTSP, using TCP, was used for setup of a session, and RTP for data transmission.

3.1.2 Metrics Studied

The study took several performance measures into account when conducting the study. Following is a list of the main focus areas on both the client and server side.

Server side:

- System Read Load - The volume of video data transmitted per unit time from the underlying OS.
- Server Network Throughput - The volume of video data transmitted per unit time by the application.
- Deadline Conformance Percentage - The percentage of frames that the server was able to transmit to the network by their respective schedule deadlines.

Client side:

- Client Frame Interarrival Time - The time between packet r_i and $r_i + 1$ arrives at the client side.
- Reception Schedule Latency - The time from when the client requests the video until it receives the reception schedule.

Caching implications:

For both patching and periodic broadcast application level caching strategies were tested. Both Least Recently Used (LRU) and Least Frequently Used (LFU) were implemented. Through tests LFU showed to provide the best performance, and hence the preferred caching algorithm. For videos with a lower request frequency from end users, patching appears beneficial over periodic broadcasting as it proved to result in lower network bandwidth [18]. Without caching the crossover is at 2.8 requests per minute.

Caching was shown to have direct influence on the crossover point between when which of the two techniques, patching and periodic broadcasting, was the most effective with regard to system bandwidth, and was greater the larger the cache was.

Signaling Costs:

In the study TCP was used as the underlying protocol for the communication to set up a client session prior to the media streaming itself. For a periodic type broadcast the time to handle a client request is fairly constant up until a threshold caused by a limitation in how many TCP sessions the server could set up without crossing the TCP timeout period. This was shown to be true when having several movies, spread over fewer channels. With only one movie spread over more channels the threshold was lower. In other words, the signaling delay increases as the client request rate increases. It was assumed that the latter is a result from the increased chance of a thread being interrupted during processing.

Cost of Delivering Data:

An interesting detail is that the deadline conformance percentage was above 99% for all the experiments. This relied on all the data being present in the cache.

3.1.3 End-to-end Performance

Periodic Broadcast:

More streams and higher arrival rates of client requests did not necessarily lead to noticeably lower deadline conformance percentage on the server side. Client Frame Interarrival Time (CFIT) depends on the OS scheduling granularity, but will also be affected by the traffic load on the network. The study was able to serve up to 600 client requests per minute.

Patching:

The study showed that with 100 Mbit available in the network, no more than 5 clients could be served per minute as the network itself would become a bottleneck. The server itself experienced no difficulties as the deadline conformance percentage stayed stable at above 99.9%.

Evidence was given that without guaranteed processor time, the number of active threads should be chosen carefully. Also, with more than one movie, scheduling to avoid synchronization between the movies is necessary to avoid high peak server network throughput.

We will now look at the third implementation, this one using periodic scheduling.

3.2 An empirical Study of Harmonic Broadcasting Protocols

In our study of broadcasting techniques in chapter 2 we concluded that we found the CHB algorithm was the most promising of the ones studied. In "An Empirical Study of Harmonic Broadcasting Protocols" [19] Cautious Harmonic Broadcasting (CBH) is simulated, and therefore in particular interest to us. For a more thorough description of CHB please refer to section .

3.2.1 The testbed

The testbed of the simulation used a server and a client machine. Both the server and client were set up to use fork to handle the separate channels transmitting the disjoint segments of the video. For video, a dummy files was generated to simulate a 180 minute movie. The dummy file was sent with a constant bandwidth from the server. In other words, the transmission was based on constant bit rate. To avoid lost I-frames in the video, this simulation chose TCP over UDP, and experimented with different TCP packet and window sizes to find the optimal use of the protocol.
To reduce the effects of disk scheduling and I/O the server was set to read the video file from disk to cache before starting its operations.

3.2.2 Results

A single PC with Pentium III and 512MB RAM could manage to support up to 2400 independent video streams with a guaranteed wait time of 5.3 seconds for a 3 hour video. The researchers concluded that more work can be done investigating the benefits of switching to a connectionless protocol and reducing the number of processes required to implement the CHB protocol.

3.3 Discussion

For higher volume video on demand "Periodic broadcast and patching services" [15] shows that periodic broadcast is to be preferred over patching. With a more effective scheduling algorithm, like shown with cautious harmonic broadcasting, the network load should be possible to decrease in comparison with GDB that uses equal bandwidth channels.
For videos with a lower request frequency from end users, patching appears beneficial over periodic broadcasting as it is truly on demand, and is also proved to result in lower network bandwidth [18]. However, if using multicast, rather than broadcast, this should result in the network load increasing only in the instance of a connected client. This is due to multicast only being forwarded along the path of routers that have clients who have requested the multicast stream. With no clients the network would only be taking up bandwidth between the server and the neighboring router. More research can be done in this area as technology has advanced, possibly making

the cross over point between which broadcasting scheme is better different than found in the first study referred to.

The two experiments differ in that “An Empirical Study of Harmonic Broadcasting Protocols” [19] bases the transmission on up to 2400 concurrent streams forked into separate processes. The other “Periodic broadcast and patching services” [15] only tested up to 24 concurrent streams using threads. Even though more streams and higher arrival rates did not necessarily lead to noticeably higher deadline conformance percentage on the server side when using threads, the numbers are one hundred times apart and should be examined more closely to draw any conclusion with regards to effectiveness. An upper level was also found for broadcasting in “Periodic broadcast and patching services” at 1670 users, due to limitations with TCP timeout period and therefore refused by the OS. “An Empirical Study of Harmonic Broadcasting Protocols” does not discuss the resources used to handle users, which clearly is a parameter to take into regard when looking at overall performance.

“An Empirical Study of Harmonic Broadcasting Protocols” chose TCP as the transport protocol. As we will see in the next chapter, TCP is connection oriented and results in overhead due to being a reliable protocol. Buffering all packets transmitted until they are acknowledged by the receiver, TCP takes up more memory at the server than UDP that was used in “Periodic broadcast and patching services”. In addition, TCP is not made for multicast. TCP is a host-to-host protocol [20], and it is advisable to use a different protocol better suited for a broadcasting algorithm. More regarding this will be discussed in chapter 5.

The underlying system should be taken into account with regard to whether use separate processes or threads to handle the streaming of the segments. “Periodic broadcast and patching services” used threads, and showed successful server performance, in particular with deadline conformance percentage. Their caching policies also put less stress on the underlying server system. On the other hand, “An Empirical Study of Harmonic Broadcasting Protocols” reported that memory on the server side could be a potential problem when streaming several videos concurrently. They proposed two solutions. One was buying more memory for the server. The other solution proposed decreasing the number of segments, showing that merging segments to use half the number of channels would result in a 4% increase in the network bandwidth. We think providers may also consider what wait time is needed to meet the demands of the client. With 60 second wait time instead of 5.3, the number of channels would be only 180. Creating a full process for each stream requires more processing power than the same number of threads would. We recommend using threads.

We will now take a closer look at protocols and standards that are commonly used in streaming VoD.

Chapter 4

Protocols and Coding Standards for Video Streaming

When requesting Video on Demand the actual data of the video has to be transmitted over a network to reach the client. We have looked at different techniques to schedule these streams efficiently. We will now look at common protocols for transmitting the data.

We will first look at the network layer which is responsible to get data packets from the server, through all intermediate links, and all the way to the end client. Next, we will look at the transport layer which goal is to provide reliable and cost effective transport from the server to the client independent of the network in use. We then move up to the application layer where we need protocols to support the applications. For an overview of the different protocols please refer to the figure 4.1 of the Internet Protocol Suite. Finally we will look at a common encoding standard for video applications.

4.1 Network Layer Protocols

For communication on the Internet the Internet Protocol (IP) is used, and operates at the network layer. The main job is routing packets from a source to its destination, as well as service the transport layer (see section 4.2). There exists several protocols for use on the network layer. Some deal with control of the Internet, like the Internet Control Message Protocol and the Address Resolution Protocol, whereas others are specifically designed for routing.

Normal IP communication is unicast. However, as we have seen when discussing

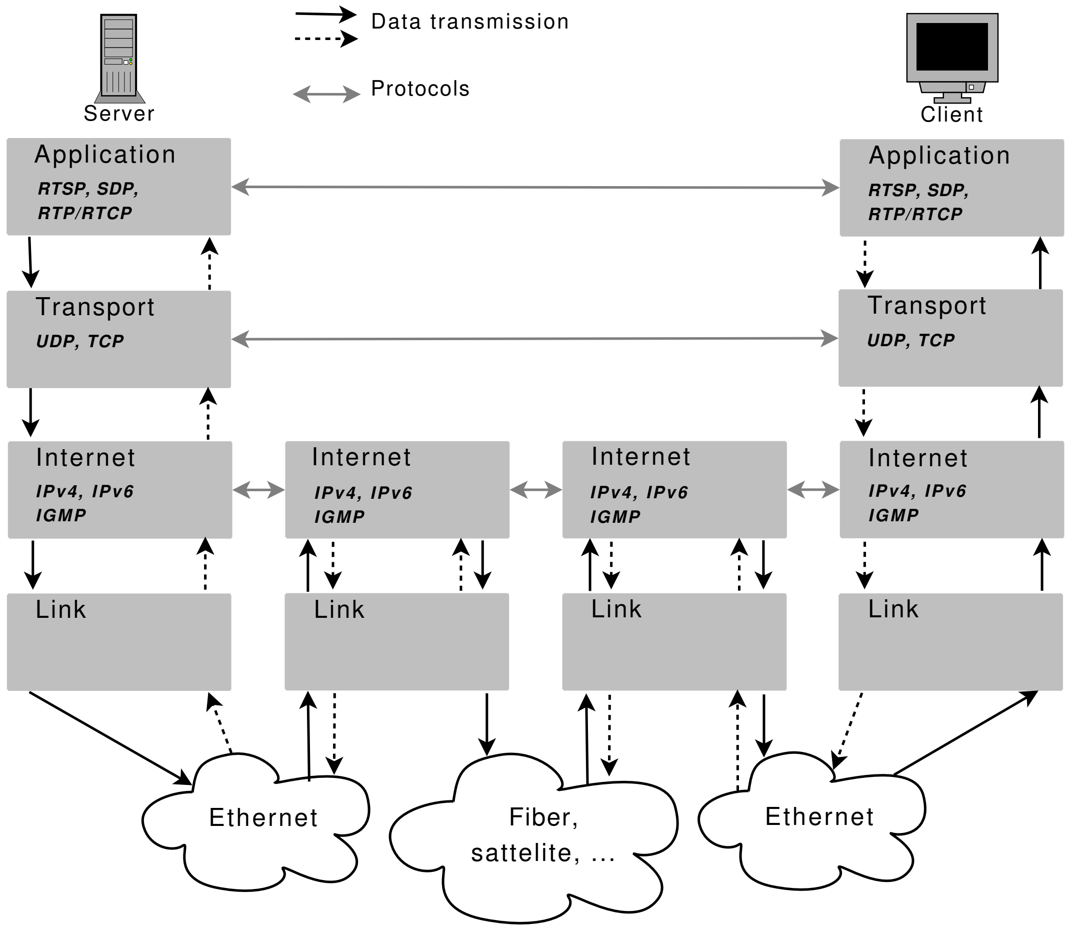

Figure 4.1: *An overview of the Internet Protocol Suite. The protocols used at the different layers are listed just below the name of the layer.*

scheduling techniques, there are many benefits from using multicast when streaming VoD. IP supports multicasting, and in IPv4 a class of addresses is set aside for multicasting, class D. The addresses span from 224.0.0.0 to 239.255.255.255. Multicasting is implemented by special multicast routers, most of which use the Internet Group Management Protocol (IGMP) [21].

Periodically, each multicast router sends a hardware multicast to the hosts on its Local Area Network asking them to report back on the groups their processes currently belong to [21]. This is known as a query. Each host sends back responses for all the class D addresses it is interested. These query and response packets use IGMP [22]. IGMP version 3 adds support for "source filtering", that is, the ability for a system to receive packets only from specific unicast source IP addresses, as required to support Source-Specific Multicast [23].
The multicast routers on their side exchange information with their neighboring routers.

Multicast spanning trees used for routing are based on the exchanged information.

4.2 Transport Layer Protocols

The transport layer links user sessions with the network, hence services both the layers above, as well we the network layer below. It provides both connection-oriented and connectionless byte streams from a sender to a receiver. The layer's basic functions are to receive data from the layer above, split the data into smaller units if needed, and pass them to the network layer for transmission.

4.2.1 User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) is a protocol for connectionless communication [24]. Packets are encapsulated in IP datagrams and sent off without needing any established connection with the destination address. In addition to the destination address, the UDP packet contains information about the source and destination port. Without the port fields the transport layer would not know what to do with the packets, as the ports guides where to deliver the payload. in other words, UDP provides an interface to the IP protocol with the feature of demultiplexing processes using ports. With few features, the protocol is both simple and fewer messages are required to be transmitted than what we will see with a connection oriented protocol. For small tasks like looking up the IP address for a domain name, this can be beneficial, as only the exchange for a request and the reply is needed. The requesting host can simply resend the request if the reply is not received, instead of setting up an entire connection.

UDP provides no guaranteed transmission, nor connection to a host. It is an unreliable protocol. The protocol does not support modification of its transmission rate, and can result in congested networks leading to packet loss. If packets arrive out of order, no support to reorder them is in place. Reordering of packets can be performed either by choosing a different transport protocol with reordering support, or by using another protocol on top of UDP providing the added services needed. Real Time Protocol (RTP) is one example of a protocol offering sequencing of packets. We will describe RTP later in this chapter. This functionality is also available at the transport layer by using the Transmission Control Protocol (TCP), which also offers reliable transmission. UDP does, however, have a 16-bit checksum field that is used for error-checking of the header and data. This way UDP ensures a level of integrity of the received data.

4.2.2 Transmission Control Protocol (TCP)

TCP was designed with the focus of offering reliable end-to-end communication over an unreliable inter-network [21]. TCP is said to be connection oriented. This means that TCP sets up a connection prior to transmitting the data, using what is known as a three-way-handshake; in a client server scenario the client will request a transmission from the server, the server will reply with an acknowledge (ACK) to the request, to which the client acknowledges to the server again. Once the connection is established, the data is transmitted until the connection is torn down by one of the hosts. Termination of the connection is done by sending a packet indicating it is finished (FIN) [20].

TCP uses packet header sequence to reorder packets that are received out of order upon reception. A client application can be ensured that all data is delivered to the layer above in the order it was transmitted.

When sending out data the TCP protocol expects an ACK to be sent as a receipt for all packets. In the case of a missing ACK, a retransmission is scheduled. Hence, using TCP you can be sure all your packets will arrive at their destination. During the connection setup a calculation of the round trip time between the client and server is also conducted. TCP uses this information to calculate how long to wait (timeout) for an ACK before retransmitting the packet. Consequently, in the event of a lost packet it can take time before the packet is retransmitted, causing a delay in the transmission. With real time applications, where arrival time is prioritized over reliability, UDP is often the preferred protocol.

TCP continues to send more packets before all ACKs are received. The number of packets in transit is slowly increased, and the number is referred to as the congestion window. In the case of a timeout waiting for an ACK, TCP reduces the transmission rate. As a result, all who use TCP will modify their usage of the bandwidth according to the perceived traffic in the network. TCP is said to be fair in this sense, as streams that share a path will reach an equal share of the bandwidth.

4.3 Application Layer Protocols

Application level protocols have an important task in creating an interface to the underlying transport layer. Applications can use the transport layer interface directly. However, several protocols exists, like Hyper Text Transfer Protocol, Session Initiation Protocol, Simple Mail Transfer Protocol, and many more. They can handle numerous functions, including synchronization, encryption and content delivery. The services may vary according to what underlying transport protocol they make use of. We will describe three that are in common use for streaming video on demand, namely Real Time Streaming Protocol, Session Description Protocol, and Real Time Protocol.

4.3.1 Real Time Streaming Protocol (RTSP)

RTSP is an application level protocol for delivery of real-time data like audio and video, and is defined in RFC 2326 [25]. The protocol is intended to control multiple data delivery sessions and can make use of transport layer protocols like UDP, and TCP, as well as the application level protocol RTP. It does not typically deliver the continuous streams itself, but can be said to act as a "network remote control" for multimedia servers. This includes allowing Trick Play commands like *play* and *pause*. The set of streams to be controlled is defined by a presentation description. The format for the description is not set, but the Session Description Protocol that we will describe below is commonly used.

4.3.2 Session Description Protocol (SDP)

SDP is a format for session descriptions (SDs) defined in RFC 4566 [26]. RFC 4566 defines a session descriptions as follows: " A well-defined format for conveying sufficient information to discover and participate in a multimedia session." A common scenario is a client requesting a description of one or more media streams. A server can reply with an SD of a media stream(s) so the client can prepare and connect to the right channel(s) to receive the data. SDP does not incorporate a transport protocol, but can contain information about what protocol to be used for a session. The protocol is ASCII based, and has fields used to describe a session. New values that can be used in the fields are updated regularly, adding to the versatility of the protocol.

4.3.3 Realtime Transport Protocol (RTP)

RTP [27] [28] [29] defines a standardized packet format for delivering real-time data. With RTP, audio and video are transmitted over unicast and multicast network services, i.e. UDP and TCP. It is typically used in conjunction with a control protocol, RTCP. Both protocols are independent of the underlying transport and network protocols. However, for video streaming RTP is often run on top of UDP (see section 4.2.1) to make use of its multiplexing and checksum services. RTP and RTCP operate on separate channels, where RTP uses the even number and RTCP the odd.

RTP provides end-to-end network transport functions. The functions include payload type identification, sequence numbers, timestamping and delivery monitoring. RTP itself is not reliable, but relies on the lower level services for such functions. RTP is also described as a protocol framework that is deliberately not complete. Separate profiles and format specifications exist. One example is the octet in the header that contains the marker bit and the payload type. These can be redefined depending on the profile requirements. There is also support for data header addition by setting the extension header bit.

The RTP control protocol, RTCP, has a primary function in providing feedback on the quality of the data distribution done by RTP. The data delivery is monitored in a manner scalable for multicast networks. With the RTCP feedback a certain level of flow and congestion control can be used for adaptive encoding. It can also be used to diagnose faults in the data distribution. To get the feedback receivers send reports to the server at a rate that is scales with the number of participants.

4.4 Coding Standards and stream formats

In "Computer Networks" [21] Tanenbaum explains that even for a 1024 x 768 pixes video, with 24 bits per pixel and shown at 25 frames per second , it needs a bandwidth of 472 Mbps. With such a great demand for bandwidth for feeding raw video, there is a need for video compression. The Motion Picture Expert Group (MPEG) standard was developed to cover motion video as well as audio coding. We will briefly describe two of them, MPEG-1 and MPEG-2, followed by an explanation of how the MPEG-2 stream is multiplexed and encapsulated for storage and transmission.

4.4.1 The Motion Picture Expert Group (MPEG) Video Codecs

MPEG-1 has has three parts, audio, video and system. The audio and video encoders work independently, and the system multiplexer integrates the two. Video compression can be achieved by taking advantage of the fact that image frames are often almost identical. Hence, it is possible to only update the changes in picture, and not necessarily the entire screen. For this MPEG-1 has four different kinds of frames [21].

I - Intracoded frames: Self contained encoded still pictures. These are the frames that contain the most information, as it contains an entire image, and are inserted once or twice per second.

P - Predictive frames: Block-by-block difference with the last frame. P-frames code only interframe differences. They can refer to I and P frames.

B - Bidirectional frames: Differences between the last and next frame. B-frames are like the P-frames, only that they can refer also to the upcoming frame, not only previous ones. They can refer to I and P frames.

D - DC-coded frames: Block averages used for fast forward. They are only used to make it possible to display low resolution images.

MPEG-2 is fundamentally similar to MPEG-1, with the exception of D-frames that are no longer supported. Instead of supporting one, it supports four resolution levels, all the way up to HDTV.

MPEG-2 addresses both audio and video, and provides the MPEG-2 system with a definition of how audio, video and other data (like subtitles) are combined into a single or multiple streams which are suitable for storage and transmission. We will not look at such stream formats.

4.4.2 Elementary Stream

[30] To each audio-visual object in a scene, there is related information that is compressed. This information is referred to as an Elementary Stream (ES). The ES is in other words the output of encoded video or audio encaptions (it can also be i.e. sub titles). Each ES in an MPEG-2 is packetized and wrapped in a structure called Packetized Elementary Stream (PES). The resulting PES in interleaved into either a Transport Stream (TS) or a Program Stream (PS) which we will describe below. The PES packets start with a lengthy header structure. The most important features are the length of the packet, a stream id, as well as presentation and decoding time stamps for the content [31] [32].

4.4.3 Encapsulation Formats

The PES packets for MPEG-2 can be encapsulated for storage or transmission in two formats, TS and PS. Below is a description of the two, including their usage areas.

PS is aimed at a relatively error-free environment such as disks (and DVDs). The Program Stream's packet's may be of variable length. When transmitting a PS over a network the ESs in the PS are multiplexed into separate streams, i.e a separate stream for audio, and a separate for video. [33].

TS combines Packetized Elementary Streams and one or several independent time bases into a single stream. The Transport Stream is designed for use in lossy or noisy media. The respective packets are 188 bytes long, including the 4-byte header [33]. The TS is well-suited for transmission of digital television and video telephony over fiber, satellite, cable, ISDN, ATM and other networks, and also for storage on digital video tape and other devices. When sent over a network only one stream is transmitted [31].

4.5 Summary

We have now described common protocols for transmission of data over an internetwork. Our focus has been on those protocols that are commonly used for streaming video on demand. We have also given a short background to the MPEG-2 video codec and encapsulation formats for transmission. This to provide a background, as we move to chapter 5, describing our design for a VoD streaming server and client.

Chapter 5

Design

In chapter 3 we looked at some existing implementations of scheduling algorithms, including Cautious Harmonic Broadcasting (CHB). Where one of the existing implementations performed real movie transmissions, another implementation using CHB used a dummy file. They thereby avoided issues that can occur when working with real image data. In this chapter we will detail a design for integrating CHB with existing media players. As such, this will affect some of our design choices to comply with existing solutions. Through such an implementation we hope to get a better understanding of the challenges that arise in writing a streaming server and a client supporting periodic scheduling.

To create such an implementation one needs to determine what content to transmit, and what encapsulation and encoding to use. Furthermore, one needs to segment the movie on the server side, and reassemble them on the client side. To do so the server needs to know how many segments to split the movie into, and the client needs to know how many to receive. The data for each segment needs to be transmitted from the server and received by the client. We will now look at the required considerations writing such applications; for both the server and client side. Last, the CHB scheduling scheme requires client buffering. We will look at different options when receiving the movie in segments, and discuss these.

5.1 Stream Format

Several stream formats and codecs exist today. MPEG-2 video encoding is perhaps the most widely used of today, and is the standard for common DVDs. It is therefor our choice of encoding as well. However, the encoding is primarily a technology concerned with movie compression from raw video and audio signals. When transported it needs to be transmitted in a suitable format. For MPEG-2 there are two options, namely Program Stream (PS) which belongs to the MPEG-1 standard, and Transport Stream (TS) that came with the MPEG-2 standard. For a description of the two please refer to section 4.4.3.

5.1.1 Encapsulation: Transport vs. Program Stream

In our work we had to decide on whether to use PS or TS. The PS transmits the video and audio separately. This can be useful when using unicast because you choose to drop less important frames at the server side to adjust to a lower bandwidth, if needed. In a multicast scenario such individual needs can not be taken into account with today's protocols. Hence, a common bandwidth will have to be chosen for all recipients. As TS was designed for use in a lossy environment, it seems to be the better solution. With PSs, lost packets can cause disruption to the synchronization between i.e. video and audio. As long as the packets are delivered synchronously, they will play well, but not necessarily with packet loss as the separate ESs sent do not contain synchronization code. In TSs the TS packet header contains more meta data that will help synchronization in the event of packet loss.

5.2 Segmentation and reassembling of the movie

To implement a periodic broadcasting scheme it is necessary to split the movie into disjoint segments. As discussed in chapter 2, we found the Cautious Harmonic Broadcasting algorithm to be the most promising. Below is be a description of what implication cautious Harmonic Broadcasting will have when determining the size of the segments and how we designed the scheduling scheme. Following is a description of how to create such segments, and how to reassemble them on the client side.

5.2.1 Design of the Cautious Harmonic Broadcasting Scheduling Scheme

The Cautious Harmonic Scheduling Scheme is described in the Harmonic Broadcasting section 3.2. All the Harmonic Broadcasting schemes operate with segments of the same size. In our code, we decided to use the Cautious Harmonic Broadcasting algorithm.

However, for the simplicity of the code, we altered the algorithm so that instead of sending segment 2 and 3 on channel 2, we send segment 2 at full bandwidth on channel 2, and segment 3 at half bandwidth on channel 3. This will not affect the arrival time at the client. However, adding an extra channel will increase the bandwidth with a little over *b*/2 added to the total. 5.2 shows a graph displaying the theoretical bandwidth expected when streaming a 1100 kbps movie according to the CHB and our edited version of CHB. Our scheduling scheme is designed as shown in figure 5.1.

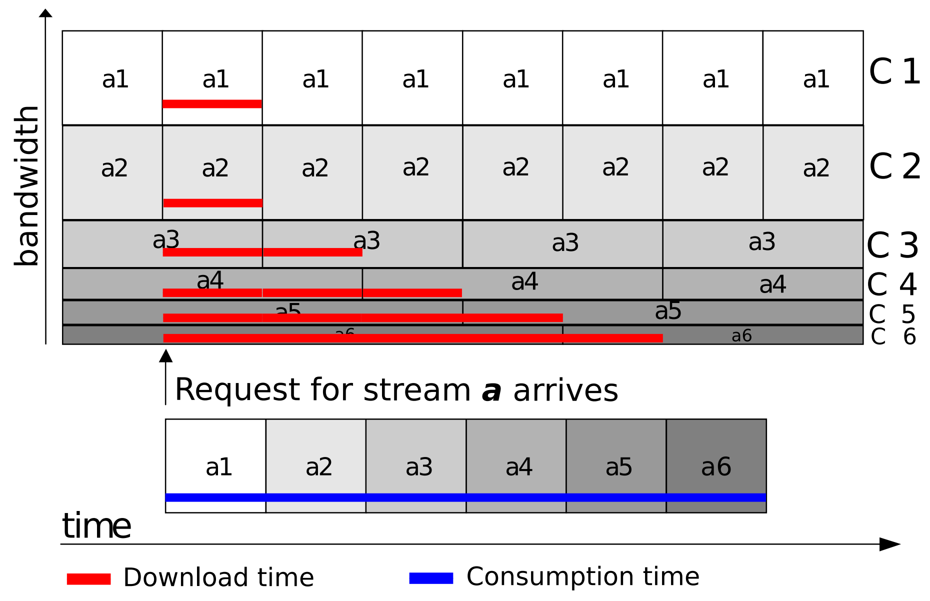

Figure 5.1: *Our implementation of the Cautious Harmonic Broadcasting Algorithm*

Implementing a periodic scheduling scheme also requires the use of logical channels, to transmit and receive the streams. We will discuss the possible technical solution on how to create such channels later in this chapter in section 5.5.1

5.2.2 Segmentation of the movie

To split a movie into disjoint segments, a design issue had to be made with regard to variable bit rate vs. constant bit rate. The scheduling theories we have discussed in this thesis are all based on a constant bit rate, hence, we decided to use a constant bit rate as well. As we chose the TS format, we decided it was wise to not split a segment in the middle of a TS packet. It may be easier to read the TS packets on the client side if they are sent in the same packet unit in the event of packet loss, and not split on two different packet units. As the TS packets have a constant size of 188 bytes, we decided to split the movie into blocks of 188 bytes, and then decide how many blocks to include in each segment.

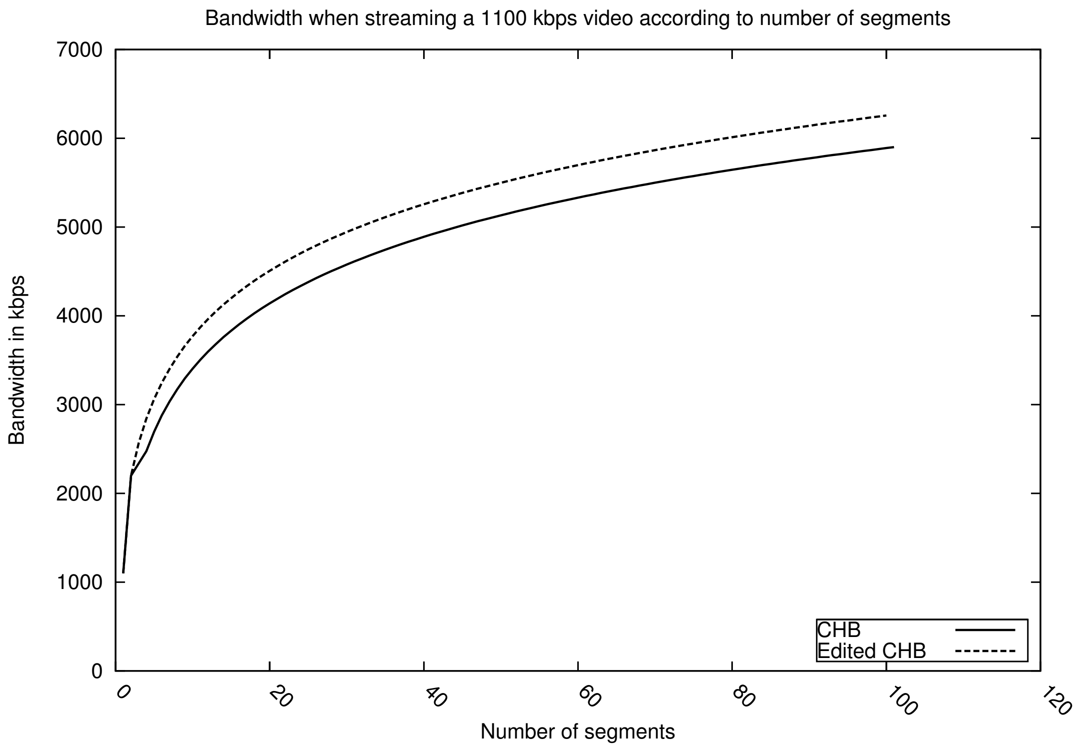

Figure 5.2: *Total bandwidth for transmitting a 1100 kbps movie using either CHB or our simplified CHB algorithm (Edited CHB). For average numbers measured see chapter 7.2.5*

There are two ways to determine how to split the movie into segments. One way is deciding how many segments there should be, and calculate how many blocks should be in each segment. Alternatively one can determine how long the client wait time should be, and calculate the number of segments based on that in relation to the entire duration of the movie. As wait time can be a determining factor in the success of the deployment of a VoD service, we decided wait time would be the better approach. This however, does require that the duration of the movie is known at the server side.

5.2.3 Reassembling the movie segments

The CHB algorithm calls for a significant amount of buffering at the client side. As we will see later in this chapter, caching will be used to buffer data packets from the transport layer. As a result, it can be difficult to keep all the segments in cache if the end user has a machine with low caching capabilities. Therefore, we decided to write the segments to disk. Segment buffers are temporary, and only required until consumption of the segment. It made sense to write the segments to separate files, as appending to a file is technically much easier than 'prepending'. Another solution could have been to map the entire video file area, and write the received data to its corresponding offset in the file. Such a solution would require more meta data about the received media, including the offset of where to write in the file, as well as knowledge of the entire file size prior to receiving the segments. In case of loss, this could be an advantage when demultiplexing the video. This is due to how media players handles missing frames. However, we expect that a solution that writes each segment to its own buffer is more flexible. It makes it easier to free the consumed segment from the clients disk. Writing to one big file would reserve a file area equal to the size of the video throughout the entire time of downloading and consumption. It does, however, raise a greater challenge when delivering the data to a demultiplexer in the media player, as the media is distributed at different locations on the disk. In that case, the media player will need to have a buffer that fetches or delivers data from the correct file buffer at the correct time. In our implementation we will not deliver the media to a player. Instead, we will reassemble the segments at the end of the download for easy assessment of the transmitted data.

When downloading the segments, the CHB scheme indicates that a client always starts the download at the beginning of the segment. We will discuss the details of this later in the chapter. However, given enough disk space, we see no reason to wait for the initiation of the download until the start of the segment. As we will see, recognizing the beginning of the stream can represent a challenge in itself. Instead, we will allow the client to start buffering the segments immediately (as seen in figure 5.3). As a result, the former and latter part of the segment will be arriving in opposite order. Our solution to this is to write the two parts to each their segment. The same structure that will reassemble the file segments, may just as well reassemble the two segment parts. There is an exception in the case where listening starts at the exact beginning of the segments. which requires only one file buffer. Such a buffering scheme will avoid any problem with recognizing the start of a segment in a data stream, there is no need to known the size of the segment in advance. It also fits well with the chosen design for reassembling all of the segments.

5.3 Wait time

The play time of a segment according to the CHB algorithm is equivalent to the longest time a client has to wait before starting the consumption of the movie. Wait time can be said to be directly linked to the number of segments. As mentioned in section 2.1.3, wait time affects the end users experience of the service, and needs to be chosen carefully. However, for our implementation, we do not seek to research what maximum

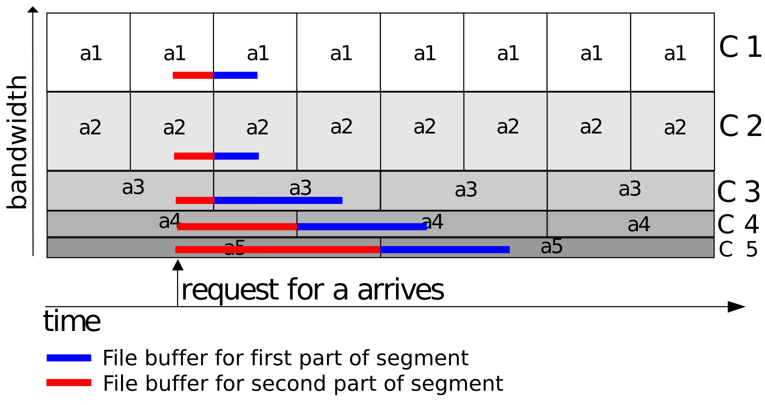
a1
a2
a3
a4
a5
C 1
C 2
C 3
C 4
C 5
bandwidth
time
request for a arrives
File buffer for first part of segment
File buffer for second part of segment

Figure 5.3: *Client listening to channels, while buffering data to be consumed later to files. The latter part of the segment is in all the shown channels downloaded prior to the beginning of the segment.*

wait times end users demand for a VoD server. Hence, our choice of wait time will not be scientifically based, but rather based on our own subjective preference. We have chosen 60 seconds as our wait time. We believe this is a wait time end users are willing to wait. For profit seeking providers it is also a suitable time slot for streaming advertisement.

5.4 Integration with existing media players

To get a fuller understanding of the real challenges of implementing a periodic broadcasting algorithm we decided to create a server and client that can easily be integrated into existing media streamers and players. To do so we needed access to source code, limiting our choice of streamers and players to open source software. Two media players that meet those requirements, VLC [5] and MPlayer [6]. Both use the same streaming code library, live555 [3]. As a result, we decided to implement our code as part of a modification and addition to live555. Using live555 gives us the benefit of having a library for an RTSP server, as well as support for SDP, RTP and TS framing. This matched well with our considerations that we will describe below.

5.5 Transmission of the segments

5.5.1 Channels

For the transport of CHB, a streamer requires channels to transmit the segments of the movie over the Internet, using the Internet Protocol Suite. In our algorithm one segment is needed per channel. One way to create channels is to have one address for each channel. This could potentially create a tremendous need for IP addresses. With IPv4 still being predominant, broadcast and multicast addresses could, be in short. The easiest solution seems to be that of using ports in combination with the multicast IP address. There is a trade off between using ports instead of separate multicast addresses for the channels. Ports operate on the transport layer, and multicast forwarding happens at the network layer. In our implementation this means the client will continue to receive traffic on all channels for the duration of the download, and not just on the channels that are remaining to be consumed. Consequently, the bandwidth at the client will be constant during the download, and not decreasing as outlined by the algorithm. Using overlay multicast is one possible solution to address that problem. A further discussion of this is found in section 8.3.

5.5.2 Routing and transport protocols

We choose multicast over broadcast as multicast will only transmit to the spanning tree made up by its member group. Broadcasting will consume resources in the entire network. The trade off with multicast is that it requires more advanced routers, adding to the total cost while decreasing the overall network traffic.

As described in section 3.2, an existing implementation using TCP has been tested. In their conclusion they suggested implementing the CHB protocol in a connectionless environment. Their rationale for choosing TCP instead of UDP was the importance of I-frames when viewing video.

As earlier concluded, broadcast and multicast makes little sense when using TCP which is a host-to-host protocol. Acknowledgment packets would put a tremendous strain on the server in the event of numerous clients. The clients on the other hand could experience significant delay caused at the occurrence of packet loss triggering retransmissions. There is a trade off between packet delay and packet loss for the client, as they are both factors that will decrease the viewing experience.

We chose to use UDP instead TCP. Considering there are drawbacks with using both TCP and UDP, and seeing how TCP is not fit for use in a multicast environment, UDP is the best solution. We did, however, take a closer look at the quality of the received file in chapter 7.3 to see if we could find any indication on the effect of a lossy environment.

5.6 Summary of requirements for a CHB server and client

Using CHB with UDP we have the following needs for setting up a streaming session between the server and client:

The server application must:

- **Establish client sessions:** In a reliable way a server must be able to give a client information needed to receive the segment streams.
- **Divide segments into fragments:** The video segments will be sent as payload in UDP packets. Keeping bandwidth as constant as possible,the segments need to be fragmented into smaller packets. These packets must be sent with a steady frequency.

The client application must:

- **Know when and where to listen:** A client requesting a video needs to know how many channels to listen to to receive all segments. Furthermore, there is a need to know when to start and stop listening to a given channel.
- **Reassemble the disjoint segments into a video** A client needs to have information about which channel corresponds to which segment to be able to deliver the segments in the right order to a file or media player.
- **Reassemble the UDP packets for each segment:** The media will be sent as payload in UDP packets. We have no guarantee that the packets will arrive in order, or without delay. The application needs to determine how long to wait for delayed or lost packets, as well as put them in order.
- **Work with existing media players:** The application needs to be able to work with the chosen media players, VLC and MPlayer.

5.7 Establishing sessions

A reliable connection needs to be established when setting up the session between the client and the server. Without reliability the server does not know if the client has received the needed session information. As the streaming of the video as a whole can be regarded as one session, we will look at each segment as part of a subsession that is also responsible for the corresponding channel.

Two common protocols are frequently used to set up sessions for real time streaming, namely RTSP in combination with SDP. RTSP requests uses TCP, hence, is reliable. Following is a description of how the two protocols can be used together to establish a server-client streaming session.

5.7.1 Design choices for Real Time Streaming Protocol (RTSP)

RTSP helps set up a session between a client and a server using TCP as the underlying transport protocol. The setup itself is therefore reliable and connection oriented. On the other hand, the streaming of the video will be using UDP. The most common commands in the protocol, and the ones relevant for our implementation, are listed below with explanation on how we see it fit the design for setting up our client sessions.

OPTIONS An OPTIONS request can be made at any time by the client. A server reply describes the commands it supports.

DESCRIBE Following an OPTIONS request a client will typically request the video on demand. We will use the format of a Uniform Resource Identifier (URI) [34]. A DESCRIBE response from the server can then respond by including a session description in the format of SDP.

SETUP Based on the session description the client will be able to set up a connection for each segment. The segments will all have corresponding URIs. The client can use the SETUP command to specify the transport mechanisms to be used for streaming the segment. In our case the client will specify the protocols and the ports it will listen to, as well as the URI for the segment. The response will specify the actual parameters the server will offer.

PLAY When a server receives a PLAY command from a client, the server will start transmitting data according to the specifications in SETUP. In our implementation, the streaming will be prescheduled, and already streaming. Only one PLAY command should be necessary to send to the server, as the client already has the information needed to receive the channels.

TEARDOWN The TEARDOWN command stops the stream delivery, freeing all resources associated with it. As our implementation streams continuously in the time period the video is offered, the TEARDOWN message is mainly to terminate the session on the client side. The server will continue to stream as determined by the service provider.

Other commands that are outside the scope of these thesis are `ANNOUNCE`, `PAUSE`, `GET_PARAMETER`, `SET_PARAMETER`, `REDIRECT` and `RECORD`. Please refer to [25] for their method definitions.

We will now outline our design of the session description that is to be transmitted as part of the DESCRIBE response from the server.

5.7.2 Design choices for Session Description Protocol (SDP)

The session description needs to contain all the information the client needs with regard to what ports to listen to, what segment corresponds to which channel, and what protocols needed to set up the client subsessions. Below is a list of the main syntax used in a session description, followed by an example of how SDP can be used to give information about segments streamed using a CHB scheduling scheme. The available fields should suffice to meet the requirements of our implementation.

v= protocol version
o= originator and session identifier
s= session name
i= session information
t= time the session is active
a= zero or more media attribute lines
m= media name and transport address
c= connection information - optional if included at session level

Another useful field is the SDP tag `a=x-fmt:`, known as the extended attribute. It is flexible and can be used for basically anything a developer wants to include. For more field and attribute values please refer to [26].

Not all the fields are relevant for our implementation. We have chosen to not send information about the length of the movie or its segments, hence the `t`-field is not needed. Using the remaining fields we will include information about the following:

- The SDP protocol version.
- The server IP address and its session identifier.
- The name of the session, given as information about the streamer
- The name of the streamed video
- What type of transmission. We have decided to use 'harmonic' instead of unicast, multicast or broadcast, letting the 'harmonic' term indicate that the session consists of several independent subsessions transmitted over separate channels.
- What kind of media the session consists of, namely video.
- What port and protocol to use for a subsession.
- The segment number corresponding to a given port.

For a sample of the SDP description please refer to section **??**.

Upon reception of the session description the client should have the sufficient information to set up the channels and start receiving the streams. However,as previously pointed out, the UDP packets may arrive delayed, and therefore out of order, as well as getting lost in the network. Consequently there is a need to reassemble the packets at the client side prior to reassemble the distinct segments with each other. In other words, our implementation needs two levels of reassembly, one of the segments with each other, and another level reassembling the packet payloads that make up each segment. A protocol providing such support at the application level is the Real Time Protocol (RTP). However, we will also discuss other options considered.

5.8 Reassembling the fragments to segments

As mentioned the movie is split into disjoint segments. The segments are split into fragments streamed by RTP over UDP from the server. We need to reassemble these fragments on the client side.

RTP is a protocol made for streaming real time data. The services include payload type identification, sequence numbering, time stamping and delivery monitoring. For our implementation we need to be able to sequence packets arriving out of order. The sequence numbers included in RTP allow the receiver to reconstruct the sender's packet sequence. The sequence numbers might also be used to determine the proper location of a packet, for example in video decoding, without necessarily decoding packets in sequence. However, with only a 16-bit sequence number [27], we could experience a wrap around of the numbers if segments are long and require more than 65535 RTP packets. This causes a problem for our implementation, as we might need to sequence packets exceeding that range. For sequencing within a short time range, the 16-bit number will be sufficient. However, to reassemble the fragments into an entire segment one needs a greater number. This is particularly important for the client to be able to recognize when the entire segment is downloaded. Streaming from an arbitrary point, information about when a movie segment has iterated to the client's listening start point is vital to stop the streaming of that particular segment.

Following is a discussion of different possibilities.

Reassembly of fragments based on TS content:

The TS packet format, and the elementary stream it is encapsulating contains data about the video. Below is a discussion to whether this information is suited for reassembly of the fragments of each separate segment.

Based on TS packet header The TS packet header does not include any sequence or presentation time information [30]. It generally requires that the underlying layers takes care of packets arriving in order. However, the TS packet header has an extension possibility through the adaptation field. The adaptation field may or may not include a program clock reference that can possibly be used for reassembly. This requires that the TS packets are never split up across segments or fragments, something we have taken into consideration. It also requires that careful consideration is taken during the generation of the TS encapsulation.

Based on PES time stamp Decoding Time Stamp (DTS) or Presentation Time Stamp (PTS) is an optional part of the PES header, indicated in the PTS_DTS_flags field [30]. However, the PES packets can be split up into several Transport Stream packets. If removing the payload from the TS packet we could have a lot of elementary stream data with no sequencing information at all. If the looping was the only cause for reassembly, this could possibly work if the data otherwise was delivered in order without loss. Unfortunately, using RTP over UDP we have no guarantee of a reliable connection.

In addition to this, reassembling requires information that is otherwise mainly left for the media player. We think a flexible solution should be able to reassemble the fragments independent of the data that is carried.

Not recommending to use the information in the TS packets, we look at the RTP packet format for other solutions. We have already established that the RTP sequence number will be insufficient.

5.8.1 Reassembly of fragments based on the RTP packet header:

In addition to the sequence number, the RTP packet header contains a time stamp as well as an optional extension header. We will now discuss these two fields as possible solutions.

Based on RTP extension header The use of the RTP extension header is not defined, but should not be used to describe the payload according to [27]. Sequencing is, like seen with TCP, a typical feature performed prior to data reaching the application. The payload of the RTP packet would be decreased, and the use of the extension header would imply a small protocol overhead of little significance. As a result, this occurs to be the a feasible and logical solution. However, live555, the streaming library we have decided to use, has no such support.

Based on RTP time stamp For MPEG files the RTP timestamp will generally contain information regarding the servers system clock reference, and should not be passed on to the MPEG decoder (see [28]).

That being said, the timestamp in RTP can be generated by a sampling clock (as opposed to the system clock). The requirements are that the timestamp is incremented monotonically and linearly to allow synchronization and jitter calculations. The calculations are included in the RTCP packets sent to the server as part of the receiver report. The timestamp is depend ant on the format of data carried as payload. or may be specified dynamically for non-RTP formats. The timestamp is most commonly used for synchronizing a video stream with an audio stream. For our implementation we use the TS format, that contains both audio and video in one stream. In addition, the TS format supports a program clock reference (PCR), hence the TS packets will contain timing information. In other words, except for monotonic and linear increments, we find that the timestamp can be used instead of the sequence number to reassemble the fragments. Wrap around should not be an issue, as we have a 68 years range available [27].

5.9 Process management

As many segments have to be transmitted concurrently, there is a great need for effective process management of the tasks ranging from disk I/O to sending and receiving the streams. In the implementations discussed in chapter 3 two different methods were in use. The first study had used threads to handle the segments, whereas the second study let each segment be handled by a separate process by using the `fork()` system call. Both threads and separate processes will require context switches adding extra strain on the processor. Several separate processes also require their own stack and heap, using more memory than a single process. There is another solution, that of using and even loop in combination with the `select()` system call. In short the `select()` function receives a set of file descriptors that listens on the desired ports. Using an event loop, events on the ports will be caught in the loop. Also other event handlers can be added to such a loop. We find that an event loop is better suited for our needs. With only one process running we save both processing power and memory for the other challenges that we have outlined with regard to buffering and scheduling.

5.10 Overview of server and client communication

We have now looked closely at many aspects to keep in mind when implementing our CHB server and client. here is an overview outlining the main structures.

The server will use an event loop to schedule the iterating segments. The segments will be transmitted as multicast streams. A TS video file will be split into segments (video source) and passed to a framer ensuring that 188 byte size fragments (or frames) are added to RTP packets as payload. RTP sinks will ensure that the RTP packets are sent at the corresponding port using UDP. Using RTSP the server will set up client sessions, sending the needed information to the requesting client through reliable communication. Once the session is established, the client will start receiving the RTP packet streams, seen as RTP sources. The RTP sources will pass the payload to a framer. The framer on the client side will deliver the fragments to the corresponding segment buffer (video sink). These segments will be handled by a main structure that keeps track of the segments and the order they need to be reassembled. Please refer to figure 5.4 for a graphic model of the server and client.

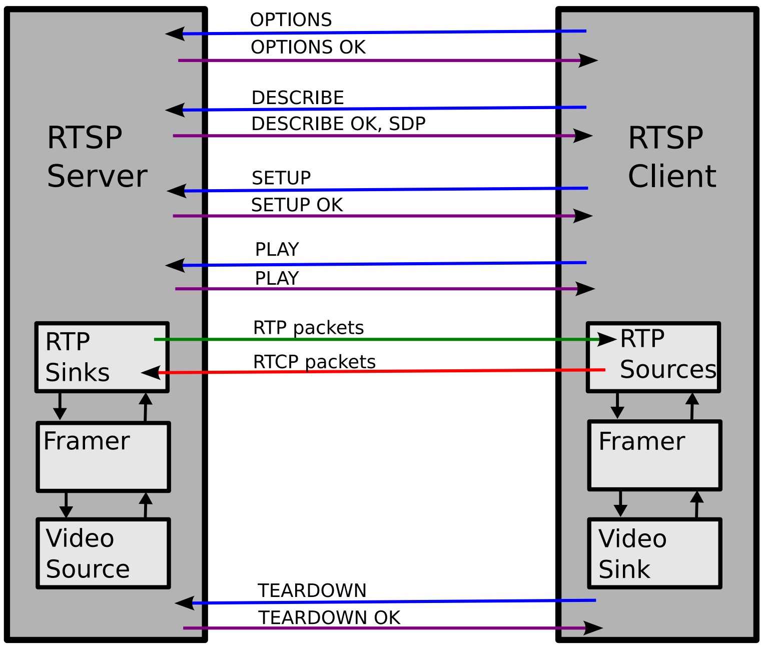

Figure 5.4: *Our implementation of the Cautious Harmonic Broadcasting Algorithm*

We will now take a closer look at how we implemented this design, using our version of CHB and with support from the live555 code library.

Chapter 6

Implementation

We have looked at considerations for creating an application for CHB scheduling, and discussed our design choices, in chapter 5. We will now describe how these choices were put into practice. One goal was to create functions that can be integrated with existing media players, like VLC and MPlayer. As such, we found that the live555 streaming library can work as a good framework for our development. We will start out with an introduction to live555, before we go into details of changes and addition made to implement CHB.

6.1 Live555

The live555 Streaming media code forms a set of C++ libraries for multimedia streaming, using open standard protocols including RTP, RTCP, RTSP, SDP and Session Initiation Protocol (SIP). These libraries can be used to build streaming applications [3].

6.1.1 Typical program flow using live555

A typical server application using live555 will set up an RTSP server to handle incoming clients by listening to a network read handler. A network read handler is an interface to the network, implemented with a socket. Other network read handlers can be established as part of further streaming that is initiated. All active network read handlers are monitored by an event loop. When a request from a client is received, the response will be added to a task queue in the same event loop, which again will trigger a transmission. The event loop basically works as follows:

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

For the event loop to be active there needs to be waiting tasks in the queue or active network read handlers. Live555 uses the `select()` system call to monitor the network read handlers, something that matches well with our design.

An application generally consists of a sink, a framer and a source. By calling `start-Playing()` from the sink object, tasks will be generated and added to the event loop. The video sink is the structure that transmits data over the network. It requests a video frame from the framer, which again fetches a complete frame from a video source. Both the framer and the video source are regarded as data sources, and respond to the function call `getNextFrame()`. The function triggers a pure virtual function by almost the same name, `doGetNextFrame()`. Each module's implementation of `doGetNextFrame()` works by arranging for a call back function to be triggered from the event loop when new data becomes available for the caller.

In other words, data passes from sources to sinks. The sink module is to receive data from a source, and for some sinks, for example an RTP sink, it involves transmitting packets over the network. A file sink on the other hand will write the data to a file. Likewise, a source may have to read from a file, or read from the network, or simply from another source.

6.1.2 Library description

The code includes the following libraries, each with its own sub directory (taken from the live555 website):

- **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 receiving) 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.

The library code also provides a selection of test programs, including a simple RTSP server and client.

6.1.3 Difficulties

One of the difficulties of using live555 is that the library is good for use, but not well commented. There is good online documentation [35] of the library structure, as well as descriptive class- and variable names. However, this does not remove the need for commented code that explains why certain design choices have been made, or what measurement is used for in example a time variable. In other words, the lack of well documented variables and functions made the library initially hard to read and get a good overview of.

6.1.4 Benefits

One benefit of the live555 library is that it is well tested. It is included for RTSP handling in open source media players, like VLC and MPlayer. The code is thorough, and uses a strict hierarchy of classes that is well designed. The library developer, Ross Finlayson, also has a development mailing list where questions regarding use of the library can be asked. The test programs also provide good support in modeling usage of the library. The structures supported by the modules in the library also match well with the requirements we set for developing our server and client streaming applications.

In our implementation of CHB, live555 had many structures that could be used directly. Out of the main libraries, only the liveMedia library was necessary to modify and extend for our purpose. Functions that could be directly used were those of RTSP session setup, including SDP, and RTP streaming, with framer support for TS sources.

6.2 Implementation of the CHB Server using live555

With live555 an RTSP server is set up with a link to a multicast session, referred to as a *ServerMediaSession*. *The ServerMediaSession* points to one or more subsessions. These subsessions are mainly created for formats like PS, where audio and video are transmitted on separate channels. Instead, we wanted to transmit segments on separate channels. To do so, we could create segments of the movie (as separate files) prior to starting the server. Alternatively, we could create support for splitting up TS files into segments by the use of multiple file descriptors pointing to respective segment offsets. We decided to implement the latter. That way, we avoid preprocessing prior to starting the server application. We could also let wait time be a start up parameter that could easily be changed without any further implications. To accomplish the above we needed a structure to create segments, and a structure to handle the segments as a video source.

6.2.1 Creating segments

A TS file would in live555 be read through a *ByteStreamFileSource* that would take care of the file descriptor and reading from the file. To handle the same for segments we set up the following structures described in the sections below:

Helper classes

HarmonicHelper.cpp has the source code for two helper classes, *HarmonicFile* and *HarmonicSegment*, and their functions.

HarmonicFile is a helper class. It finds the duration of a video and calculates the file offsets to create file descriptors that point to the start of a segment. In our design, we assumed a constant bit rate. To find the correct offsets we did the following:

- Found the movie duration by using a pregenerated reference file, TS index file (made with live555's *MPEG2TransportStreamIndexer* program). The duration, if known, could have been included as a start up parameter. We chose to keep the generation of the index file separate from the program as it was time consuming, and would cause a delay in starting the server. There could be other benefits of generating an index file, which we will mention in chapter 8.
- Calculated the number of segments, *duration / wait time*.

- Calculated the number of TS packets (of 188 bytes) in the file, $file\ size/188$.
- Calculated the number of TS packets in a segment, $number\ of\ TS\ packets/number\ of\ segments$.
- Calculated the number of bytes in a segment, $number\ of\ TS\ packets * 188$.
- Calculated the segment offsets, $segment\ number * bytes\ in\ a\ segment$

The Harmonic file generates *HarmonicSegment* objects, passing along the corresponding segment start offset, and the end offset.

HarmonicSegment is the second helper class. It creates an instance of the *HarmonicByteStreamFileSource*. By knowing the start offset, the *HarmonicFileSegment*s opens the video file (TS file) and seeks to the start offset. When creating a *HarmonicByteStreamFileSource*, this file descriptor is passed along together with the start and end offset.

Handling looping and timestamps

HarmonicByteStreamFileSource.cpp has the source code for the *HarmonicByteStreamFileSource* class. It is a subclass of the *ByteStreamFileSource* used to read TS files in live555. Different from its parent class, the *HarmonicByteStreamFileSource* will not close when reaching the end of a file. Instead, *HarmonicByteStreamFileSource* will seek back to the segments start point once it reaches the end of segment block in the video file. This looping of a segment in a file is a necessity to continuously stream the same segment over a channel in an CHB implementation. To perform the looping a new callback function for the sources was added to the library, `handleReset()`.

For the client to recognize the start of a segment, we decided in our design to use the RTP timestamps. To enable this, the timestamp needs to start at 0. This is against the recommendation of the RTP protocol [27], which is to start with a random timestamp. However, we needed a value that could be used to recognize a beginning, something 0 does. As a result, prior to looping a segment, it is desirable to transmit any remaining data buffered in an RTP packet, before resetting the time stamp. If not, an RTP packet may contain data from both the end of the segment and the beginning. We want to avoid frames from the beginning of for example segment s_i at the beginning of segment s_{i+1}. The `handleReset()` function does exactly that. The timestamp, on the other hand, is generated by the *HarmonicByteStreamFileSource* object. This will automatically set back to 0 when reading from the start offset of the TS file segment. For a brief overview of the implemented functions and their collaborations with main functions in other classes, refer to figure A.1 in the appendix.

Last, a small function returning the segment number of the file source was added for scheduling purposes. A further description is found in section 6.2.4.

6.2.2 Establishing client sessions

The *ServerMediaSession* takes care of generating a session description (SD) sent to the client upon an RTSP request. To inform the client application about the streaming, we decided to make a change in the common use of SDP. SDP generally indicates the session type, for example broadcast or meeting, in the attribute (a) type field. As CHB indicates a special streaming scheme, we decided to use this field to specify that periodic scheduling is in use. We use the term harmonic, named after the algorithm. The SDP attribute fields also contain control information indicating a track number for each channel to be streamed over. As the segments are added to the *ServerMediaSession* in an orderly matter, the SD uses the track number to indicate the segment number.
The c field contains connection data, in our case the multicast address which to listen to.
The m field following gives the client a media description of the segment. The description includes the protocol to use for streaming, which multicast port to listen to, as well as a format description. The format is in our case MPEG-2 TS, which is indicated by the number 33. See figure 6.1 for an example of the SD.

```
v=0
o=- 1226703493445253 1 IN IP4 192.168.101.230
s=Session streamed by "harmonicMPEG2TransportStreamer"
i=elephant.ts
t=0 0
a=tool:LIVE555 Streaming Media v2008.01.18
a=type:harmonic
a=control:*
a=range:npt=0-
a=x-qt-text-nam:Session streamed by "harmonicMPEG2TransportStreamer"
a=x-qt-text-inf:elephant.ts
m=video 8888 RTP/AVP 33
c=IN IP4 239.255.42.42/2
a=control:track1
m=video 8890 RTP/AVP 33
c=IN IP4 239.255.42.42/2
a=control:track2
...
```

Figure 6.1: *SD sample, included in the server's RTSP DESCRIBE response*

6.2.3 Streaming

The RTP functions in live555 take care of streaming RTP packets. The *MPEG2Transport-StreamFramer* makes sure to request complete TS packets from the *HarmonicByteStream-FileSource*, making up frames suitable for RTP over UDP. UDP supports packet sizes up to 65535 bytes. However, the underlying Ethernet has a limitation of 1500 bytes. Consequently, the RTP packets do not exceed that number, avoiding fragmentation of the packets. As mentioned in our design, splitting up TS frames can make the demultiplexing on the client side more difficult.

6.2.4 Scheduling

The *MPEG2TransportStreamFramer* is responsible for calculating the timestamps that are used for scheduling the transmission of the RTP packets containing the TS frames. We based the generation of segments on a constant bit rate for streaming. A part of this assumption was the expectation that the RTP timestamp would be used for scheduling the transmission of the RTP packets. However, as we found out late in the implementation, the scheduling in live555 for transport streams are based on an estimation calculated in the `MPEG2TransportStreamFramer::updateTSPacketDuration-Estimate()` function. By looking at the program clock reference in the TS packets, an estimate of the next scheduling is given. To differentiate the bandwidth for the segments, each *HarmonicByteStreamFileSource* has a function returning the segment number n. The segment number corresponds to the decreasing factor of the bandwidth at which the packet should be transmitted. By multiplying the estimated frame duration, the scheduling should be postponed n times that duration. As this is a method based on the the program clock reference for TS, it is best suited for movies with constant bit rate. However, many movies, including the ones we have available, use variable bit rate. We will do experiments to see if we can get an indication as to what effect the constant bit rate based transmission might have when streaming a variable bit rate movie, as well as see if we can see any consequences of using an estimate that is not necessarily accurate. Refer to chapter 7 for the results.

6.3 Implementation of the CHB Client using live555

Similar to the server side, there is support in live555 for establishing a client RTSP session to receive multiple streams with RTP over UDP. The RTSP client mirrors the server structure when establishing an RTSP session, *MediaSession*, established by the RTSP functions. The media session has subsessions that points to one file sink per subsession. Upon receiving the SD from the server's 'RTSP DESCRIBE' reply, the client sends a 'RTSP SETUP' message for each stream to the server. As a result, each segment described by the SD, is connected to a media subsession. The subsession takes care of establishing an RTP source to read the data from the network, and a TS framer to deliver the video frames to a video sink, in our case a file buffer. For a simple class overview see figure 6.2. However, there is no support in live555 to join the data from

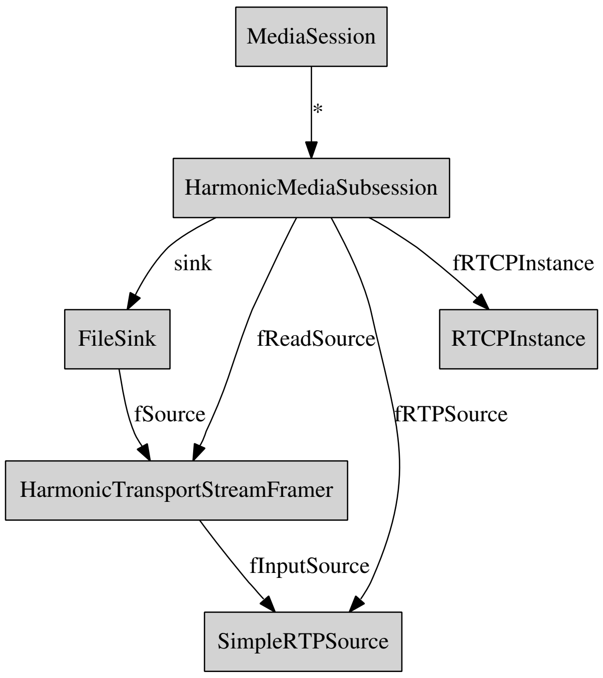

Figure 6.2: *A MediaSession object has pointers to multiple HarmonicMediaSubsessions. Each subsession has a FileSink, HarmonicTransportStreamFramer and a SimpleRTPSource, as well as an RTCPInstance. The variable names in the source code are found on the edges between the objects.*

the streams back into one TS stream or file. In addition, the *MediaSubsession* class in live555 had no support for establishing more than one sink to deliver data to. As described in our design, we decided to use two file buffers. The segmentation of the movie on the server side was done at the application level, and the joining has been designed to image this. To handle these functions we did the following changes and additions to the live555 library:

6.3.1 Handling RTSP session setup

MediaSession.cpp has the source code for the media sessions established by RTSP for the client side in live555. *MediaSession* is the one place where there are pointers than can lead to all existing file sinks (through the subsessions), and therefore the logical place to keep track of downloaded segments and merge them if desired. We altered the *MediaSession* source code to recognize CHB streaming. If harmonic is included in the SD, a structure for saving file descriptors for each file buffer is established. In our implementation all the segments are joined to one file upon complete transfer. However, the structure is made so that the file descriptors should be easy to feed to a demultiplexer in a media player by adding some support functions. In addition, instead of pointing to *MediaSubsession* objects, *HarmonicMediaSubsession* objects are created for our purpose.

6.3.2 Receiving segments

HarmonicMediaSession.cpp has the source code for the *HarmonicMediaSubsession* class. It is a subclass of *MediaSubsession* in live555, and is initiated by the *MediaSession* object based on the SD. Its only function is to set up a file descriptor to be used as a file buffer for the latter part of the segment, as shown previously in figure 5.3. A file descriptor for the former of the two buffers is initiated when the segment has looped back to start. For a simple call graph outlining the main functions in the RTSP session setup, refer to figure 6.3. For a more detailed collaboration diagram of the parent class MediaSession, refer to figure A.2 in the appendix.

6.3.3 Buffering and segment boundary recognition

HarmonicTransportStreamFramer.cpp has the source code for the *HarmonicTransportStreamFramer*, a subclass of the live555 *MPEG2TransportStreamFramer*. An important feature implemented in the class is that of recognizing when a segment has reached the end of an iteration and being looped back to start on the server side. This to trigger the switch from one file buffer to the next one. When starting to listen to the network, *HarmonicTransportStreamFramer* ensures that the RTP timestamp of the first packet is recorded as the 'start time'. The timestamp is fetched from the RTP source object. After each frame is read from the source, the timestamp is checked, and whether the end of the segment is reached is kept track of. If the segment end is not reached, and the

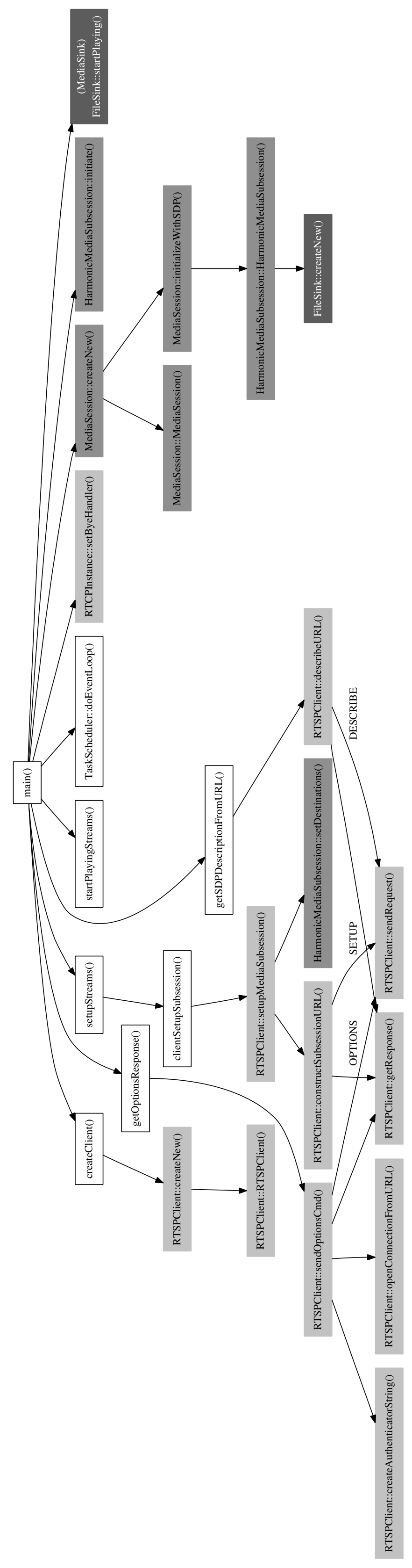
main()
(MediaSink)
FileSink::startPlaying()
HarmonicMediaSubsession::initiate()
MediaSession::createNew()
MediaSession::initializeWithSDP()
HarmonicMediaSubsession::HarmonicMediaSubsession()
FileSink::createNew()
MediaSession::MediaSession()
RTCPInstance::setByeHandler()
TaskScheduler::doEventLoop()
startPlayingStreams()
getSDPDescriptionFromURL()
RTSPClient::describeURL()
DESCRIBE
HarmonicMediaSubsession::setDestinations()
RTSPClient::sendRequest()
setupStreams()
clientSetupSubsession()
RTSPClient::setupMediaSubsession()
SETUP
RTSPClient::constructSubsessionURL()
OPTIONS
RTSPClient::getResponse()
getOptionsResponse()
createClient()
RTSPClient::createNew()
RTSPClient::RTSPClient()
RTSPClient::sendOptionsCmd()
RTSPClient::openConnectionFromURL()
RTSPClient::createAuthenticatorString()

Figure 6.3: *Simplified call graph for the RTSP setup on the client side. When a function is implemented in a parent class, the class names are indicated in square brackets.*

timestamp is less than the time stamp when starting to listen (start time), we can assume to have reached the end of the segment. In that case an added callback function to the library is called, namely `handleSegment()`. The function is implemented in the parent class *FramedSource*. `handleSegment()` calls on a function with the same name in the file sink, resulting in the file descriptor of the current buffer to be added to the *MediaSession*. The file descriptor pointed to by the file sink is then substituted by a new file buffer for the first part of the segment. The rest of the stream is afterwards written to the newly added file descriptor. Furthermore, when the RTP timestamp is equal to or greater than that of the start time and we know that the segment has looped, a `handleClosure()` function is called. When all subsessions are done, this results in a tear down of the RTSP session, and the MediaSession reassembles all the segment parts into one TS file in the /tmp/ folder on the computer. The file name is currently harmonic.ts. Pseudo code of the process above is found in 6.1. For a graphic overview of the function calls using *HarmonicTransportStreamFramer*, refer to figure 6.4

```
RTPSource * rtpSource = (RTPSource *)fInputSource;

if(segment_buffer == NOT_STARTED){  /* First frame */
  startTime = rtpSource->currentRTPTimestamp();
  segment_buffer = SECOND_PART;
} else if( (segment_buffer == SECOND_PART) &&
        (rtpSource->currentRTPTimestamp() <= startTime) ){
  /* We have reached the end of the segment. Calling a segment
     handler that will substitute the file descriptor in the file
     sink */
  handleSegment(this);
  segment_buffer = FIRST_PART;
} else if( (segment_buffer == FIRST_PART) &&
        (rtpSource->currentRTPTimestamp()  >= startTime) ){
  /* We are back to the start of the segment, and the two segment
     parts should be buffered. Closing down the media subsession
     and its structures */
  handleClosure(this);
  return;
}
```

Listing 6.1: Handling segment iteration on the client side

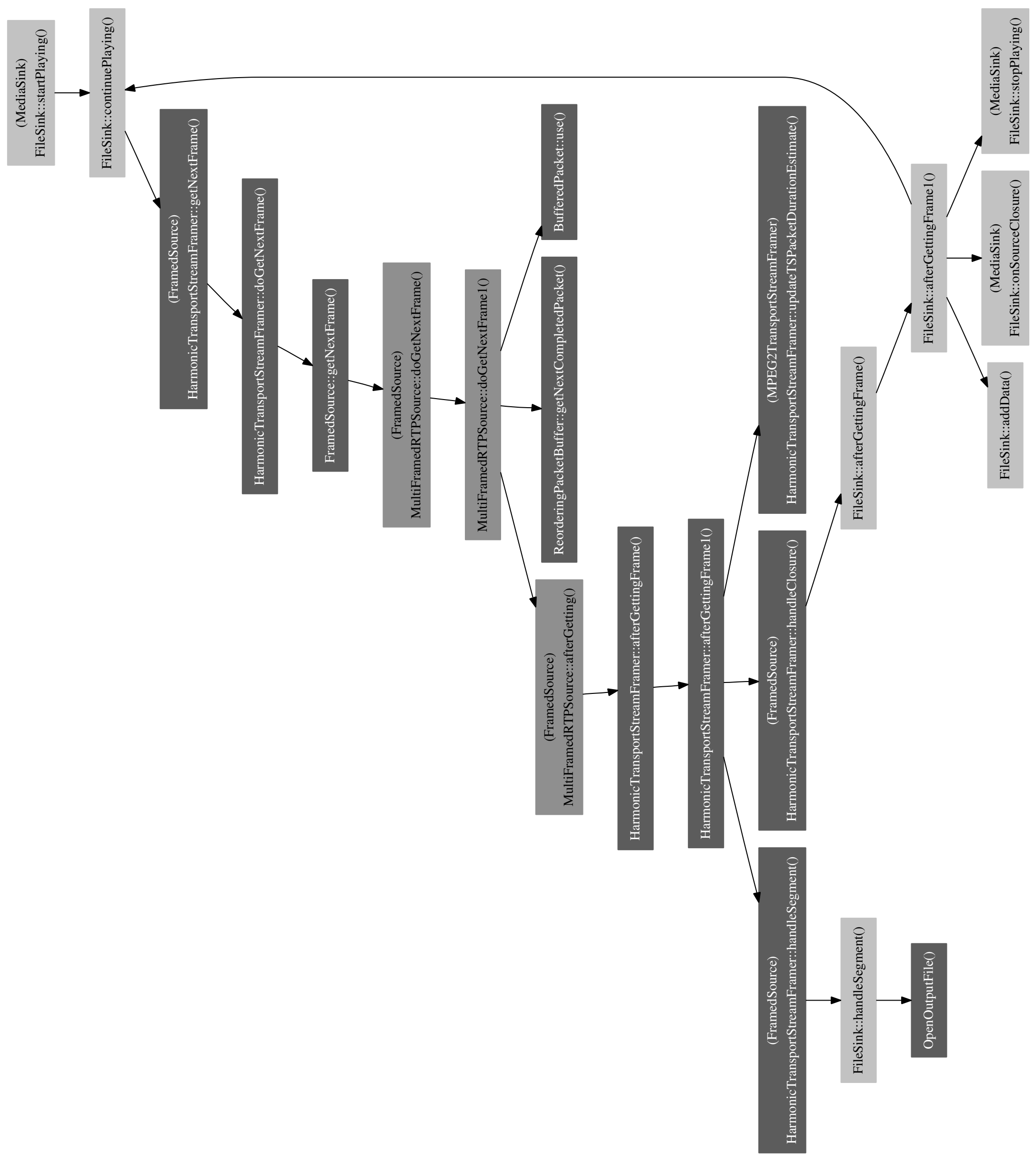
(MediaSink)
FileSink::startPlaying()
FileSink::continuePlaying()
(FramedSource)
HarmonicTransportStreamFramer::getNextFrame()
HarmonicTransportStreamFramer::doGetNextFrame()
FramedSource::getNextFrame()
(FramedSource)
MultiFramedRTPSource::doGetNextFrame()
MultiFramedRTPSource::doGetNextFrame1()
BufferedPacket::use()
ReorderingPacketBuffer::getNextCompletedPacket()
(FramedSource)
MultiFramedRTPSource::afterGetting()
HarmonicTransportStreamFramer::afterGettingFrame()
HarmonicTransportStreamFramer::afterGettingFrame1()
(MPEG2TransportStreamFramer)
HarmonicTransportStreamFramer::updateTSPacketDurationEstimate()
(FramedSource)
HarmonicTransportStreamFramer::handleClosure()
(FramedSource)
HarmonicTransportStreamFramer::handleSegment()
FileSink::afterGettingFrame()
FileSink::handleSegment()
OpenOutputFile()
FileSink::afterGettingFrame1()
(MediaSink)
FileSink::stopPlaying()
(MediaSink)
FileSink::onSourceClosure()
FileSink::addData()

Figure 6.4: *Simplified call graph of the most important functions with an RTP server.*

6.4 Summary

We have implemented CHB as an RTSP streaming server and client. SDP is used to describe the type of transmission in use, which we called harmonic. The server is set up to multicast the segments as RTP streams over UDP, using the CHB scheduling scheme. In the SD sent as part of the RTSP streaming initiation, all information for the segments in the video are described, including channel and segment number. The clients receive the streams through separate interfaces, and buffers the data in files. The client session has a structure to store file descriptors pointing to the beginning of each file buffer that can be used to deliver the segments to a demultiplexer in the correct order.

We will now look at some experiments conducted to evaluate the performance of the streaming application.

Chapter 7

Experiments

To evaluate our implementation, we have performed a selection of tests and measurements. The purpose has been to investigate the performance of the CHB algorithm through the implemented application. In this chapter, we describe the test environment and setup, in addition to describing our results.

7.1 Test environment

For the experiments, we set up an emulated network consisting of a server, an network emulator, a switch and several client machines. Only one client was needed for our testing purposes. For specifications on the machines, refer to the description in section A.2 in appendix A. For a graphical overview of the test setup, refer to figure 7.1

A switch can not route multicast packets, but rather broadcasts them to all connections. The application was set to multicast the video content. Consequently, all packets from the server were transmitted to all clients connected to the switch. This made it impossible to listen seclusive to the communication from the server to one specific client. However, it did make it possible to measure the performance over the network at all times the server was running. For network emulation, a computer was set up with a bridge, using the netem application to emulate a network with delay and loss. All communication between the server and the network was streamed through the network emulator. Bash and python scripts were used to automate the testing processes, as well as the python gnuplot extension for generation of graphs to analyze the network traffic. To capture network traffic, tcpdump was used. Following, scripts were

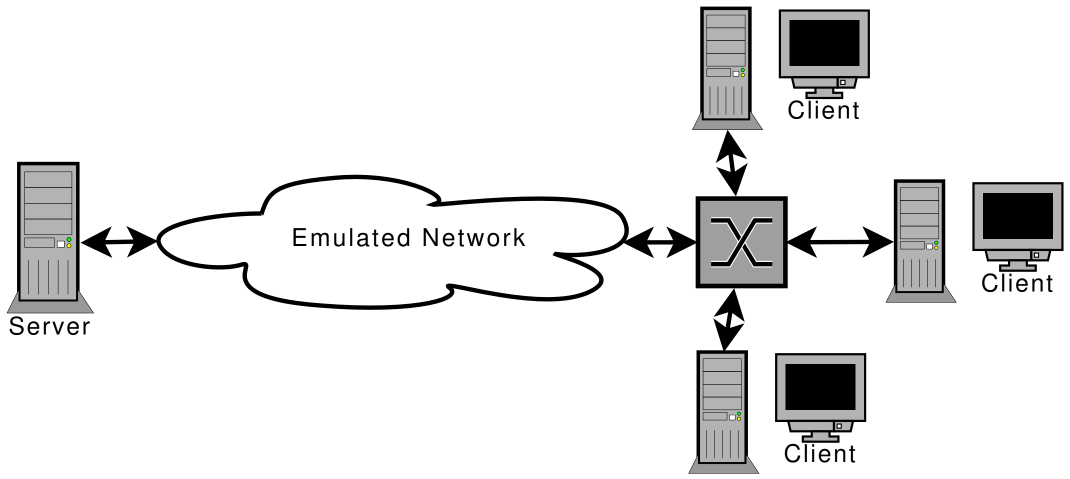

Figure 7.1: *Test environment*

run to create log files with the desired content for easy input to gnuplot.

Emulating round trip delay and packet loss

When evaluating the CHB algorithm, it was of interest to see how it performed in a realistic network. In our case, it is of main interest to see how well it would perform locally in Norway. Seeing how these services often follow the ISP, and knowing that autonomous systems do not always allow for multicast across their borders, we decided to test under the conditions that is common for a normal Norwegian ADSL user (see appendix A).

To measure what is a realistic network in Norway, we measured the round trip time (using ping) from a private Telenor ADSL connection in Oslo to several hosts throughout Norway. For the specific locations and measurements, refer to table A.1 in the appendix. The results showed a round trip time from 53 to 77 milliseconds, with an average just below 65. The packet loss was generally at 0%, but we did experience up to 1%. We have therefore chosen to perform the test under ideal conditions of no delay and no loss, a 32.5 (*round trip time*/2) millisecond delay, and a loss spanning from 0% and up to 1% with 0.25% increments.

7.2 Testing the scheduling performance

In order to test the performance of the actual application, we set up a number of tests. All tests were performed with the same video file, “Elephants Dream” [36], which is a 654 second long movie. We also used “The Wrong Landing” by Chema Hernandez [37] for shorter tests, as it is only 270 seconds long. For the test results reported in this thesis, the former of the two videos were used. We set the maximum waiting time for a client before consumption to 60 seconds.
We did the following tests, using tcpdump for the generation of log files for our analysis.

7.2.1 Start up delay

Description

The CHB algorithm assumes that streaming on all channels can be triggered at the same time. However, the test environment used a computer with one CPU, processing tasks in a sequential way. As a result, there was expected a small scheduling start up delay for the streams, increasing with the number of channels. We wanted to measure how big the delay was, to check if it had any impact on the consumption at the client side.

Results

In figure 7.2, we see the initial gap between the segments for the very first packet sent from the server upon receiving the packets at the client side. The network settings for this result are a round trip time of 65 milliseconds and 0.25% packet loss. In our tests, we found that the figure is representative also for the other network settings (as described in the test setup), as the network delay is not taken into consideration since all it does is add to the wait time. It does not affect the scheduling itself since UDP packets are transmitted despite loss or congestion. It was still interesting to see how the scheduler performed, and if there were cases where there was a greater difference in the start up time of the various streams (for the same movie transmission). Due to how the different segments were added to the scheduler, the last segments were scheduled first (as shown in 7.2 where segment 11 in corresponding stream 11 is transmitted prior to stream 1). This ensures that the algorithm works, even if it had been depending on

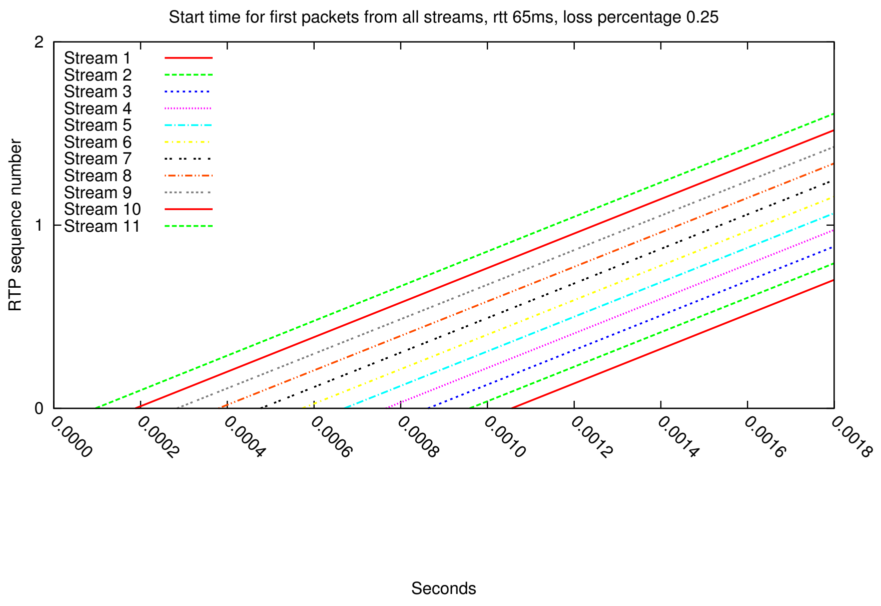

Figure 7.2: *Start up delay: The initial scheduling delay between the movie segments for the initial first (and second) packets for all streams at the client side. Note that stream 11 is transmitted prior to stream 1.*

the initial scheduling. However, because both initial segments are sent at full bandwidth, the order at initiation could have been according to the segment number. This is because the algorithm makes sure the client always has a full segment in the buffer, except from the very first one. The exact differences in microseconds are displayed in table 7.1. It is at most 4.283ms, and small given that we should have the length of a segment at 60 seconds in our buffer most of the time. The network delay in this test does not affect the client in other ways than adding to the wait time, unless the interarrival time between packets differ. Consequently, our tests indicate that the algorithm can be said to perform without risk for missing deadlines at the client side when using a 60 second wait time for our test movie.

7.2.2 Transmission to one client

For these tests we wanted to investigate to what extent the server varied the bandwidth for the different segments. We recorded the receiving time for packets at a client while receiving an entire movie. We did two different analysis based on the logs generated.

	No delay	Delay 65ms				
Stream	**0 %**	**0 %**	**0.25 %**	**0.5 %**	**0.75 %**	**1.0%**
Stream 1	0.001050	0.000946	0.000960	0.000969	0.000953	0.004283
Stream 2	0.000953	0.000848	0.000863	0.000871	0.000857	0.004186
Stream 3	0.000857	0.000754	0.000767	0.000773	0.000760	0.004090
Stream 4	0.000761	0.000656	0.000670	0.000677	0.000664	0.004004
Stream 5	0.000664	0.000559	0.000574	0.000580	0.000567	0.000580
Stream 6	0.000568	0.000463	0.000477	0.000484	0.000470	0.000486
Stream 7	0.000471	0.000371	0.000381	0.000388	0.000376	0.000386
Stream 8	0.000374	0.000270	0.000284	0.000292	0.000279	0.000291
Stream 9	0.000276	0.000176	0.000188	0.000194	0.000184	0.000187
Stream 10	0.000180	0.000088	0.000092	0.000098	0.000085	0.000100
Stream 11	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Table 7.1: *Initiation delay (in seconds) for all streams at the start up time of transmission measured at a client*

Description of part 1

The first analysis was done based on data from a client waiting for the video at the initiation time of the server. This ensured that the RTP timestamp for the first packet on all channels would be 0. This made it easier to create a graphic display of all the streams, and to display their arrival at the client, where the x axis would be the arrival time, and the y axis the RTP timestamp. The timestamps used at the client in the tcpdumps are those on the client computer, written as seconds.

The RTP time stamp indicates how far in the video segment the stream is receiving (under the assumption of constant bit rate). The RTP timestamps will increase up to the same value at the completion of the download. The seconds since epoch shows the arrival time at the client. A difference in angle between the graphs showing the first stream and the following streams is an indication of decreasing transmission frequency (of the same number of bytes), and also decreasing bandwidth as the segment sizes are the same. Common for both tests was that the RTP timestamp was used in correlation to the arrival time at the client machine. Consequently, the higher the bandwidth, the steeper the curves appear.

Results for part 1

As expected, the benchmarks showed that the first two initial segments were transmitted at the same rate. As seen in figure 7.3, a fan shape is formed, underlining that the segments are indeed transmitted at their decreasing bandwidth (as the angle is increasing). It is, however, also evident that the fan is not perfect. We believe this irregularity is caused by the issue of variable versus constant bit rate. A more thorough reflection of this issue is found in chapter 8 (Discussion).

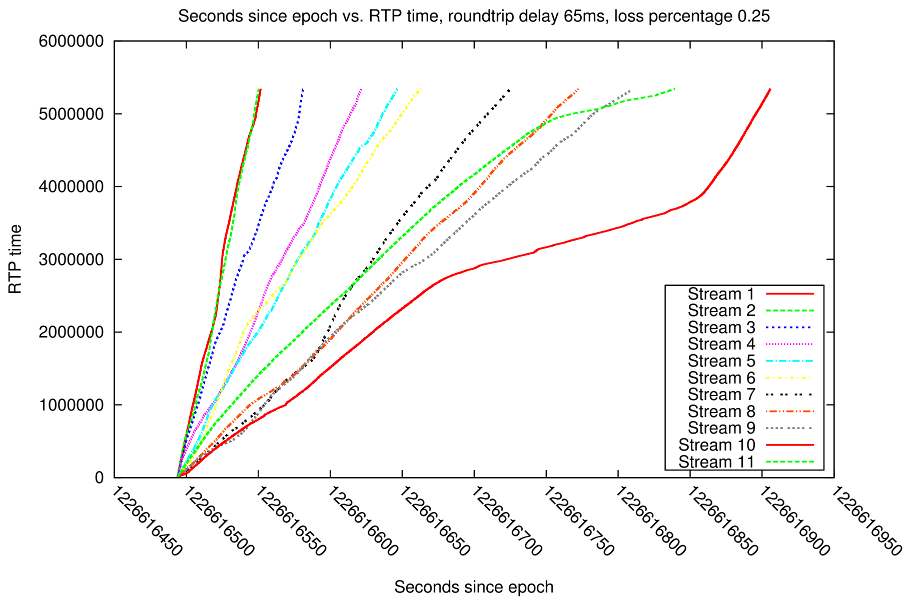

Figure 7.3: *Transmission to one client from the start up time of the server until completion of one iteration of a movie*

Description of part 2

The second analysis was based on logs from a client that started to listen at an arbitrary point after the server had been running for a while. The logs from such a transmission were expected to show the client starting to listen in the middle of a segment, and would stop upon reaching the same RTP timestamp in the stream after an iteration of the segment on the server side.

Results for part 2

For the the results that started streaming from an arbitrary point (see figure 7.4), we have added an offset to the RTP timestamps to avoid the graphs to overlap each other. We have removed the RTP time stamp, but the values for each stream are corresponding to those of the previous test. In figure 7.4, we can observe that the first and second segments are not being transmitted in parallel as one would expect. This can be caused by the initial delay discussed in test 1, though one would expect a smaller difference that what we see in our test. Another possible cause is that the scheduling is based on variable bit rate, whereas the timestamps assume constant bit rate. As a result, the segments are not seen as same length time wise as falsely indicated by the RTP timestamps. The segments are of equal size byte wise. More about this issue in chapter 8. Despite this issue, we are still able to observe that the client receives the entire segments according to the RTP timestamps. In the graph below, we can read that the start of segment one is received by the client approximately 50 seconds after starting to receive the stream, and the segment as a whole is received within the next 10 seconds. The second segments appears to be received within the same time frame as expected.

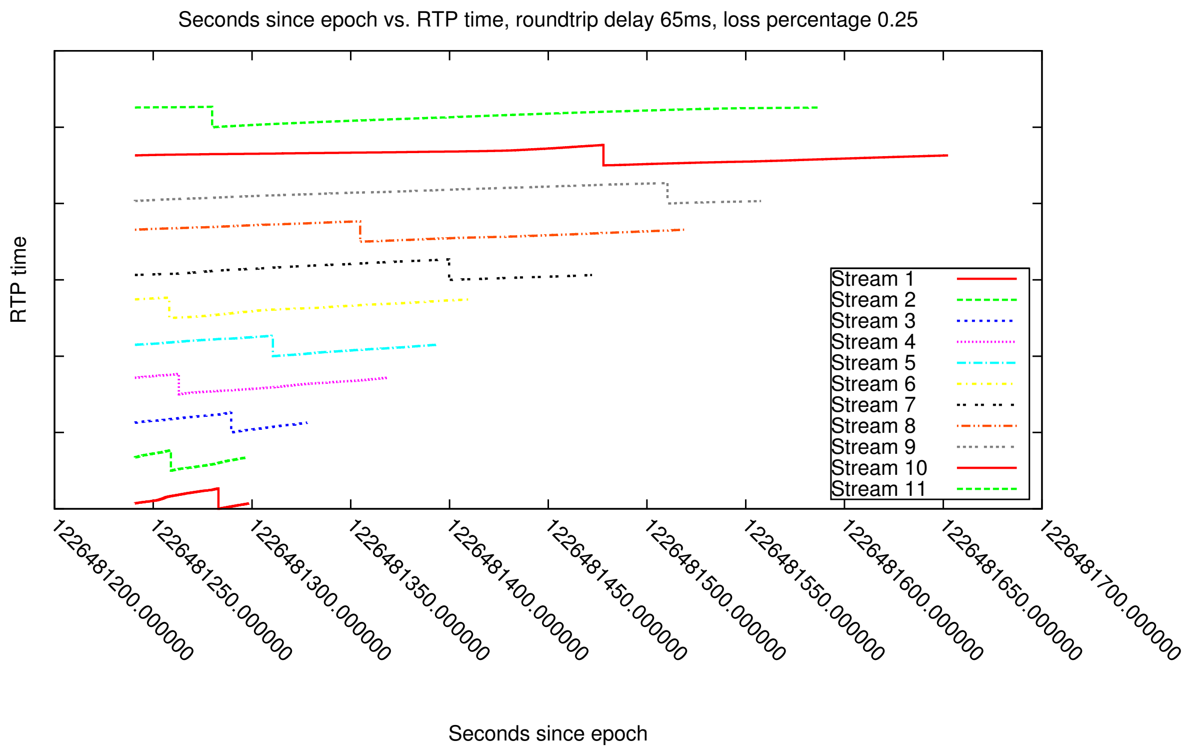

Figure 7.4: *Transmission of one full movie to one client at an arbitrary start time. The fall in the graphs indicate the place where a segment is looping, making it necessary for a client to reassemble the segment on a specific channel, in addition to reassemble the segments prior to consumption.*

We found it interesting to see how the CHB works in practice in comparison to the theoretical model. The theoretical model assumed that the download of a segment would start only at the initiation of a new iteration of a segment. However, in practice the client might just as well download and reassemble the segment. In the theoretical alternative, the client would have to stay in listening mode to be able to determine if a packet was indeed the beginning of a new iteration, and consequently be using bandwidth in doing so. By downloading immediately the client ensures even better that all segments are available upon time of consumption, and makes better use of the available bandwidth during the initial wait time at the client side. However, the client will need to have enough available space on disk to buffer the file segments.

7.2.3 Transmission of several iteration of the movie

Description

We were interested in how the scheduling performed after transmitting the same movie repeatedly. To measure this, we set up one client to be in listening mode for the duration of 2100 seconds, which corresponds to over 3 iterations of the entire movie at 654 seconds. Capturing the network traffic we were able to measure how the scheduling of the repeated segments was performing. The result can be seen in figure 7.5. In the figure an offset has been added to each stream so the lines do not overlap in the graph. That way it is easier to see how the streams are scheduled in comparison to each other.

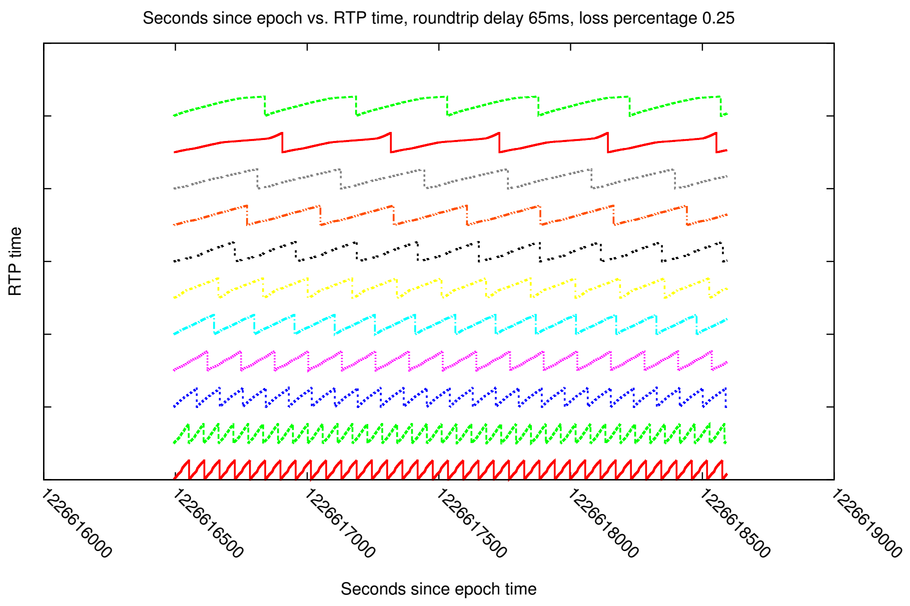

Figure 7.5: *Transmission of several iterations of a movie to one client from start of the server. Stream 1 is at the bottom (red).*

Results

For a graphical overview of the transmission of the iterating streams, refer to figure 7.5. A segment start is found just after the drop in the graphs, as then the RTP time is reset to 0. From there the time stamp increases. We see this as a wave pattern in our graphs, where the drop is the indication of the time the seek back to start has occurred. As before, the steeper the curve grows, the greater is the bandwidth. From the graphs we can see that one iteration of the two last segments takes less time than the time needed to consume the video. The maximum wait time is set to 60 seconds, and the time needed for consumption is 654 seconds. This means that the client, under the assumption of constant bit rate) may use just below 714 seconds to download the video, and still have the segments in time for consumption. However, as seen in the graph, the download takes approximately 425 seconds. The segment that takes the longest to download is a direct measure of the total download time. The relationship between the actual download time and the time required to receive the entire movie must be below 1 to indicate that the movie has been received on time. We get the following formula:

$$\frac{\textit{download time}}{\textit{wait time} + \textit{movie duration}} < 1 \tag{7.1}$$

$425/714 = 0.6$ indicates that our movie was received within approximately 60% of the time needed for consumption. Having a great portion of the movie downloaded prior to the consumption time, the client needs a substantial amount of disk space for buffering. With today's computers we do not see this as a great challenge, but found it worth pointing out as a requirement. To give a more clear example: If a video has wait time equal to its length, only one segment will be needed. If starting to listen to the stream just a few seconds after an iteration occurred, almost the entire movie will have to be buffered prior to consumption. If not starting to download and buffer until the start of the first segment, there is less need for buffer space, as consumption of the first segment will happen immediately. If wanting to download from the time the client starts to listen, which can be at any arbitrary point in the iteration, a greater need for buffer is required at the client side. A greater buffer may, however, create support for limited Trick Play functionalities (like rewind and pause). The client is also less vulnerable for jitter and unexpected delay in the transmission.

When the segments are described in the algorithms, they are supposed to be transmitted at the same time, and iterate so that the segment at times becomes aligned. From figure 7.5 we can easily observer that this is not the case when implemented with segments that cannot be triggered at the exact same time. She scheduling also skews this pattern. There is one benefit of this difference from the theory to our practical implementation, that of peaks in the bandwidth. If the iterations had occurred as described mathematically, there would be times at which all the segments would be back to start, attempting to transmit at the exact same time. This could cause a peak in the bandwidth, and possibly not supported at the client end. By having segments that are skewed in the temporal alignment, we avoid these peaks to a greater extent.

7.2.4 Scheduling performance based on client wait time and number of segments

Description

Wwe wanted to check if movie transmission were downloaded within the guaranteed time. The guaranteed download time is the duration of the movie, in addition to the maximum wait time. In the section above we described a download indicator, the fraction of the guaranteed download time and the actual download time. We will refer to that fraction, given as a decimal number, as the 'download indicator'. The download indicator is defined by the equation given in the section above, 7.2.3. If the download indicator is above 1.0, it indicates that the download took longer than the guaranteed time. That means the video arrived sower than that of the consumption rate. If the download indicator is below 0 it indicates that the video was received without causing any consumption delay.

Knowing the the wait time has direct influence on the number of segments, we also wanted to check the download indicator in comparison with number of segments.

To determine the download indicator for different wait times and different numbers of segments we used both our test movies. It should be mentioned that these two movies have different resolutions as well as bandwidth, so the numbers may not be comparable.

To measure the download time of the segments we We defined the download time as following: *The download time is the time between the RTSP PLAY response from the server reaches the client, until the client send an RTSP TEARDOWN message*. To do so we captured the network traffic while downloading the movie in a 0 ms delay, 0% loss test environment. We used a variety of wait times, splitting the number into different numbers of segment. Refer to tables A.2 and A.3 in the appendix to see the metrics.

Results

Looking at figure 7.6 we see no patterns between the two movies. The only correlation is that the first two segments both arrive in half the guaranteed time, or less. This is expected from the scheduling scheme.

However, if looking at figure 7.7, we see a similar pattern form for the two videos. The statistical material is not sufficient to draw any conclusion. The graphs indicate that the

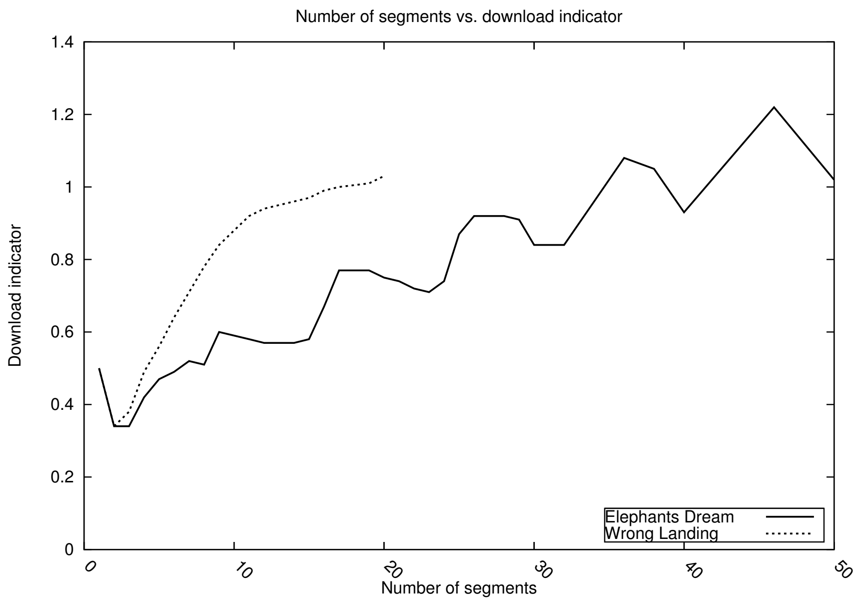

Figure 7.6: *Number of segments vs. download indicator*

smaller the wait time, the greater the download indicator. However, CBH was made to always transmit the segments on time. A small wait time also indicates small segments. Many of our other tests have pointed towards the variable bit rate scheduling estimate in live555 as a possible cause for unexpected results. This seems likely also for this test. It can appear as if the smaller the segment, the harder it is for live555 to make a proper scheduling estimate. However, like our other tests, further investigations with different length movies, preferably with the same playback bandwidth, would need to be tested with.

7.2.5 Actual bandwidth

Description

We wanted to check the actual bandwidth of the transmission, to compare if to the estimation we made in our design, seen in figure 5.2. To measure this we calculated the bandwidth based on the received bits per second, measured from the number of bytes transmitted from the server within a time frame of 450 seconds.

Results

In figure 7.8, a graph over the actual bandwidth of the transmission of the video is shown. We note that the bandwidth indeed varies by a considerable number. The

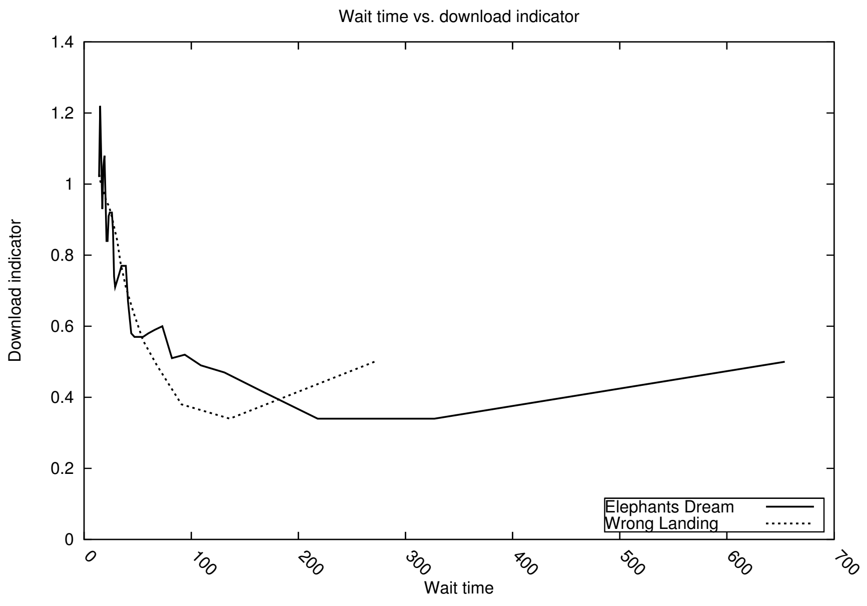

Figure 7.7: *Wait time vs. download indicator*

averages are as shown in table 7.2, and the variation in bandwidth is displayed in the graph in figure 7.8. Note that graphs are only shown for a selection of the streams. Also note that the measured average is based on the transport layer packets, including protocol overhead.

	Stream 1	**Stream 4**	**Stream 7**	**Stream 10**	**All streams**
Measured Average	1247	557	300	176	5991
Theoretical average	1100	367	183	122	3872

Table 7.2: *The measured and theoretical bandwidth in kbps for our implementation of the CHB scheduling algorithm. The measured bandwidth includes that of protocol overhead.*

We have not found a clear reason to why the bandwidth is so much higher than that estimated. The transport and application layer protocols do cause a certain level of overhead. The RTP payload is always a multiple of 188 (the byte size of a TS frame). Thus, this also adds some overhead. However, the mentioned overhead does not measure up to a number that would indicate the measurements we are observing.

A movie of 654 seconds, divided into 60 second segments should result in 11 channels. The bandwidth for the last channel should then be equal the frequency of 10 times the length of the segment. That is approximately 600 seconds. However, in figure 7.5, we found the download time for our movie to have been closer to 425 seconds. We draw the conclusion that there are indication of the segments being scheduled for faster transmission than necessary. There is a possibility that this is caused by the estimation

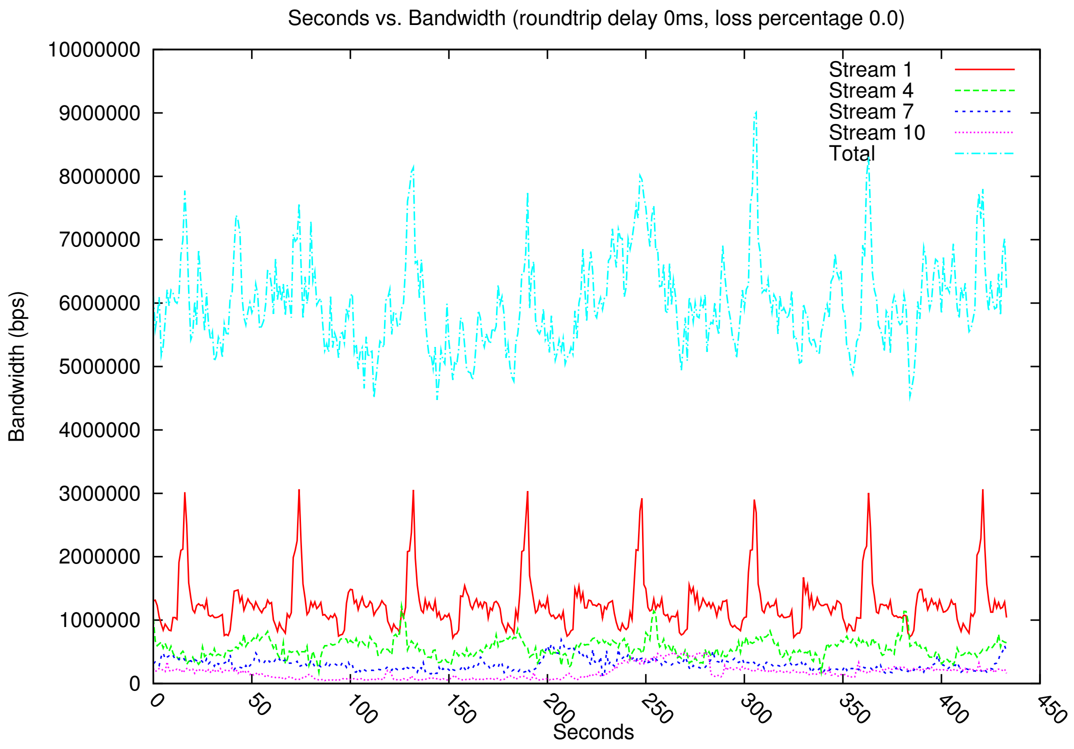

Figure 7.8: *Transmission of the movie from the server for the duration of approximately one iteration of the movie .*

used to schedule the transmission of the packets. However, further research is needed to determine if that is the cause.

7.3 Video quality at the occurrence of packet loss

Discuss difference of VLC and MPlayer... and the need for a media player that handles lossy streams. As most media is still sent by TCP, not all current media players take this into consideration (including VLC, but excluding MPlayer).

Description

We wanted to see how a video file streamed with our implementation performed in the event of packet loss in the network. We were in other words interested in the quality of experience for an end user. For packet loss we used the same network delay and loss as for all the other tests. We simply started the server and had a client download the same video for all the network setting combinations. The client saved the video for further assessment.

There are several ways to measure quality in video. Well known are the objective quality metrics Peak Signal to Noise Ration (PSNR) and Structural SIMilarity (SSIM). Both PSNR and SSIM seek to measure the similarity between two images, where one image is the original image, and the other is the one being compared.

PSNR is perhaps the most widely used objective video quality metric. It is based on a pixel by pixel comparison of an image [38]. PSNR is usually expressed in terms of the logarithmic decibel scale, where the higher the number the better result of the comparison.

SSIM seeks to compare three main metrics in images, that of luminance, contrast and structure [38]. The SSIM index is a decimal value between 0 and 1. A value of 0 would mean zero correlation with the original image, and 1 means the exact same image [39].

Results

PSNR and SSIM both require temporal alignment of the two videos. Temporal alignment is quite a strong restriction and can be hard to achieve in practice, even for shorter clips [39]. In a lossy environment this is a problem, as even few packet losses can skew the alignment, and give a wrong impression of the end result. Our experience from our initial results in table 7.3 indicated that the objective metrics were inaccurate for reflecting the user perceived quality. We did SSIM and PSNR calculations based on the entire video file, in addition to only the first 60 seconds. The results showed a big difference in the results, an indication that lack of temporal alignment can be a possible cause. We therefore chose to not do any further measurements using objective metrics, but finding better methods for measuring quality of experience.

Only one sample of the movie still seems to indicate that there generally is a decreasing quality the more loss experienced in the network.

In addition, the test shows that even in a lossless environment the received video does not equal the video transmitted. We believe this can be due to the possible overlap in frames when joining the two halves of a segments together. Another possible cause is errors in the video file, specifically missing synchronization bits in the TS header. The live555 library then drops frames as a result.

	No delay	**Delay 65ms**				
Method	**0 %**	**0 %**	**0.25 %**	**0.5 %**	**0.75 %**	**1.0%**
PSNR full movie	63.194	29.575	15.750	19.027	18.684	15.599
PSNR 60 seconds	100.000	86.932	12.480	29.620	27.544	12.422
SSIM full movie	0.915	0.690	0.406	0.528	0.513	0.396
SSIM 60 seconds	1.000	0.967	0.311	0.816	0.735	0.306

Table 7.3: *PSNR and SSIM values based on the received video files*

7.4 Summary

Through our testing we have showed that we have a working prototype of an implementation of the CHB stream scheduling over a multicast protocol. The transmission of the segments are occurring more rapid than estimated. We have proposed that the reason behind this lies in the estimation of the duration of TS frames in live555 used to calculate the scheduling. We also found that the prototype does not always download the movie in time for consumption. The reason for this is unclear, but we would like to investigate if the variable bit rate scheduling can be a cause. We recommend implementing segments that are split up in equal time lengths instead of by byte size to further investigate this issue.

Observing several iterations of the streaming segments we observed that the nice temporal alignment of the segments in the scheduling scheme does not reflect that which we experience in real life. This is due to all segments using the same CPU, and gives a skewed temporal alignment. A benefit is that peaks in the transmission, and thereby also the bandwidth, are less likely to occur.

In our assessment of the quality of the received video we found that objective metrics to measure quality of experience is lacking. PSNR and SSIM give insufficient indication as to how an end user might experience a video transmitted over a lossy network.

We have looked at and analyzed various performance results. In the next chapter we will discuss some of the issues observed in more detail. We will also look at other issues we found of interest throughout our work on implementing our prototype CHB streamer.

Chapter 8

Discussion

Through our study of scheduling techniques, our application design, development and testing, we have gained valuable experience and insight to issues that arise when implementing the CHB periodic broadcasting scheme. We will now discuss our experiences, both with direct connection to our development and experiments, but also the broader picture of deploying VoD in an internetwork.

First, we will discuss the scheduling performance of our prototype, and then the lack of fault tolerance. We then move on to discuss network and channel related considerations that can be made. Last, we discuss optimization possibilities and general usage of a service based on a CHB streaming application.

8.1 Issues when creating disjoint segments

In our experiment, transmitting to one client (as seen in section 7.2.2), we saw irregularities in the bandwidth consumption. If the bandwidth had been stable we would have experienced less fluctuations. The timestamps used in the graph 7.3 were based on the RTP timestamp that was generated under the assumption of a constant bit rate transmission. However, the live555 library provided a calculation of the estimated duration of all the TS frames transmitted in one RTP packet, and used the estimate as its base for scheduling the transmission. Where our timestamps were based on a constant bit rate, the actual scheduling of the packet was based on the assumption of a variable bit rate. Despite the scheduling timestamp being multiplied to fit that of the send out rate for the respective channels, the transmission rate for segments was clearly higher than the theoretical calculations would indicate, leading to increased bandwidth consumption.

We believe that the conflicting assumptions, with regard to bit rate, is the cause for the unexpected bandwidth measured. In live555, the calculation of the duration is an estimate. Further testing of the scheduling would be required to determine how accurate the estimation is, and what effect it has on the transmission. We can not guarantee that the video frames are received in time when transmitting a variable bit rate movie with constant bit rate. A better way would be to split the movie into segments based on the time, and not blocks of TS frames (bytes). The index file mentioned in the implementation chapter contains such information, and the library also contains functions for deriving the byte offset of a given time in the movie. In other words, it should be possible to base the transmission on variable bit rate with some modifications to the code. It is uncertain if this also affects the scheduling that is currently causing a higher transmission rate. Further investigation is required to determine the effects of this, and is thus left as further work.

8.2 Fault tolerance

Congestion in the network can lead to packets being dropped by routers. As a result, the end user may experience loss of data. A lossy environment is a reality that streaming applications need to take into account. We will now look at how loss can affect the end users' experience of a VoD session.

8.2.1 Performance in a lossy environment

In a lossy environment with up to 1% packet loss, we experienced a significant amount of disturbance when playing back the received video. PSNR and SSIM did not reflect this drop, in the experience of the video quality. Finding ways to measure the quality experience using objective metrics is currently a challenge.

Transport Streams have support for error correction which adds fault tolerance. This is accomplished by adding a forward error correction field to the packet. The transport error indicator in the TS header flags the error as unrecoverable by the demodulator. Error correction like this seeks to fix errors in existing TS frames, and not make up for missing ones. Consequently, there is a need for encapsulation formats and codecs that compensate for the effects of missing TS frames.

8.2.2 Errors when reassembling segments

Consider the following scenario: A client starts listening at the movie during the n^{th} iteration on the server side. The scheduler has just primitively transmitted an RTP packet, because of a scheduling deadline, this despite the RTP packet having space for more TS frames. Once the segment loops to the next iteration, $n + 1$, the TS frames added to the RTP packets may not be aligned in the same way as the RTP packets in the previous iteration. The timestamp of one RTP packet may still be aligned with the n^{th} iteration, and have either more or less payload than that of iteration n^{th}. If such a skewed alignment is present when the client is receiving an RTP packet with the timestamp that indicates that the segment is complete, there might be a few TS frames too many, or too few. As a result, the joining point of the two parts of the segment may have repeated or missing frames.

As a result, even in a lossless environment a client may receive segments that are not identical to that on the server side. This will only be visible at the place in the video where the segments are joined.

Using the timestamps as implemented in our prototype, there is no way to avoid this loss or duplication caused by the reassembly. However, if we change the application to use the RTP extension header we have more options. An extension header can include an offset where to write to in the segment file. If the segment size is included in the SD, the segment can be mapped on the disk, and the file sink writes the frames to their correct offsets as indicated by the RTP extension header. If we are missing frames, the client can choose to listen for a longer time. If receiving the same frame twice, the client can simply overwrite the previous. It is possible to include offsets for each TS frame, as they have a constant size of 188 bytes. It is a trade off of extra overhead in the RTP header versus a small overwrite in the segment buffer. Implementing such a scheme is only necessary if the loss or duplicate frames seen in the present implementation decrease the quality of experience. Further assessment would be needed to determine if that is the case.

8.2.3 Support for resuming

CHB is a near VoD scheduling technique that has limited support for Trick Play functionality. If the client buffers already consumed data, rewind and pause could be used in the media player. Fast forward, on the other hand, will not work as skipping in a multicast stream is not supported as the stream can service several end users. The lack of skipping is also a problem in the case of failure. The cause of failure can be anything from server error, power shortage and network congestion. If the transmission is interrupted by more than a short time period, the CHB technique provides no support for resuming if the buffer is lost. The last segments are transmitted at a much slower rate than the first ones. Late segments will have a transmission time lasting almost the entire duration of the movie. If interrupted, it is unlikely to start downloading again from the same position in the segment. Unfortunately, we see no feasible solution to this vulnerability in the broadcasting scheme. Being able to download from several sources could be possible, but CHB was designed to avoid transmitting the movie simultaneously and would thus not take full advantage of its capabilities.

8.3 Network and channel considerations

The segments in our implementation were transmitted over channels created by the combination of a multicast address and the use of ports. In the Internet backbone, intermediate routers, between a source and a destination, do not make use of data beyond the network layer. Multicast group memberships manage the network layer, and do not take ports into consideration. As a result, our implementation would transmit the same bandwidth for the entire duration of the movie, from end-to-end. This is because the entire movie is transmitted over the same multicast address, with each segment identified by a unique port. Thus, clients are unable to unsubscribe from the segment channel. The CHB technique, on the other hand, is designed to decrease the bandwidth. It decreases the bandwidth rapidly once the two first segments are buffered on the client side. In our testing we had to make our analysis based on data that was receiving all the server streams continuously. Alternatively, we can use a application layer multicast overlay scheme, to filter on ports. Deploying a CHB technique for VoD without using overlay networks requires logical channels created by the use of separate multicast addresses. Sending each segment with a unique multicast address potentially requires a considerable number of addresses. If used across ISPs boundaries, this represent a challenge with availability of IPv4 class D addresses. This may already be the reason why VoD is mostly supported within ISP networks only.

One way to reduce the need for multicast addresses would be for ISPs to support Source-Specific Multicast in their networks using IGMPv3 [23]. By listening to the multicast from only specific sources (source filtering), an ISP could potentially multicast over the same addresses for several movies, as long as the server addresses for transmitting the movies are different. Given that periodic broadcasting is mainly beneficial for the top 10-20 popular movies, providing 10-20 separate unicast IP addresses is for most ISPs feasible. Given a 1 minute wait time for 3 hour long movies, this would result in 180 multicast addresses. If having 20 movies, the total need for addresses would increase to 200. Whereas without source filtering the need for addresses would be $20 \times 180 = 3600$. In the latter example, cooperation to support VoD across ISPs would be more difficult with IPv4.

IPv6 increases the address space from 32 bits to 128. With this increase, there will be a substantial number of addresses available. As such, the problem of an insufficient number of multicast addresses, even when transmitting across the border of autonomous networks, would be solved. However, ISPs may still want to limit the use of multicast across their service area.

Another solution could be to add support for handling the transport layer in routers. This would however require major change in the deployed infrastructure. Today, bottlenecks in the network are not found in the cables and transmission media, but rather in the processing power of the routers and switches [21]. In addition, in RFC 1958 [40], principles for the network layer are outlined. They include keeping the architecture simple, consider costs. Adding support for another layer would clearly make the routers much more complex. Current routers would need to be upgraded, or most likely substituted by new ones, adding substantial cost for the ISPs.

8.4 Optimization and the applicability of CHB

Periodic broadcasting protocols were developed to scale to video playback distributions following Zipf's law [21]. This was also the case for the implementation studied in section 3.1 [15]. This study concludes that the network bandwidth becomes a bottleneck for the Patching protocol which limits the service to 5 clients per minute for a three hour video at 1.5 Mbps. Periodic broadcasting, using multicast in combination with RTSP, proved to scale much better. The benefit of patching is that of true on demand delivery. Periodic broadcasting, including our implementation of CHB, will induce a wait time before playback commences. However, as far as the use of bandwidth in the network is concerned, our implementation of CHB will not use any bandwidth beyond the first multicast router if no clients have requested the video. Once the movie is requested, the bandwidth will increase, though no more than the maximum for all channels, and only along the spanning tree of group members. That being said, our test results showed that a movie playing back at 1.1Mbps, with a 60 second maximum wait time, would consume almost 6 Mbps when transmitting all the segments. It is worth noting that this scales to support all members in a multicast group. Today, major Norwegian ISPs generally offer DSL connections from 768 to 20000 kbps. From this we can conclude that it is feasible with today's technology to put a CHB streaming server into operation for a number of subscribers.

There are also possibilities for making a CHB service on demand, by sending the first segment by unicast instead of multicast. The service will be delivered on demand, and at the same time take advantage of the benefits of CHB. Such a service will increase the bandwidth linearly for the transmission of the first segment as the number of clients increase. However, it would be for a short clip equal to the duration of the wait time. To lower the stress on certain paths, the download of the unicast stream could be distributed to proxy servers or a set of streaming servers. The SD could reflect where a client was to download the multicast stream from, choosing to even out the traffic among differing routes as a means for traffic shaping. The total bandwidth usage would be the same, but would not need to stress the same pathway for all clients for the unicast transmission.

Chapter 9

Conclusion

We have designed and implemented an application for scheduling data streams over a multicast protocol. We have evaluated the implementation, as well as discussed both the performance and the infrastructure such an implementation is dependent on. We will now summarize our work and point out our most important contributions. Last, we will present challenges that we believe will contribute to this area of research.

9.1 Summary and contribution

We have looked at different techniques for scheduling streams over a multicast protocol. Out of the techniques examined we found the Cautious Harmonic Broadcasting (CHB) algorithm to be the most promising with regard to bandwidth consumption at the server, and in the network, as well as offering a near video on demand service. By studying existing implementations and simulations, we gained insight in performance issues motivating our own design of a streaming server for Video on Demand (VoD).

We have also become acquainted with common protocols and standards for implementing periodic scheduling techniques. This served as a good background when choosing the live555 framework for developing our own video streaming application. It also helped us choose a media format that was beneficial for streaming in a lossy environment; Transport Streams (TS).

Our main contribution is that of designing a working prototype using our own version of the CHB scheduling algorithm. The implementation works with an existing framework, live555, that is used by open source media players like VLC and MPlayer. As a result of the assessment of our prototype, we have suggested that conflicting assumptions in constant and variable bit rates in video streams can interfere with CHB scheduling. We also observed that the temporally aligned scheduling scheme in CHB easily becomes skewed. This can, however, be a benefit in avoiding peaks in the bandwidth. Last, we have looked at optimization possibilities and the applicability of the CHB algorithm.

Based on experiments conducted using the prototype, we have established that creating and deploying a CHB streaming service is feasible with today's infrastructure.

9.2 Future work

In our work, we found irregularities in the streaming, as well as a higher bandwidth consumption than expected. We have raised the question whether this may be caused by variable bit rate scheduling of segments that are the same size, byte wise. To investigate this further, it would be interesting to see how the server application performs when creating segments based on duration instead of size. Live555 supports an index file to seek to time based offsets in the TS file. Hence, support for such modifications to the code is readily available. Along the same line it would be useful to do tests on the estimated packet duration that is used when determining the scheduling time for transmission of RTP packets. Further research into how the estimation is calculated is essential to develop an application that streams closer to the bandwidth modeled theoretically.

As described in our discussion, the reassembly of the segments may cause loss or duplication of frames. We would find it interesting to investigate the effects of using the RTP extension header to include more meta data regarding the TS frames transmitted.

As the server prototype uses UDP ports for streaming, the application is generally best suited for an overlay multicast protocol. We would like to modify the prototype to support channels based on multicast addresses, this to be able to perform more authentic experiments. That way the benefits of multicast routing would be taken fully advantage of, and any overhead caused by using an overlay network would diminish.

Appendix A

A.1 Measurements of network delay from Oslo to selected Norwegian destinations

The round trip between an ADSL user with a download bandwidth of 3000 kbit, and upload at 350 kbit, to selected servers throughout Norway was measured using the ping application. Table A.1 displays the results from a count of 100 packets sent, whereas the same test was repeated several times with similar, though not logged, outcome.

Location	**URL**	**rtt min**	**rtt avg**	**rtt max**	**rtt var**	**%loss**
Andenes	www.andoyposten.no	66.243	68.348	79.627	1.531	0
Narvik	www.narvik.kommune.no	74.485	76.707	98.494	2.452	0
Trondheim	www.mamoz.no	51.622	53.308	57.955	0.93	0
Bergen	www.securehosting.no	56.977	59.308	63.042	1.016	0
Kristiansand	kristiansand.folkebibl.no	49.643	54.644	169.248	17.633	0
Oslo	www.linpro.no	62.45	65.454	80.302	2.425	1
Norway		60.237	62.962	91.445	4.331	0.167

Table A.1: *Measuring round trip time using ping from an ADSL home user to selected servers throughout Norway bottom line displays the average*

A.2 Specification of test machines

All machines used were set up with Linux Ubuntu 7.04 (feisty) or higher.
Server:
Ubuntu 7.04 (feisty)
Memory: 768 MB
Processor: Intel Pentium 4, CPU 1.60 GHz

Network Emulator:
Ubuntu 7.04 (feisty)
Memory: 512 MB
Processor: Intel Pentium 4, CPU 1.60 GHz

Clients:
Ubuntu 8.04 (hardy)
Memory: 512 MB
Processor: Intel Pentium 4, CPU 1.60 GHz

A.3 Movie specifications

The Wrong Landing
Duration: 270 seconds
Resolution: 320 x 240
Frames per second: 25.0
Video: Unknown
Audio: 64.0 kbit/4.54% (ratio: 8000->176400)
Transmission bandwidth: Unknown. Used 650 kbps for testing

The Elephant's Dream
Duration: 654 seconds
Resolution: 512 x 288
Frames per second: 24.0
Video: 800.0 kbps
Audio: 224.0 kbit/14.58% (ratio: 28000->192000)
Transmission bandwidth: 1100 kbps

A.4 Test results, Scheduling performance based on client wait time and number of segments

A.4.1 The Wrong Landing

Wait time	Download time	Segments	Download indicator
271	270.29	1	0.50
136	138.89	2	0.34
91	137.94	3	0.38
68	166.28	4	0.49
55	183.49	5	0.56
46	203.07	6	0.64
39	221.58	7	0.71
34	238.90	8	0.78
31	254.09	9	0.84
28	263.45	10	0.88
25	270.89	11	0.92
23	275.33	12	0.94
21	278.81	13	0.95
20	280.71	14	0.96
19	282.07	15	0.97
17	285.41	16	0.99
16	287.35	17	1.00
15	290.25	19	1.01
14	292.93	20	1.03

Table A.2: *Metrics from wait time vs download indicator investigation for 'The Wrong Landing'*

A.4.2 The Elephant's Dream

Wait time	Download time	Segments	Download indicator
654	655.47	1	0.50
327	330.08	2	0.34
218	294.13	3	0.34
164	343.47	4	0.42
131	366.08	5	0.47
109	370.67	6	0.49
94	385.51	7	0.52
82	377.15	8	0.51
73	434.59	9	0.60
66	426.23	10	0.59
60	416.55	11	0.58
55	404.59	12	0.57
51	400.17	13	0.57
47	399.26	14	0.57
44	406.93	15	0.58
41	467.53	16	0.67
39	535.78	17	0.77
37	532.33	18	0.77
35	528.37	19	0.77
33	514.28	20	0.75
32	508.06	21	0.74
30	494.88	22	0.72
29	482.79	23	0.71
28	505.79	24	0.74
27	589.12	25	0.87
26	628.35	26	0.92
25	626.71	27	0.92
24	624.99	28	0.92
23	616.46	29	0.91
22	567.36	30	0.84
21	567.49	32	0.84
20	646.72	34	0.96
19	727.23	36	1.08
18	706.43	38	1.05
17	626.13	40	0.93
15	815.36	46	1.22
14	681.53	50	1.02

Table A.3: *Metrics from wait time vs download indicator investigation for 'The Elephant's Dream'*

A.5 Collaboration diagrams and call graphs

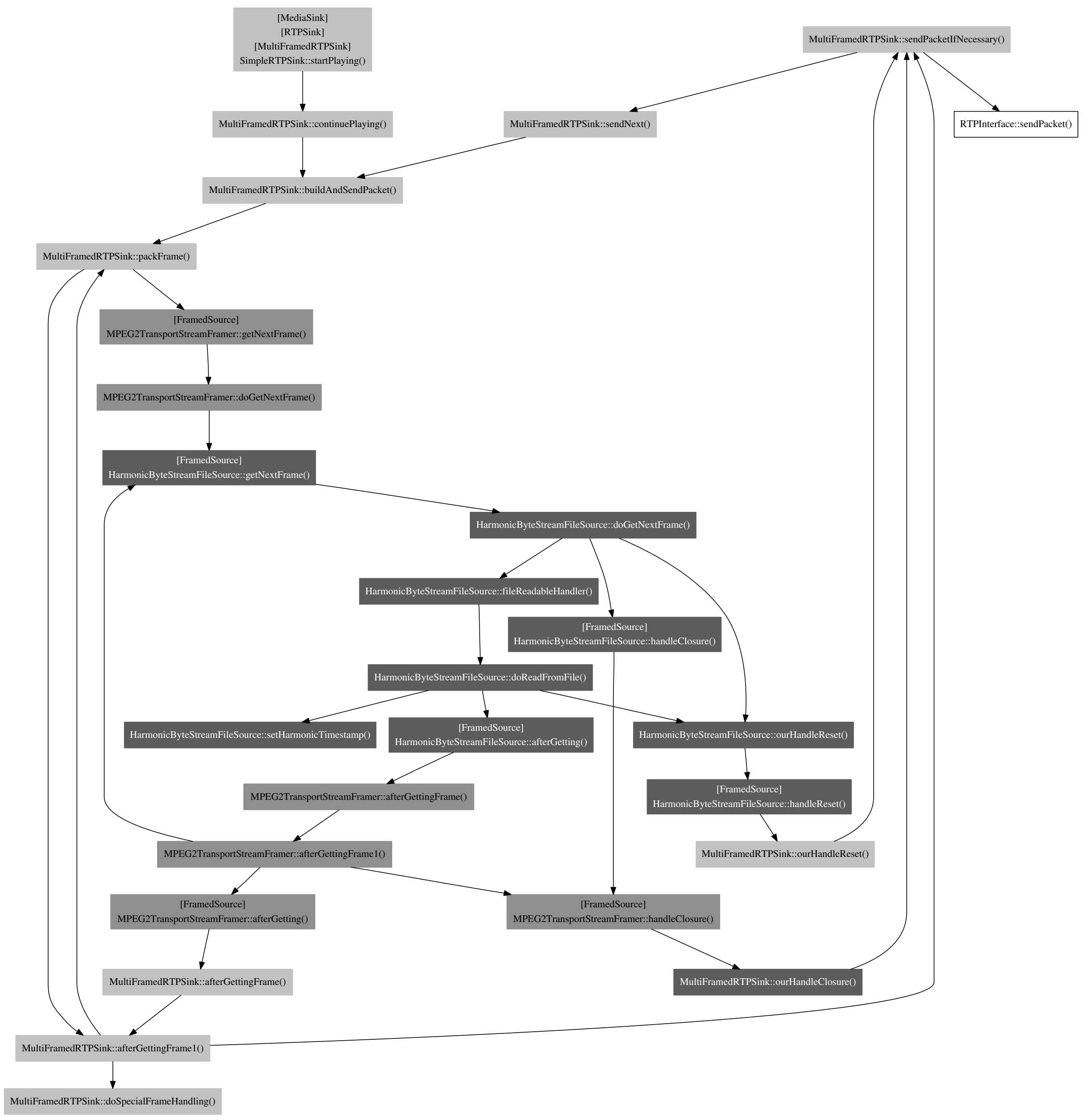
[MediaSink]
[RTPSink]
[MultiFramedRTPSink]
SimpleRTPSink::startPlaying()
MultiFramedRTPSink::sendPacketIfNecessary()
MultiFramedRTPSink::continuePlaying()
MultiFramedRTPSink::sendNext()
RTPInterface::sendPacket()
MultiFramedRTPSink::buildAndSendPacket()
MultiFramedRTPSink::packFrame()
[FramedSource]
MPEG2TransportStreamFramer::getNextFrame()
MPEG2TransportStreamFramer::doGetNextFrame()
[FramedSource]
HarmonicByteStreamFileSource::getNextFrame()
HarmonicByteStreamFileSource::doGetNextFrame()
HarmonicByteStreamFileSource::fileReadableHandler()
[FramedSource]
HarmonicByteStreamFileSource::handleClosure()
HarmonicByteStreamFileSource::doReadFromFile()
HarmonicByteStreamFileSource::setHarmonicTimestamp()
[FramedSource]
HarmonicByteStreamFileSource::afterGetting()
HarmonicByteStreamFileSource::ourHandleReset()
[FramedSource]
HarmonicByteStreamFileSource::handleReset()
MPEG2TransportStreamFramer::afterGettingFrame()
MPEG2TransportStreamFramer::afterGettingFrame1()
MultiFramedRTPSink::ourHandleReset()
[FramedSource]
MPEG2TransportStreamFramer::afterGetting()
[FramedSource]
MPEG2TransportStreamFramer::handleClosure()
MultiFramedRTPSink::afterGettingFrame()
MultiFramedRTPSink::ourHandleClosure()
MultiFramedRTPSink::afterGettingFrame1()
MultiFramedRTPSink::doSpecialFrameHandling()

Figure A.1: *Simplified call graph for the server sink, framer and source. Functions implemented in parent classes have the parent class name indicated in square brackets.*

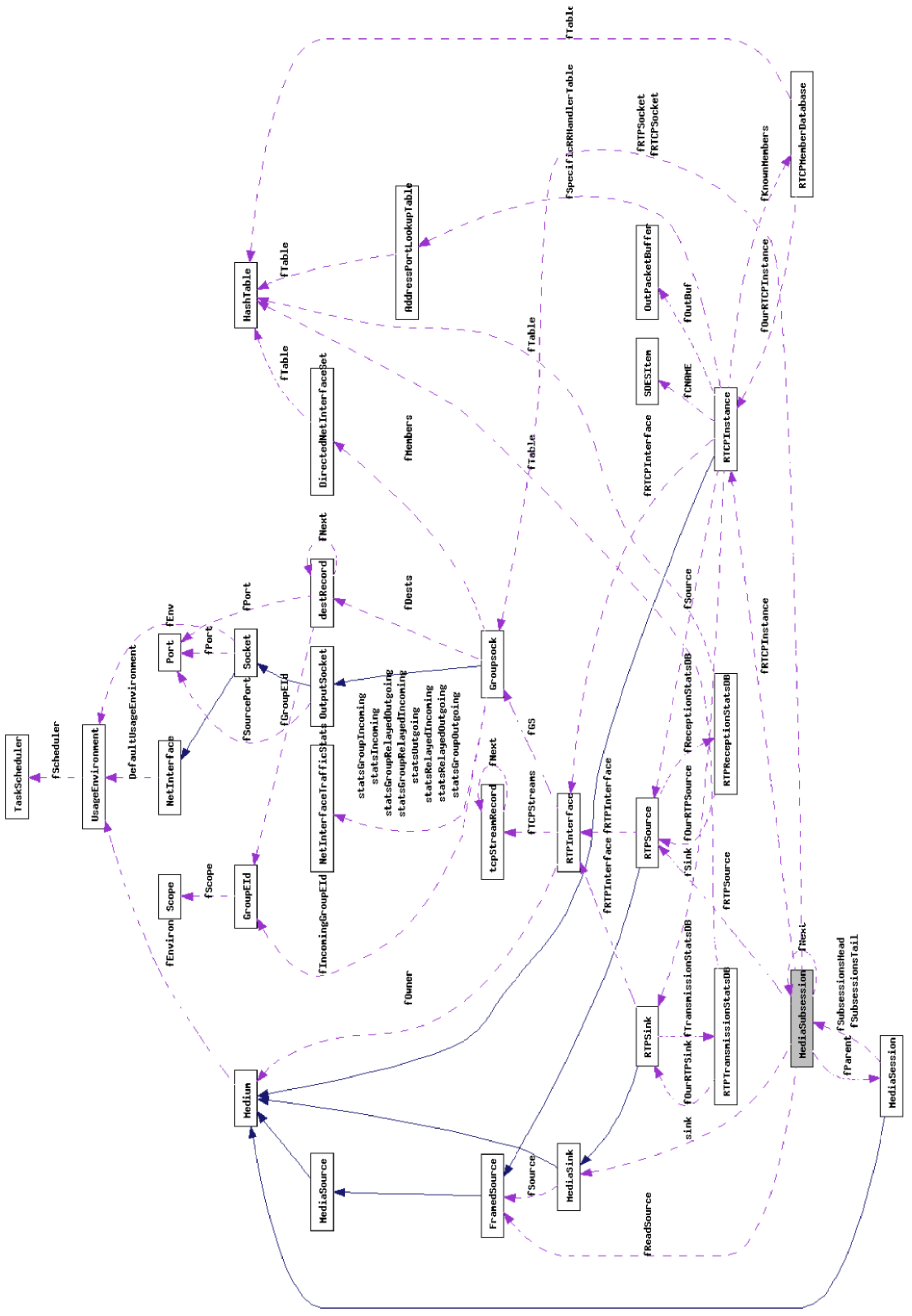

Figure A.2: *Collaboration diagram for MediaSession. The diagram is without our additions to the library. Diagram taken from live555 online documentation [3].*

Bibliography

[1] Carsten Griwodz and Pal Halvorsen. Inf5071: Performance in distributed systems, distribution part 1. `http://www.uio.no/studier/emner/matnat/ifi/INF5071/h07/undervisningsmateriale/06-distribution-1.ppt`, November 2008. [Online; accessed 06-November-2008].

[2] Steven W.Carter Jehan-Francois Paris and Darrell D.E.Long. Efficient broadcasting protocols for video on demand. In *Proc. 6th Int. Symp. Modeling, Analysis, and Simulation of Computer and TeleCommunication Systems (MASCOTS'98)*, pages 127–132, July 1998.

[3] Ross Finlayson. Live555 streaming media. `http://www.live555.com`, March 2008. [Online; accessed 18-September-2008].

[4] Blizzard Entertainment. World of warcraft. `http://www.worldofwarcraft.com`, November 2008. [Online; accessed 17-November-2008].

[5] VideoLAN Project. Videolan media player. `http://www.videolan.org`, July 2008. [Online; accessed 18-September-2008].

[6] MPlayer team. Mplayer. `http://www.mplayerhq.hu`, September 2008. [Online; accessed 18-September-2008].

[7] Steven W. Carter, Darrell D. E. Long, and Jehan-François Paris. *Video-on-demand broadcasting protocols*. Academic Press, Inc., Orlando, FL, USA, 2001.

[8] Asit Dan, Dinkar Sitaram, and Perwez Shahabuddin. Scheduling policies for an on-demand video server with batching. *IBM TR RC*, (19381), 1993.

[9] Thomas D.C. Little and Dinesh Venkatesh. Prospects for interactive video-on-demand. *IEEE Multimedia 1(3)*, 1(3):14–24, 1994.

[10] Leana Golubchik, John C. S. Lui, and Richard R. Muntz. Adaptive piggybacking: A novel technique for data sharing in video-on-demand storage servers. *Multimedia Systems Journal 4(3)*, pages 140–155, 1996.

[11] Kevin Almeroth and Mustafa Ammar. On the use of multicast delivery to provide a scalable and interactive video-on-demand service. *IEEE JSAC 14(3)*, pages 1110–1122, 1996.

[12] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *Multimedia Systems Journal 4(4)*, pages 197–208, 1996.

[13] L. Juhn and L. Tseng. Harmonic broadcasting for video-on-demand service. *IEEE Transactions on Broadcasting 43(3)*, pages 268–271, 1997.

[14] Derek Eager, Mary Vernon, and John Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. In *Multimedia Information Systems Conference*, 1999.

[15] Michael K Bradshaw Bing Wang Lixin Gao Jim Kurose Prashant Shenoy Don Towsley and Subhabrata Sen. Periodic broadcast and patching services: implementation, measurement, and analysis in an internet streaming video testbed. In *Proceedings of the 9th ACM int. conference on Multimedia*, pages 280–290, Ottawa, Canada, September 2001.

[16] L. Gao, J. Kurose, and D. Towsley. Efficient schemes for broadcasting popular videos TITLE2:. Technical Report UM-CS-1998-016, , 1998.

[17] Lixin Gao and Don Towsley. Supplying instantaneous video-on-demand services using controlled multicast. In *In Proc. IEEE International Conference on Multimedia Computing and Systems*, pages 117–121, 1999.

[18] Lixin Gao, Zhi li Zhang, and Don Towsley. Catching and selective catching: Efficient latency reduction techniques for delivering continuous multimedia streams. In *In Proc. ACM Multimedia*, pages 203–206, 1999.

[19] C. Dara Vesuna and Jehan-Francois Paris. An empirical study of harmonic broadcasting protocols. *Computacion y Sistemas*, 7(3):156–166, 2004.

[20] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[21] Andrew S. Tanenbaum. *Computer networks*. Prentice-Hall, Englewood Cliffs, N.J., 2nd ed. edition, 1989.

[22] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376 (Proposed Standard), October 2002. Updated by RFC 4604.

[23] H. Holbrook, B. Cain, and B. Haberman. Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast. RFC 4604 (Proposed Standard), August 2006.

[24] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

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

[26] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566 (Proposed Standard), July 2006.

[27] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

[28] D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar. RTP Payload Format for MPEG1/MPEG2 Video. RFC 2250 (Proposed Standard), January 1998.

[29] M. Civanlar, G. Cash, and B. Haskell. RTP Payload Format for Bundled MPEG. RFC 2343 (Experimental), May 1998.

[30] Information Technology - generic coding of moving pictures and associated audio information: Systems. International Standard ISO/IEC 13818-1:2000(E), International Organization for Standardization and the International Electrotechnical Commission, 2000.

[31] Fernando Pereira and Touradj Ebrahimi. *The MPEG-4 Book*. Prentice Hall PTR, 2000.

[32] Fred Halsall. *Multimedia Communications: Applications, Networks, Protocols and Standards*. Addison-Wesley, 2001.

[33] Ralf Steinmetz and Klara Nahrstedt. *Multimedia: Computing, Communications and Applications*. Prentice Hall PTR, 1995.

[34] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396 (Draft Standard), August 1998. Obsoleted by RFC 3986, updated by RFC 2732.

[35] Dimitri van Heesch. doxygen:source code documentation generator tool. `http://www.stack.nl/~dimitri/doxygen/index.html`, May 2008. [Online; accessed 18-September-2008].

[36] Blender Foundation Netherlands Media Art Institute. Elephant's dream. `http://www.elephantsdream.org`, October 2008. [Online; accessed 01-October-2008].

[37] Chema Hernandez. The wrong landing. `http://sph3re.tv/download_the-wrong-landing_267.html`, September 2007. [Online; accessed 13-September-2007].

[38] H.R. Wu and K.R. Rao. *Digital Video Image Quality and Perceptual Coding*. CRC Press, 2005.

[39] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, April 2004.

[40] B. Carpenter. Architectural Principles of the Internet. RFC 1958 (Informational), June 1996. Updated by RFC 3439.