UNIVERSITY OF OSLO
Department of Informatics

Periodic Broadcasting Protocol - implementation and measurements

Master thesis

Tommy Gudmundsen

May, 2009

Periodic Broadcasting Protocol - implementation and measurements

Tommy Gudmundsen

May, 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Multimedia content consists of both video and audio content, and have traditionally been the domain of television broadcasters to deliver to the consumers. The multimedia content receiver used by most consumers is the television, and any required subscription decoder devices. For movies the traditional, and still dominating, approach is renting a movie on a physical medium like a DVD disk from a video rental store.

As technology advances in the computer and networking areas, more people then ever are now equipped with powerful personal computers and high speed connections to the Internet. This development and increase in Internet connected consumers, along with popular Internet multimedia services like Youtube featuring user generated content, are factors that help establish this platform as a competitor to the traditional multimedia platforms. The consumers that take part in the computing technology evolution, are getting more and more accustomed to multimedia content being available on the computer platform, and this also raises the requirements on content providers for them to remain competitive.

Among the content providers currently following this line in Norway, are some of the online news providers like, www.vg.no, www.dagbladet.no, along with several others. These content providers are all providing multimedia content as part of their news coverage. This is just one case that supports our argument that consumers are getting more accustomed to multimedia being a natural part of the computing platform.

Video on-demand refers to the media content being available to the consumer whenever the consumer desires. This is very different from the traditional broadcasting scheme, where content is available only according to a particular schedule, determined by the broadcasters.

Some of the traditional broadcasters in Norway, are also making parts of their content available on the computing platform. Examples are two of the largest Norwegian broadcasting corporations like NRK (www.nrk.no/nett-tv) and TV2 (www.tv2sumo.no). The NRK are making a some of their archived TV-shows, for which they hold the distribution rights, available as video on-demand, free of charge. The other example is TV2, through the TV2 Sumo brand, where a payed subscription is required. TV2 is also actively promoting this distribution platform, by making popular TV-shows available to consumers on this platform, before the content is available on the traditional broadcasting platform.

There also exist other computing based video-on demand providers like SF Anytime (http://www.sf-anytime.com), that provide mainly movies to the scandinavian market. In the U.S market there are rising providers like Hulu (www.hulu.com), providing video on-demand movies as well as TV-series.

Multimedia is a very rich content form, where the amount of data involved is very high, compared to just text and motionless pictures. This high amount of data, consumes a very large amount of bandwidth, when transferred from the media server to the client. This means that the content providers, as well as the internet service providers, will have to support very high loads, in terms of network bandwidth. Most of the existing solutions today are based on a unicast model where each video on-demand user will consume the same amount of network resources. As the number of users grow, the content providers will no longer be able to serve their new customers, as their solutions will run out of resources, and is unable to scale. This leads us into our problem statement.

1.2 Problem Statement

As we stated in our background section, the unicast data transmission model employed by most of the video on-demand solutions available today, is not a scalable solution, and will have limits on the number of customers that can be served. In order to address this issue, researchers have proposed different multicast stream scheduling techniques, where the multimedia transmissions for the same media are shared among the users.

The multicast approach is not well suited for video on-demand solutions without the use of stream scheduling techniques. This is because there is an inherent conflict between the sharing property in multicast, and the on-demand property in video on-demand. The periodic broadcasting class of stream scheduling techniques typically divides the media into disjoint segments, and repeatedly multicast the different segments on separate channels, according to some algorithm. These techniques mitigates the leap between the sharing and on-demand properties, enabling efficient use of our network resources, and providing scalable solutions.

In this thesis we will examine the periodic broadcasting class of stream scheduling algorithms, and the Cautious Harmonic Broadcasting algorithm in particular. Much of the research that have been performed in the areas of stream scheduling algorithms, are mainly based on mathematical analysis and simulations. We therefor seek to create a framework for implementing periodic broadcasting stream scheduling algorithms, and to employ our framework to implement the cautious harmonic broadcasting algorithm. In order to examine the performance and operation of our implementation, we will perform experiments in a testbed and analyze the results, as well as comment on the experienced issues.

As a base for implementing our stream scheduling framework, and cautious harmonic broadcasting algorithm, we have chosen to use the live555 [3] streaming media library. This library implements most of the primitives and standards compliant protocols we require for our implementation. The live555 library is open source, extensible, and employed in media-players like VLC [4] and MPlayer [5].

1.3 Main Contributions

In this thesis we have argued that the current unicast delivery schemes for video on-demand are unable to scale when the number of consumers are rising. Valuable network resources are not being used in an optimal way, where all the consumers watching the same movie, all use exclusive video streams transmitting the same content. A multicast infrastructure is not available in todays global Internet, but as demand for more efficient network utilization keeps rising, in parallel with growing use of multimedia content and a growing number of users, internet service providers are gaining incentives to implement multicast support in their autonomous systems. This multicast infrastructure within an ISPs domain, is required for establishing a scalable multicast based video on-demand service, and we also argue that this is the most realistic short term scenario for the deployment of a scalable video on-demand service.

We have examined different stream scheduling techniques, and chosen to focus on the periodic broadcasting class of solutions, due to its scalability, as well as predictable resource consumption, regardless of the number of consumers. The most promising algorithm, considering its demands for server bandwidth, client waiting time, client access bandwidth, client buffer requirements, as well as complexity, is the Harmonic variant named cautious harmonic broadcasting (CHB). The CHB algorithm have been criticized for its high requirement for channels, translating into a high number of IP multicast addresses, but our scope of applicability is running this algorithm inside of an internet service providers autonomous system, and thus will not be unmanageable.

The main contributions of this thesis is the implementation of the cautious harmonic broadcasting algorithm, as well as a stream scheduling framework (SSF), that supports implementing periodic broadcasting class algorithms. By implementing a SSF we support implementing other promising periodic broadcasting algorithms in our future work, in a more consistent approach. The SSF is built by extending the live555 streaming media library, enabling future integration into existing media-players like VLC and MPlayer. The implementation have been tested through running experiments, and measuring the performance and operation of the CHB algorithm, using both constant bit-rate and variable bit-rate media.

We conclude that the implementation of periodic broadcasting algorithms are realistic and feasible, and that the CHB algorithm performs well in most of our experiments. However we also suggest that more research is necessary in the area of transmission scheduling calculations, and especially for small media segments, and even more so for the ones based on a variable bit-rate media. The outcome of further research in this area, would be to ensure on-time delivery of the media regardless of media segment size and bit-rate variations.

1.4 Outline

In chapter 2, we establish terminology and concepts related to stream scheduling techniques, as well as examine different classes of stream scheduling techniques proposed in the literature. After the examination of different stream scheduling techniques we examine some of the existing implementations in chapter 3. In chapter 4, we describe the most relevant protocols and standards required in a streaming media implementation, before we turn our attention to our design and requirements in chapter 5. After exploring our design we move on to chapter 6, where we elaborate on our implementation and how our design has been realized. In chapter 7, we describe the experiments performed in order to test and measure our implementation, as well as discuss the results. After exploring the measurements results, we conclude and summarize our work in chapter 8.

Chapter 2

Scheduling techniques

In this chapter we are going to examine different classes of stream scheduling techniques for video on-demand. We argued in our introduction the resource intensive nature of multimedia, as well as the need for stream scheduling techniques used in conjunction with multicast technology. Performing delivery of the same data to all users, and repeating all the steps involved, without any sharing of data, mean we are utilizing our resources poorly. The stream scheduling techniques examined in this chapter, all aid in better utilization of our network and server resources.

We start by establishing some of the terminology and concepts used in relation to stream scheduling techniques, before moving on to the exploration of three different classes of stream scheduling techniques.

2.1 Terminology and concepts

The following lists some of the common terminology and concepts we use when discussing stream scheduling techniques.

2.1.1 Terminology

Unicast Is the most common form of communication in use today, where the data is transferred from one sender to one receiver. Unicast does not allow for sharing of communication resources, and the resource consumption increases linearly with the number of users.

Multicast Enables one or more senders to deliver data to a group of interested receivers. The receivers join multicast groups, and the network is responsible for building a spanning-three from the sender to all the receivers, which is used as forwarding paths to deliver data to all receivers. This enables the most efficient use of the network resources, by ensuring that the same data is only sent across the same link once. The network resources needed at the sender, thus becomes independent of the number of participants in the multicast session. Multicast requires additional management, and more advanced network routers than unicast. This description is specified as Any-Source Multicast (ASM). Source-Specific Multicast (SSM) adds support for filtering traffic based on the source, so that only multicast traffic from a specified source is forwarded to the receiver. The receiver needs to specify the sources it wants to receive from when it joins the multicast group.

Broadcast Sends its data from one sender to every receiver. A typical example of the broadcasting scheme is traditional television transmission, where one sender sends the same data to every receiver. The broadcast scheme has a high resource consumption, where the entire network carries all the transmitted data regardless of any receiver presence.

Segment A non overlapping part of a media, like a video file, is referred to as a segment. The media can be divided into fixed or variable sized segments.

Channel An addressable resource used to transmit data between the server and client.

Stream Is a continuous flow of data, transmitted across a network path. A typical example of a stream would be the transmission of a segment over a network channel.

Session In our context a session refers to the transfer of an entire media, like a video file. The session streaming a media file will typically be a continuous process in most stream scheduling algorithms, that runs until the server is terminated.

Consumption rate Also referred to as the playback and playout rate, is the rate a client will receive at in order to consume the media, i.e watching a video is done at its consumption rate.

Wait time The time period between a clients request for a media, and the time at which the client can begin consuming the media.

Trick play Refers to VCR functions on a media stream like the ability to play, pause, stop, forward and rewind.

2.1.2 Concepts

Video on-demand (VoD) Refers to the users ability to gain access to the media at the time the user desires, as opposed to a traditional broadcasting schedule set by the broadcasters.

Broadcast Does not support any VoD access, but operates according to a fixed schedule.

True VoD (T-VoD) Gives the user full control over when to access a media, without any delay.

Near VoD (N-VoD) Provides the user access in a VoD way, but delays exist, and there will be a waiting time before the media is accessible.

In the following sections we will examine different classes of stream scheduling techniques available.

2.2 Delayed on-demand delivery

The delayed on-demand delivery scheme is based on the true on-demand delivery scheme, as it will attempt serving client requests as quickly as possible after the arrival request. The element that separates delayed on-demand delivery from true on-demand delivery is the waiting time the client experience after its request for a video has been received by the server. This delay is used by the server to collect and group client requests, before transmission begins, or perform adaptations to merge clients onto existing streams. The delayed on-demand delivery scheme thus has a dynamic scheduling mechanism, where the client arrival pattern affects the transmission schedule. This dynamic schedule support also implies a higher server complexity than fixed schedule solutions. In this section we will examine two different delayed on-demand delivery schemes, batching and adaptive piggybacking.

2.2.1 Batching

The idea behind the batching technique is to group as many clients onto as few streams as possible, thereby saving network resources by allowing the client to share the data transmission. This grouping of client requests will infer a delayed response for the waiting clients, and this period is referred to as the batching window [6].

For this approach to be effective, and aid in saving network resources, it depends on many clients accessing the same media within a limited time period, the batching window. This requirement for effectiveness also mandates a balanced approach to determining the size of the batching window. If the batching window is set too large, the client waiting time may exceed the limit a user is willing to wait before abandoning the request. On the other hand, if the batching window size is set too small, the network resource savings will diminish.

The batching window may be fixed or variable, depending on the batching policy applied. The batching policy determines how we group client request together, and also what media is being served first. If the batching server only have a limited number of channels it uses for streaming, the client requests will contend for these resources, and the batching policy will determine the outcome. Next we will cover two of these batching policies, namely First-Come-First-Server (FCFS), and Maximum Length Queueing (MLQ).

If we arrange all the incoming client requests into a First In First Out (FIFO) queue, where each client requests a specific video, this request queue will collect client requests for the duration of our batching window. After the batching window limit is reached, we proceed by applying the batching policy in order to select the requests that will get serviced.

The FCFS batching policy will select the first client request in our FIFO queue, and allocate resources for the media requested by the client at the head of the queue. Next the FCFS policy will examine the rest of the FIFO queue for clients requesting the same media, and group these clients together on the resource allocated. This approach may lead to limited savings of network resources if there are no other clients in the FIFO queue requesting the same media. On the other hand this approach ensures a maximum client waiting time, and request processing fairness.

The MQL batching policy does not pick the client request at he head of the FIFO request queue, instead it examines the entire client request queue to determine the media that have the most requests. The media with the most requests will then get streaming resources allocated, and the clients requesting this media will be grouped together and served. This approach will be the most efficient way of saving network resources, as it will always be able to group the most amount of clients onto a single stream. This approach will favor popular media, over less popular media requests. This means that a fairness issue exists when using MQL. If a client request is at the head of our FIFO request queue, and is the only client requesting a particular media, it may never get served if request for more popular media are continuously added to the client request queue. This fairness issue may be reduced by applying a maximum client waiting time limit along the MQL batching policy.

2.2.2 Adaptive piggybacking (stream merging)

Unlike the batching technique, the adaptive piggybacking, or stream merging, approach does not have an initial client waiting period, but starts serving the client immediately upon request arrival. When the streaming server have multiple clients receiving the same media, the adaptive piggybacking server will alter the media display rates for the streams in progress, in order to achieve resource savings. [7].

If the server have two clients, *A* and *B*, both in progress of receiving the same media, but on different streams. Where client *A* started playback some time before client *B* started its playback. The adaptive piggybacking idea is to merge these two clients onto the same stream by changing the display rates until they are at the same position in the media playout. This is performed by lowering the display rate for client *A* while increasing the display rate for client *B*. This alteration will lead to the two clients ending up at the same position in the media playback schedule, and the two streams can be merged.

The underlying assumption here is that he user watching the streamed media is unable to perceive display rate manipulations as long as the manipulations are kept within a

5% rate change [8]. Adaptive piggybacking may also be used together with batching to further increase the resource savings, however the implementation complexity may in sum be significant. [9]

Next we move on to examine another class of stream scheduling algorithms, prescheduled delivery.

2.3 Prescheduled delivery

The prescheduled delivery scheme is based on the principle where the server is running a fixed schedule for streaming media content. This static schedule is set upon server startup and does not change during server operations, unlike the delayed on-demand delivery scheme does. This also implies that the server resource requirements are fixed, and easier to calculate and scale. The initial resource requirements for prescheduled delivery schemes tends to be quite high, compared to the initial resource requirements of other schemes. This means that the prescheduled delivery scheme will be most beneficial in higher load scenarios, compared to some of its delayed on-demand delivery counterparts. The scalability, in terms of the number of concurrent clients it servers, is also one of the strong sides of prescheduled delivery techniques, as the server resource consumption remains fixed independent of the number of clients is serves.

The prescheduled delivery schemes are also referred to as periodic broadcasting schemes, and involves dividing videos into segments, and streaming these on separate channels, according to a transmission schedule. These segments are then repeatedly broadcasted on the channel, and the scheduling is designed to ensure that all segments are received in time for consumption, ensuring continuous playback. This technique shifts much of the complexities to the clients, requiring clients to follow a reception schedule, and buffer and reorder received data.

The prescheduled delivery scheme can be vulnerable to failure, as it is a single instance running the static schedule, a failure in this instance will result in a failure at all the clients currently receiving, making this a single point of failure.

In the following sections we will examine a set of different prescheduled delivery techniques proposed in the literature.

2.3.1 Staggered Broadcasting

Staggered broadcasting [10] is among the earliest, and simplest periodic broadcasting protocols proposed. The principle is to allocate K channels, that all stream the entire media repeatedly, but with staggered starting times. The client waiting time will then depend on the duration of the media S, divided by the number of channels allocated, $w = S/K$. Each of the allocated channels streams the media at its playout bandwidth b, and the total bandwidth consumption B, increases linearly with the number of channels. $B = b * K$. This technique is not among the most efficient, in terms of bandwidth consumption, but has been put into practice in production systems.

One of the attractive properties with staggered broadcasting is that it can provide some trick-play functions, by jumping between the different staggered streams, allowing the client to discretely navigate back and forward in the streaming media. Staggered broadcasting is also among the techniques that does not require very much complexity in the receiving client.

2.3.2 Pyramid Broadcasting

Pyramid broadcasting [11] is a technique that assumes higher client capabilities than many of the other periodic broadcasting techniques, in terms of required client access bandwidth and client storage.

Pyramid broadcasting divides its available bandwidth into fixed size HIGH-bitrate channels, with one channel for each segment. Each segment is repeatedly broadcasted on its assigned channel. Pyramid broadcasting divides its media into variable size segments that grows exponentially according to a geometric series. The first segment, representing the earliest part of the media, is the smallest segment, causing this segment to be broadcast most frequently and determines the client waiting time for the technique.

The client only starts downloading segments from the start, and a client will begin consuming the first segment as soon as it has started downloading it. By only downloading the segments from the beginning avoids the need for client reordering. All segments are downloaded according to its consumption sequence, as soon as the client begins consuming segment N, the client will start downloading segment $N+1$ at its earliest occurrence.

Pyramid broadcasting also supports interleaving several different media files on the same set of channels. The media files are divided into the same number of segments, and each media segment of the same relative size alternates over its assigned channel. A graphical representation of pyrmid broadcasting is available in figure 2.1.

In the worst case, the client will need to download from all the channels concurrently, and the client access bandwidth requirements reaches its maximum requirement, equal to the server bandwidth. Also the client buffer requirement may be as high as 50% of the media.

2.3.3 Skyscraper Broadcasting

Skyscraper broadcasting [12] was proposed as a novel technique with lower client access bandwidth and storage requirements than Pyramid broadcasting, while still keep-

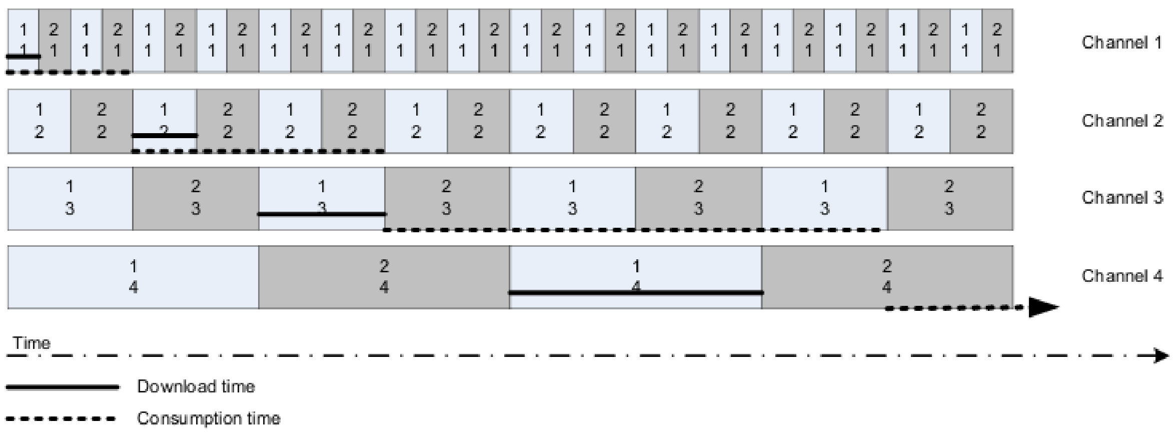

Figure 2.1: Pyramid broadcast. *Adapted from Griwodz and Halvorsen [1]*

ing an acceptable client waiting time.

Skyscraper broadcasting uses fixed bandwidth channels, all with a bandwidth corresponding to the media playback rate. The segmentation scheme for Skyscraper is effectively based on fixed size segments, and multiple segments are allocated to channels according to a channel allocation series. The Skyscraper proposal in [12], uses variable size segments, but again sub-divides the segments that are larger into fragments, so that we effectively end up working with equal sized blocks of media data, and we get more than one segment allocated to a channel. A graphical representation of skyscraper broadcasting is available in figure 2.2.

The channel allocation series, also referred to as the broadcast series [12], describes how we map segments onto channels, from a recursive function. A sample channel allocation series for 11 channels may be:

```
[ 1, 2, 2, 5, 5, 12, 12, 25, 25, 52, 52 ]
```

This series allocates one segment on the first channel, two segments on channel two and three, and so on. The segments are mapped in sequence, so the segment order for our three first channels would be 1 - 2,3 - 4,5 respectively.

The structure of Skyscraper broadcasting ensures that the client needs to receive from at most two channels concurrently, and needs to buffer at most two segments at any one time. The client requirements in terms of client access bandwidth and storage, are thus lower than in the Pyramid broadcasting technique.

2.3.4 Harmonic Broadcasting

Harmonic broadcasting (HB) [13], was introduced as a more bandwidth efficient alternative to its pyramid based predecessors, for the same client waiting time achieved.

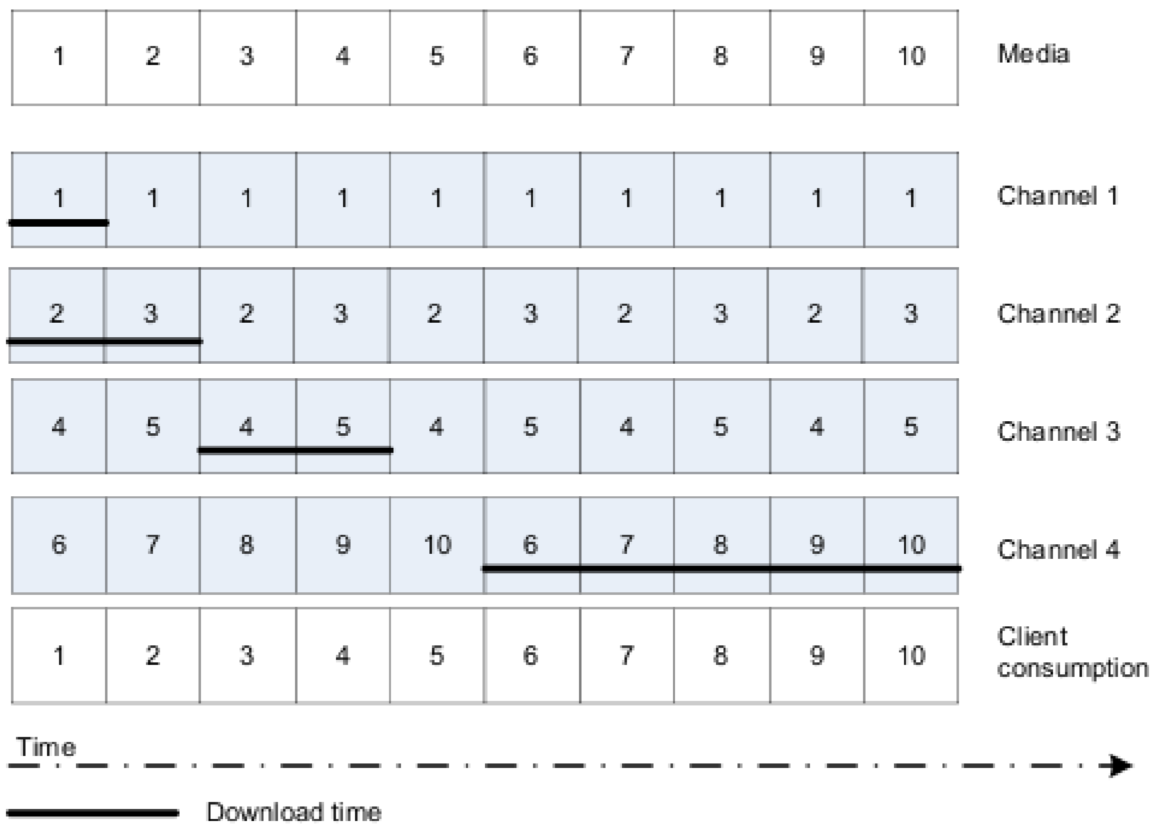

Figure 2.2: Skyscraper broadcast. *Adapted from Griwodz and Halvorsen [1]*

HB divides the media into n equally sized segments S. The algorithm also allocates n channels with a decreasing bandwidth for each channel i. The playout bandwidth b, is referred to as full bandwidth. Each of the channels are allocated $\frac{b}{i}$ units of bandwidth, and each segment S_i is assigned to a single channel, where it is repeatedly broadcast. A graphical representation of the HB algorithm is shown in figure 2.3.

The client waiting time for the HB algorithm corresponds to the duration d of the first segment. The bandwidth requirement for the HB algorithm grows according to the harmonic number, by adding the bit-rates for the different channels like: *1/1 + 1/2 + 1/3 ... 1/n*. This implies that we are able to achieve low client waiting time, without a large increase in the required bandwidth, as the harmonic number grows very slowly when adding additional channels. This has also been one of the critiques towards the HB algorithms, as it requires a significant number of channels.

In the paper "Efficient broadcasting protocols for video on demand" [2], by Paris, Carter and Long, the authors proved that the HB algorithm is not always able to deliver all of its data on-time. The HB algorithm needs to add an additional *(n-1)d/n* units of time, in order to guarantee on-time delivery. As a result Paris, Carter and Long proposed two alternatives to the HB protocol which corrected the delivery guarantee problem. These algorithm are examined in the following sections.

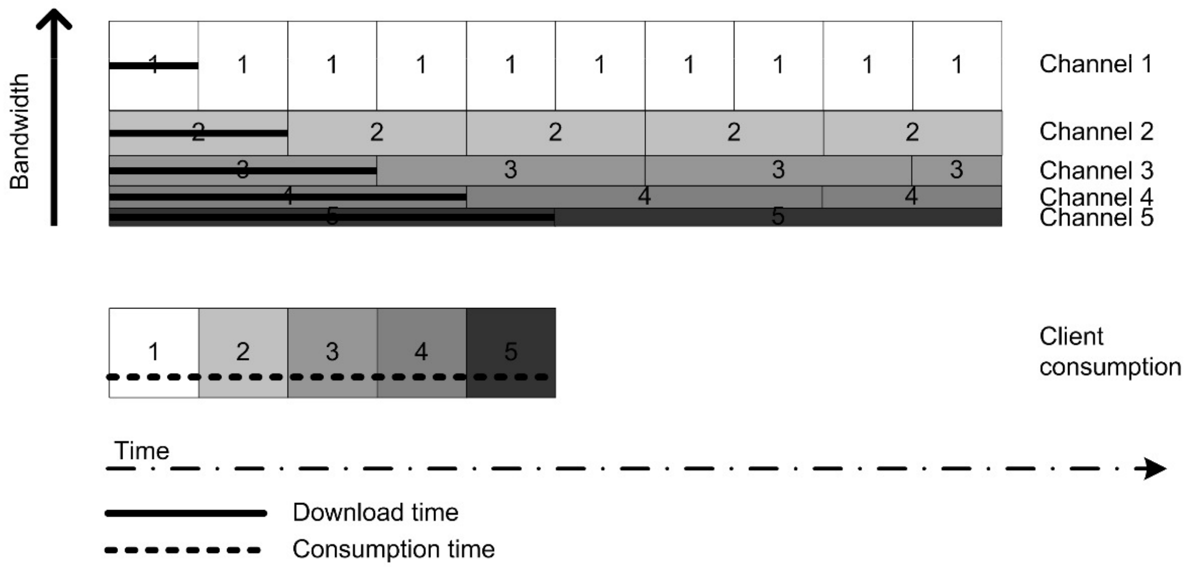

Figure 2.3: Harmonic Broadcast. *Adapted from Griwodz and Halvorsen [1]*

2.3.5 Cautious Harmonic Broadcasting

Cautious Harmonic Broadcasting (CHB) [2], improves upon the on-time delivery issue in the original HB algorithm. The changes performed involves rearranging the channel bandwidth allocations. CHB allocates two full bandwidth channels, as opposed to just one in HB. The second channel is used as a shared channel, alternately streaming segments 2 and 3. At the same time the the remaining channels are allocated bandwidth $b_i = \frac{b}{i}$ *for* $i = 3, \dots, n-1$. This scheme guarantees on-time delivery, as all of the first 3 segments are transmitted at full bandwidth. This will increase the total bandwidth requirement for CHB with approximately *b/2* units, for a high number of channels. The graphical representation of CHB is available in figure 2.4.

2.3.6 Quasi-Harmonic Broadcasting

The second proposal in [2], was Quasi-Harmonic Broadcasting (QHB). The QHB algorithm has the advantage of offering no additional bandwidth requirement compared to the HB algorithm, however the QHB complexity is higher.

The QHB algorithm still streams the first segment repeatedly at full bandwidth, where the segment occupy one time slot. For each segment *i, for* $1 < i <= n$, the segment is broken into *im-1* fragments for some parameter *m* , and the client will receive *m* fragments from each channel per time slot. By further dividing each time slot into *m* equally sized subslots, then the client will receive a single fragment during each subslot. One of the key principles in QHB is the way the fragments are arranged. For an illustration of the layout we refer to figure 2.5.

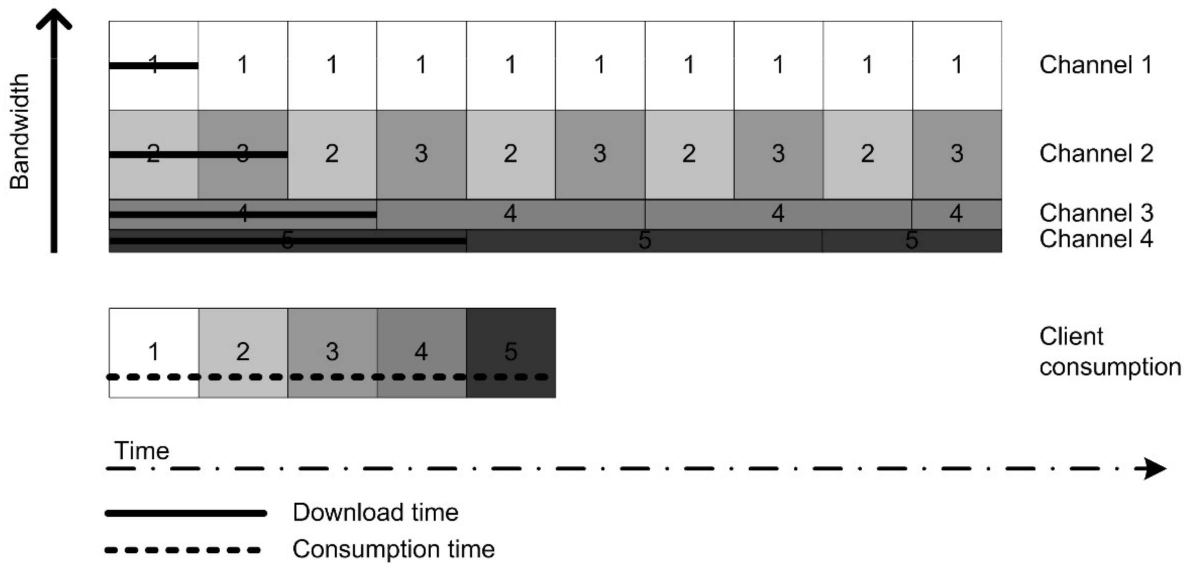

Figure 2.4: Cautious Harmonic Broadcast. *Adapted from Griwodz and Halvorsen [1]*

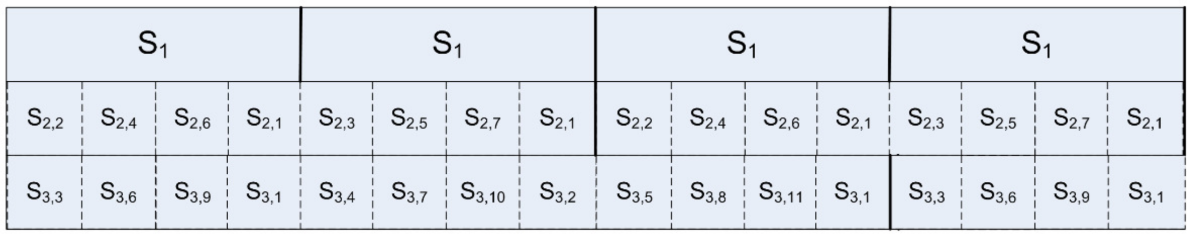
S_1 S_1 S_1 S_1
$S_{2,2}$ $S_{2,4}$ $S_{2,6}$ $S_{2,1}$ $S_{2,3}$ $S_{2,5}$ $S_{2,7}$ $S_{2,1}$ $S_{2,2}$ $S_{2,4}$ $S_{2,6}$ $S_{2,1}$ $S_{2,3}$ $S_{2,5}$ $S_{2,7}$ $S_{2,1}$
$S_{3,3}$ $S_{3,6}$ $S_{3,9}$ $S_{3,1}$ $S_{3,4}$ $S_{3,7}$ $S_{3,10}$ $S_{3,2}$ $S_{3,5}$ $S_{3,8}$ $S_{3,11}$ $S_{3,1}$ $S_{3,3}$ $S_{3,6}$ $S_{3,9}$ $S_{3,1}$

Figure 2.5: Quasi-Harmonic Broadcast. *Adapted from [2]*

For channel i, the last subslot of each time slot is used to send the first *i-1* fragments of S_i in order.

The rest of the subslots sends the other *i(m-1)* fragments, such that the k^{th} subslot of slot *j* is used to send fragment *(ik+j-1) mod i(m-1)+i*. [2].

2.4 Client-Side Caching

The client-side caching class of stream scheduling techniques relies on buffering of media at the client side, receiving multicast streams, in addition to any potential unicast patch streams necessary to receive the entire media on-demand. The client-side caching scheme offers on-demand delivery, and require significant client resources. We are in the following briefly going to describe two client-side caching approaches.

2.4.1 Patching (Stream Tapping)

The patching scheme [14], also referred to as stream tapping [15], is based on the server multicasting the entire media repeatedly. Clients connect to the multicast stream, and begin receiving the media. If a client arrives after the current multicast stream has started, the client requests a patch stream, that is used to serve the part of the media missed by the client. The patch stream is a unicast stream, and is triggered on demand, which makes the patching scheme a true video on-demand scheme. As suggested in [14], the patching scheme may be combined with other schemes, such as batching, to gain better server and network resource utilization, at the expense of a slightly reduced on-demand response. One principle consists of merging large batches onto multicast patch streams. The patching scheme does not provide the same scalability we can achieve in the periodic broadcasting schemes, but are less resource demanding for less popular media.

2.4.2 Hierarchical Multicast Stream Merging (HMSM)

In the hierarchical multicast stream merging (HMSM) technique [16], the combined use of different techniques like, piggybacking, patching and the variant of skyscraper named *dynamic skyscraper*, are employed to achieve a true video on-demand service. The transmissions are all based on multicast, where the clients receive data at a rate that is higher than the playout rate by reception from multiple streams, like in skyscraper. The streams are also accelerated as in piggybacking, and groups of users are merged in large multicast groups. The scheme provides significant resource savings on the server side, but the complexity of the HMSM scheme, may be a significant implementation challenge.

2.5 Summary and discussion

In this chapter we have examined three different classes of stream scheduling techniques. All aiming to aid in better server and network resource utilization. We have examined the delayed on-demand delivery techniques, the prescheduled delivery technique, also referred to as periodic broadcasting, and the client-side caching technique.

Through our study of different stream scheduling techniques, we have decided to further work with the periodic broadcasting class of techniques. The main arguments for this selection is the scalability, and applicability for serving highly popular media to a high number of users. Among the available periodic broadcasting techniques, we have chosen to implement the cautious harmonic broadcasting algorithm. We arrived at this choice after examining the properties for bandwidth use, client waiting time, as well as application complexity and client side buffer requirements. For the bandwidth utilization alone, we could have chosen the QHB algorithm, but have abandoned it due to its complexity compared to CHB. We believe that CHB is the algorithm that has the most promising bandwidth saving potential, as well as a good balance in the other properties mentioned. We also believe that the current state of technology, is well capable of supporting both a CHB server as well as a client.

After examining the options with regards to stream scheduling techniques, we move on to examining existing implementations, with our focus directed toward the harmonic type of stream scheduling algorithms.

Chapter 3

Existing Implementations

In chapter 2 we examined different classes of scheduling techniques. Most of the literature proposals are based on mathematical analysis and theoretical simulations, due to the lack of contemporary infrastructure, enabling realistic tests. There exist however some work addressing the scheduling techniques in a more practical manner. As we established in chapter 2, we find the periodic broadcasting class of algorithms the most interesting to study further, and the CHB stream scheduling technique in particular. In this chapter we examine a paper where the authors have performed a practical simulation using the CHB scheduling technique. We also move on to examine in more detail the contemporary work conducted by one of our peers, in their experiences implementing the CHB scheduling technique.

3.1 An Empirical Study of Harmonic Broadcasting Protocol

In chapter 2 we examined the CHB algorithm in more detail, and concluded that it was our choice for implementation. It is therefore of particular interest to us that this is the algorithm studied in the paper "An Empirical Study of Harmonic Broadcasting Protocols" [17]. The authors have performed experiments in a simulation environment, focusing on the performance of the CHB protocol itself.

3.1.1 Experiment setup

The authors set up a client server pair used to execute the simulations, where they ensure accurate clock synchronization between the two parties, to achieve more accurate measurements. The server design is using pre-forked processes to handle the different streams, this implies one process managing one stream or channel. The client side forks each stream into a new process upon connection.

The media used in the experiments is not a real media, but a constant bit-rate dummy file, generated and stored in memory. The dummy file is stored in memory and compressed, these measures are taken to avoid being affected by the disk I/O subsystem delays in the measurements. The compression is also added to ensure that no intermediary system on the data transport path will be able to compress the media and thus affect the measurements.

The authors have selected the reliable TCP protocol as their transport protocol, however they have also discussed and considered using the non-reliable UDP protocol. The most important argument for choosing the reliable and connection-oriented TCP protocol, was to ensure no loss of important frames in the video. A series of tests was also performed on the TCP usage, in order to find and use the most suitable TCP window size. The broadcasting protocol selected for simulation by the authors was the CHB periodic broadcasting protocol.

3.1.2 Results

The experiments showed that a single Pentium III computer with 512 MB memory, was able to serve 2400 concurrent streams, providing a maximum client waiting time of 5.3 seconds, for a 180 minute video. The authors claim that CHB is a strong contender for distributing popular video to large audiences.

Beyond this number of concurrent streams, the system reliability deteriorated, and significant delays in packet deliver was experienced. The authors suggests further research using a connectionless protocol, as opposed to the use of a connection oriented protocol in their experiments. The authors also suggests to reduce the number of processes involved in implementing the CHB protocol.

3.2 Scheduling of data streams over a multicast protocol

Upon beginning the work on this thesis, a similar and closely related work was already in progress by another master student. In her thesis “Scheduling of data streams over a multicast protocol” [18], Fredrikson have examined some of the same objectives, set forth in this thesis. The work performed by Fredrikson in [18], was completed towards the end of 2008, a few months prior to the completion of our work. Thus we were able to gain insight into the experiences of closely related work, and address some of the issues experienced.

3.2.1 Description

In [18], the objectives are closely related to our own, in particular the implementation of the CHB stream scheduling technique.In the following section we will explore the most important issues discovered in [18], and how they influence our own design. It is also worth remarking that as the two similar thesis evolved in parallel, they do not share any implementation artifacts like the source code.

3.2.2 Future work suggestions

This section will sum up some of the key issues experienced by the author of [18], that was suggested for further research. These issues will have an influence on our design and implementation.

CHB implementation

The CHB implementation was modified in order to simplify the code, and avoid transmission of multiple segments on the same channel. Instead of alternating segments 2, and 3 on channel 2, as in the CHB design in [2], segment 2 was the only segment transmitted on channel 2. Segment 3 was instead transmitted on a new channel at half the playout bandwidth. These changes results in an increase in the total bandwidth consumption of approximately, one half of the playout speed.

In our thesis we will follow the CHB specification put forth by the authors in [2].

Bandwidth consumption

The measurements performed in [18], indicated that the total bandwidth consumption of the implementation well exceeded the theoretical calculations. The author suggests that this is related to the transmission scheduling mechanism in the live555 library, and possibly also affected by the use of variable bit-rate videos.

We intend therefore to further examine these issues in our work, and also comparing the effects of different bit-rate variations, by using both constant and variable bit-rate media in our measurements.

Segmentation scheme

The segmentation scheme employed in [18], divided the media into segments based on its byte size. When handling variable bit-rate media, this scheme will not correspond to equal sized segments, time-wise. The author experienced segments not arriving on-time for consumption, and non ordered completion sequences for some of the steams, and suggests examining the issue further.

Segment reassembly

The segments transmitted from the server to the client needs to be reassembled at the client side, due to RTP fragmenting the segments into discrete packets, or fragments. For the client side to be able to correctly reassemble the RTP packets into a full segment, Fredrikson mad use of the RTP timestamp field. This was done by manipulating the RTP timestamp on the server side, setting it to zero at each segment start. The receiver then used two buffers to store the segment parts, switching buffer upon RTP timestamp reset. The client was thus able to examine the RTP timestamps, and determine if the arriving packet belonged to the first or second part of the segment. The segment parts was later combined into the full segments.

This use of the RTP timestamp, could cause loss or duplication that the client was unable to detect. (We refer to [18] chapter 8 for details.) There is also no way to discriminate different segments on the same channel when employing this scheme. The author therefore suggests RTP extension headers as an alternative, which we are going to explore further in chapter 5.

Channel design

The channel design in [18] uses the same IP multicast address for all of its logical channels, using port numbers for separation. This channel design does not work well with IP multicast, because the port numbers are a construction that belongs on the transport layer, not the network layer. As the multicast routers operate on the network layer, and are unable to distinguish the different channels, the bandwidth consumption through the network will not decline when the preliminary channels are complete. The multicast traffic will remain until the movie is completely downloaded.

3.3 Summary

In this chapter we have examined existing implementations of the CHB scheduling techniques. The first paper used a multi process design, using reliable transport of segments, achieving good results, and concludes that the CHB algorithm is an important contender in the distribution of popular media to large audiences. In our second paper, we examine the work recently performed by one of our peers, exploring some of the same objectives as our own. In this paper we focus on the experiences and future work suggestions set forth. We find issues related to scheduling, channel design, as well as segmentation and reassembly. The results from these papers will help guide our design, as well as providing us with a better foundation for making our design decisions. We now move on to exploring the commonly used protocols in media streaming systems, as well as notes on media encodings.

Chapter 4

Protocols and coding standards for video streaming

In this chapter we will very briefly list the most important protocols and media coding standards used in our work. Additional comments to these protocols and standards, will be provided in context of its use.

Real-time Transport Protocol (RTP) [19] RTP is designed for transport of real-time data, and offer services like sequence numbering and time-stamping, and will be employed as our real-time protocol in stream scheduling implementations.

Session Description Protocol (SDP) [20] SDP is designed for describing sessions, and has will be employed as the format we use for this purpose. The SDP protocol will also been extended using custom attributes, used to describe metadata in our stream scheduling implementation.

Internet Group Management Protocol (IGMP) [21] Is the protocol we will employ for multicast group membership management.

Protocol Independent multicast - Sparse Mode (PIM-SM) [22] PIM-SM will be used for multicast tree building, and multicast traffic forwarding.

The Motion Picture Expert Group (MPEG) video codecs In our work we employ the MPEG-2 media encoding, using the transport stream (TS) encapsulation. The TS encapsulation is the best suited for use in lossy environments, and is therefore preferred over the program stream (PS) encapsulation format. The encapsulation formats are specified in system part in the ISO / IEC 13818-1:2000 standard.

Chapter 5

Design

After examining different stream scheduling techniques in chapter 2 and looking at exiting implementations in chapter 3, we selected the CHB stream scheduling algorithm to be implemented and tested in our stream scheduling framework. In this chapter we are going to examine the most important design requirements and issues, required for implementation of our solution. We will also provide arguments and rationale for our design decisions, where this is required. We start by examining the media stream format.

5.1 Stream format

In our study we have decided upon using the MPEG-2 media encoding, due to its widespread use, and also good support in the live555 streaming media library we are going to employ. The MPEG-2 media format supports two different schemes for encapsulating the media. The Program Stream (PS) format is designed for error free environments, like playback from a DVD disk. The other option is the Transport Stream (TS) encapsulation, which is designed for more lossy and unreliable mediums. The TS also contains synchronization information embedded in the TS, making it more robust and resistant to loss of TS packets. We therefore chose to use the MPEG-2 TS stream format in our work.

5.2 Media segmentation and reassembly

In the CHB stream scheduling algorithm we need to divide the media into segments on the server side, transport these to the client, and reassemble the media on the client side. In this section we are going to discuss the options we have and describe our design.

5.2.1 Segment creation

When dividing a media into its respective segments, there are two main options. We can perform the division by byte-size or by duration in time. The CHB segmentation scheme uses segments of equal size, and we will now explore the two options available.

Time based segmentation

In a time based segmentation scheme we divide the media into segments based on the duration of the media. In our design, using MPEG-2 TS media files, this means that we need a way of accessing the position in the media that corresponds to a given instance in the media duration. We require the byte offset in the media file corresponding to the given time instance. The reason for this requirement is that we rely on the underlying operating systems file handling functionality, and the operations available requires byte related information to operate.

In order to obtain the byte offset into the media for a given time instance, we need an index file that provides this mapping. The live555 library we have chosen to use, supports creating the required index file by parsing the MPEG-2 TS media, and extracting the time instance for each of the I-frames in the media. This index file thus also provides us with the total duration of the media, which we require in order to divide it by time.

The index files resolution is dependent on the I-frame density, and thus will not alway return the exact position that corresponds to our query, but the closest one available. This may lead to slight variations in the size of the different segments, but the differences are not big enough to cause scheduling problems. The index file also makes sure that the positions returned are a multiple of 188 bytes which corresponds to the MPEG-2 TS packet size.

By using this index file we are able to divide variable bit-rate media, as well as constant bit-rate media files, into equal length segments. And thus we chose to include this segmentation scheme into our design.

Byte based segmentation

The byte based segmentation scheme, is the simpler of the two, as it does not require any additional support beyond the file handling functions provided by the underlying operating system. In this approach we simply divide the media into equal byte sized segments. The only issue to maintain is that we make our divisions a multiple of 188 bytes, which corresponds to the MPEG-2 TS packet size. This ensures that we do not split a TS packet into two different segments.

For true constant bit-rate media files, the byte based segmentation scheme and the time based segmentation scheme, will not differ by any significant amount in the resulting segmentations.

For completeness in our framework we have chosen to support both of these segmentation schemes.

5.2.2 Reassembly of media segments

The CHB stream scheduling technique will receive multiple segments simultaneously, and thus there is a need for client buffering, used to hold the segments until consumption time.

In our client design we have chosen to receive the entire media, and reassemble the different segments into the full media. This design is due to our desire to test our framework and CHB implementation, and this allows us to validate the entire media after reception. In later stages, it is possible to alter our design to support continuous delivery of the received media to a media player, but at this stage we chose not to add the media player delivery mechanism.

The CHB stream scheduling algorithm assume that the receiver obtains all the segments from their beginning, and does not start receiving a segment at any other point in its progress. As we will see in the section describing our channel design, we use IP multicast to deliver segments. This design implies that the client would have to listen to the channel transferring the segment, and discard all the received data until the segment beginning was detected. This approach would waste network resources, and we therefore design our client to support reception from the instance it starts.

This support for beginning reception at an arbitrary point in the segment progression, may cause our client to receive the last part of a segment before the first part. To address this issue we need to note the starting position of our segment reception, and use this offset when reassembling the media. Another consequence of the support for beginning the reception at an arbitrary place in the schedule, is for channels streaming multiple segments in sequence. The client will start receiving data according to a schedule, and there might be a mismatch between the client schedule position and the servers current schedule position. This requires support for hot-swapping the receiving buffer, so that the data received is assigned to the correct segment. We will examine how we obtain starting point information as well as segment affinity in a later section.

Each of the segments will be buffered in separate files, and when all the segments have been received, the segments are combined into the full media, and will be validated against the source media.

5.3 Transmission of segments

Like most kinds of data, multimedia data like audio and video, must be transported to a client in order to be available and useful. For this transport there is a need for a mechanism that is able to transport the data, in a way that is suitable for the data that needs transfer. This mechanism is referred to as a channel. In our thesis the channels are transporting multimedia data, with real-time properties, and the channel design must reflect this. By real-time properties we refer to the need to deliver the multimedia data on-time, and before its playback deadline expires. This also implies that it is more important to receive the multimedia data on time, than to make sure we receive all the data, without any lost parts. The arguments for the claim that it is more important to receive the data on time, than to receive all the data without any loss, is based upon the consequences it has for the receiving client. When watching a video, the consequences of loosing part of the data, due to packet loss for instance, will be a momentarily degradation of quality in the video that is played back. If however, the receiver operates in a reliable way, using TCP, and does not accept any lost data, the client would have to suspend the playback of the video until the lost data is retransmitted from the sender. If the channel transporting the video stream is subject to some packet loss, the video playback would be regularly suspended, while waiting for retransmissions, and the impact would result in a more disruptive playback of the video. We therefore argue that the channel design should be based on a transport scheme without reliability guarantees like UDP.

5.3.1 Channel design

The channel design is based on support for IPv4 multicast, and thus we assign a single multicast IP address to each channel. This design will have a significant requirement on the number of multicast IP addresses required, but as we stated in our introduction chapter, we argue that this design would be realistic when operated inside the bounds of an ISPs autonomous system. This design will benefit from the network layer transport of multicast traffic, as the client will be able to leave the multicast groups as soon as all the channel data is received, thus reducing the network load as soon as possible in the edges of the multicast tree. We have also chosen to use different transport layer port numbers for the different channels, in addition to different multicast IP addresses. This design is due to the inability to discriminate incoming multicast traffic destined for the same port number, but on different IP addresses, in some versions of the linux kernel.

5.3.2 Protocols

The protocols used in transferring our media, will be RTP over UDP/IP for the transmissions, as we have argued at the beginning of this section. For IPv4 multicast support, we have selected the IGMP group management protocol, and the PIM-SM protocol for multicast forwarding, and tree management. More information on the testbed setup is available in chapter 7.

5.4 Channel and session scheduling

In this section we will describe the process and scheduling primitives necessary for enabling our client to receive media data from the streaming server in the intended order. In the CHB stream scheduling algorithm we have multiple segments and multiple channels, and some of the segments share a single channel. For our framework client to be able to support other periodic broadcasting algorithms, using different schemes for how channels and segments are related and used, we need a way to express this relation, so that the client are able to receive and reassemble the media correctly. We first introduce the concepts and their meaning, before describing the design.

Channel schedule: The ordering of segments on a channel.

Session schedule: The ordering of channels in a session.

Channel cycle: A complete channel schedule iteration.

Session cycle: A complete session schedule iteration.

The cycle concept is implicit, and determined at runtime by the client and server. A cycle defines the total span of a channel schedule or a session schedule. When a channel schedule has been run from start to finish, it completes one channel cycle. On the client side this implies that the segments on this channel are received. On the server side this implies running the channel schedule again in a new channel cycle. This process repeats until the server is terminated. The same principle applies to the session schedule.

The client needs to obtain the entire cycle to get all the data, and when the client have obtained all the cycle data, it no longer needs to receive data from the active channel or session.

There are two kinds of schedule in the SSF, channel schedule and session schedule. A channel schedule describes the ordered allocation of segments to a channel. An example channel schedule may look like this:

```
{[S1, S3, S5]}
```

This channel schedule states that a cycle is streaming segments *1-3-5* before the cycle ends and is started again from the beginning.

A session schedule describes the overall session, defining what channels the client must receive from at any given time. When a channel is listed in the session schedule, it means that the client should receive an entire cycle on this channel. The session schedule may be both serial and parallel in nature. A serial session schedule means that the client only receives from one channel at any given time. A parallel session schedule means that the client must receive from two or more channels simultaneously. One example of each kind follows:

```
Serial session schedule: { [C1],[C2],[C3],[C4] }
```

One cycle for the entire session, where the client receives all the content on channel 1, then all the content on channel 2, and so on.

```
Parallel session schedule:  { [C1, C2], [C3, C4], [C5, C6] }
```

One cycle for the entire session, where the client receives first channel 1 and 2 simultaneously, then channel 2 and 3 simultaneously, and so on.

When a client has received an entire session cycle, it is done receiving. For each of the session schedule channels, the client must receive an entire channel cycle, with all the specified segments.

The behavior in the CHB stream scheduling algorithm is to receive from all the channels simultaneously, and thus we need not specify a session schedule, only the channel schedules are required. We therefore make this the default behavior for the client when no session schedule is specified.

5.5 Session management

The session concept refers to the process of transferring a media from the streaming server to the client, and all the steps involved. The session needs to be established, executed or performed, before it finally terminates. All of these tasks needs to be designed and handled by our system. In our system we have chosen to employ the SDP protocol for the purpose of describing and conveying information about a session. For us to be able to convey all the information we require in order to support stream scheduling techniques, we define a set of custom attributes in our SDP format, both for the session level and the media level.

5.5.1 Session definition

The session is created and made available when our server starts, and is initialized. It is therefore the servers responsibility to define the session it offers to clients. The server will dynamically create a SDP file describing the available session. In order for our system to support the CHB stream scheduling algorithm, we need to define a set of custom attributes in SDP. The following list will sum up these requirements.

- Specification of the number of segments and channels.
- Description of each of the segments.
- For each of the channels we require a channel schedule description.

For details on the realization of these custom attributes we refer to chapter 6.2, and also the sample SDP file in listing 6.2 in the same chapter.

5.5.2 Session establishment

The client needs to access the SDP file generated by the server, in order to establish the necessary channel connections, and receive the media segments. One of the options would be to implement the RTSP protocol in our server and serve the SDP file through this instance. However we believe that the ability to serve the SDP file through an RTSP server, would add more overhead and complexity than necessary. The reasoning behind this design decision is that our server does not support any trick-play functionality, and the ability to serve the SDP file is a small part of the RTSP protocol. The most important requirement for serving the SDP file to our client is that it be, served over a reliable protocol, to ensure that a correct session can be established. Our proposal is to serve the SDP files from a web-server, running the HTTP protocol over TCP. An added benefit from this design is that the serving of the SDP file to potentially thousands of clients, becomes disconnected from the streaming server and would not induce the added load onto this server. We also remark that a web-server would make a good platform for presentation of the different videos that are available to the user.

5.5.3 Session execution and termination

After the client has obtained the SDP file describing the session, and parsed the SDP file in order to establish the required channels, segments, and channel schedules. The client begins executing the session and channel schedules. This implies that the client starts receiving segments over the established channels, in the sequence and order specified. After the client have received all the different segments for the entire session, the client will terminate the session.

5.6 Reassembling fragments into segments

Upon segment transmission on the server side, the segment data is fragmented into smaller blocks of data, and transmitted in an RTP packet. This fragmentation needs to be handled correctly at the receiving client, in order to assure correct reassembly of the segments. In the CHB stream scheduling algorithm our design needs to support streaming several segments over a shard channel, in sequence. This design in combination with our design choice where the client can begin receiving data at any point in the channel schedule, implies that we require information in the incoming RTP packets about segment affinity and segment progress. In order for us to fulfill this requirement we have chosen to use the RTP extension headers option, in the RTP protocol. The live555 library currently does not support RTP extension headers, and thus we need to implement this functionality. We will go into more detail on how this design is realized in chapter 6. We also require the segment data to receive in order for us to be able to correctly reassemble the fragments into segments, for this issue we will use a reordering-buffer in our receiving client. A final requirement is that a single RTP packet cannot contain data from two different segments, as we would be unable to detect this in our RTP extension header usage. The default behavior for the live555 library is to allow a certain time to ensure the RTP packets are filled up before transmission, this behavior will need to be overridden to avoid data from different segments in the same packet.

5.7 Integration with existing media players

We seek to provide a design that will support integration with existing media players. We have examined the live555 streaming media library [3], which is in use by existing open source media players like VLC [4] and MPlayer [5]. In our design we therefore chose to extend and build upon the live555 library.

5.8 Process management

As seen in our study of existing implementation in chapter 3, a CHB implementation using separate processes for each channel, leads to significant memory overhead, as well as extensive context switching when the number of channels are high. We therefore take the recommendations put forth in [17] into consideration, and select a single process design for our server and client.

5.9 Summary

In this section we have covered the most important design requirements, and issues for our stream scheduling framework and CHB algorithm. In the next chapter we move on to see how these requirements are realized and put into practice.

Chapter 6

Implementation

In this section we will examine our stream scheduling framework and CHB stream scheduling algorithm implementation, as well as how our design requirements have been realized.

6.1 Live555

Live555 streaming media [3] is an open source library written in C++, intended for implementation of streaming media applications. The library implements a series of open standard protocols such as RTP/RTCP,RTSP,SDP,SIP, as well as support for several popular media encoding formats. The Live555 library is also used in several open source media players to support streaming media, such as VLC [4] and MPlayer [5]. This integration into existing media players, along with the fact that the library is extensible, makes the library a good choice for implementing our stream scheduling framework upon.

6.1.1 Live555 library description

The following list describes the main parts of the live555 library in the authors own words, obtained from [3].

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

UsageEnvironment The *UsageEnvironment* and *TaskScheduler* classes are used for scheduling deferred events, for assigning handlers for asynchronous read events, and for outputting error/warning messages. Also, the *HashTable* class defines the interface to a generic hash table, used by the rest of the code.

These are all abstract base classes; they must be subclassed for use in an implementation. These subclasses can exploit the particular properties of the environment in which the program will run - e.g., its GUI and/or scripting environment.

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

6.1.2 Live555 concepts and typical program flow

It is very important to understand the key concepts and how the media data travels inside the live555 library on a conceptual level. We will therefore introduce the key concepts first and then move on to the more detailed parts.

Source This is a source of media data, it may be a file source, or a network source receiving RTP packets.

Sink The sink acts as the destination for the media data.

Filter A filter is positioned between a source and a sink, and takes on the role of a source when interacting with the sink. This concept allow us to add extra functionality in the media data path. A typical example of a filter is a framer.

These three concepts operates within the bounds of a client side or a server side. This means that data traveling over a network, in form of RTP packets for example, does not flow from a source to a sink in the live555 concepts. Rather, the RTP packets is a side-effect of an RTP sink.

The flow of the live555 library, along with a description of the task schedulers basic operation is next described, again in the authors own words obtained from [3].

Applications are event-driven, using an event loop *TaskScheduler::doEventLoop()* that works basically as follows:

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

Also, before entering this loop, applications will typically call
someSinkObject->startPlaying();
for each sink, to start generating tasks that need to be done.

Data passes through a chain of sources and sinks - e.g.,

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

(Sources that receive data from other sources are also called *filters*.)

Whenever a module (a sink or one of the intermediate filter sources) wants to get more data, it calls *FramedSource::getNextFrame()* on the module that's to its immediate left. This is implemented by the pure virtual function *FramedSource::doGetNextFrame()*, that is implemented by each source module.

Each source module's implementation of *doGetNextFrame()* works by arranging for an *after getting* function to be called (from an event handler) when new data becomes available for the caller.

In the next section we examine the details of our stream scheduling framework.

6.2 Stream Scheduling Framework overview

In this section we will provide a brief overview of the most important component in the SSF. We will separate the overview into the client and server side components. The use and interaction between the different components will be examined in the sections following this brief overview, and we will not go into the component details in this section. Before examining the client and server components we will examine the files that provides the base for initializing and creating the client and server components.

6.2.1 The Media Session Descriptor file

The MSD file is used by the server to determine the media session parameters, and the controller type. This means that for each media we want to send using our server we define a MSD file for the session. in listing 6.1 we provide a sample MSD file and will briefly examine the different parameters.

medianame= provides the name of the media we are streaming.

mediafilename= specifies the path and name of the media file we are streaming.

clientwaitingtime= gives the maximum client waiting time for the media in seconds, this is again used to calculate the number of segments we require to satisfy the maximum client waiting time.

streamschedulingalgorithm= specifies the stream scheduling algorithm we are going to use for streaming the media. This determines the type of controller the server creates to handle the streaming session. Currently we only support the CHB algorithm.

segmentdivisionmethod= is used to determine how we divide our media file into segments. Currently two options are supported, *time* and *bytes*. Time division dictates use of the playout duration to divide the media into segments of equal length. The bytes scheme will ignore the playout duration and divide the media into equally byte-sized segments.

```
medianame=The Movie
mediafilename=test.ts
clientwaitingtime=27
streamschedulingalgorithm=CHB
segmentdivisionmethod=time
```

Listing 6.1: Sample MSD file

6.2.2 The Session Description Protocol file

As described in our design chapter, the SDP protocol is used by our client to be able to connect with the session server and receive data. The session SDP file is dynamically generated by our streaming server upon creation and initialization. A sample SDP file is shown in listing 6.2. In this section we will examine the custom attributes we have added to our SDP file in order for the clients to be able to support our streaming server. We do not examine all of the standard SDP parts.

We will first describe the session parameters, then go on to the channel (m=) parameters.

a=segment-count:24 specifies the number of segments in the session. This aids the client in initializing its datastructures.

a=channel-count:23 specifies the number of channels in the session, and is also intended for client initialization aid.

a=segment-description:1:25469:27207 Each of the segments available on the server is described in the SDP file. This segment description provides two parameters when streamed in the byte based segment division scheme, and one additional parameter when streamed using the time based segment division scheme. The first parameter is the segment ID, identifying the segment. The second parameter is the size of the segment. In our MPEG-2 TS CHB implementation, the size is provided as the number of 188 bytes TS packets. The third parameter available under the time based segment division scheme provides the length of the segment playout duration in milliseconds. This also implies that the segments are loosely coupled in the session, and described independent of the channels at this stage.

Next we examine the channel specifications in the SDP file. We have chosen two channels with different schedules.

m=video 1234 RTP/AVP 33 This is a standard SDP media description line, and specifies the media type (video), port number (1234), profile (RTP/AVP) and media payload type (33) that corresponds to the MPEG-2 TS payload type.

c=IN IP4 239.255.42.42/7 The c= line specifies the connection information for this media, and the address type and parameters. This is the multicast address of the channel.

a=channel-schedule:[1] The channel schedule custom attribute is used to map segments onto channels. This channel only involves a single segment and is quite simple.

a=channel-schedule:[2,3] This channel schedule contains a schedule for segments 2 then 3. This means that the client will first receive data from this channel into segment 2, then segment 3 before the channel cycle completes and segment 2 begins again. As we will see later, the segment id is carried in the RTP Extension Header, allowing the client to switch sinks if the client starts receiving data in the middle of a server transmission, which is likely to occur.

```
v=0
o=- 1240266009982954 1 IN IP4 172.16.3.208
s=The Movie
i=The Movie
t=0 0
a=tool:SSF Cautious Harmonic Broadcasting Controller
a=type:broadcast
a=control:*
a=segment-count:24
a=channel-count:23
a=segment-description:1:25469:27207
a=segment-description:2:25489:27250
...
a=segment-description:23:25708:27500
a=segment-description:24:25578:27335
m=video 1234 RTP/AVP 33
c=IN IP4 239.255.42.42/7
a=channel-schedule:{[1]}
m=video 1236 RTP/AVP 33
```

```
c=IN IP4 239.255.42.43/7
a=channel-schedule:{[2,3]}
...
m=video 1276 RTP/AVP 33
c=IN IP4 239.255.42.63/7
a=channel-schedule:{[23]}
m=video 1278 RTP/AVP 33
c=IN IP4 239.255.42.64/7
a=channel-schedule:{[24]}
```
Listing 6.2: Sample SDP file

We will now go on to provide an overview of the server components.

6.2.3 Server side components

The key server side components of the SSF are included in this overview, we also include some of the live555 library components we use in the following sections for completeness.

SSFServer is the component responsible for managing the different controllers, and the address factory. It is responsible for reading and parsing the MSD files provided and use the information to create the correct controllers.

SSFAddressFactory is responsible for managing the server address space, keeping track of available multicast IP addresses. This address factory is a server-wide instance, making sure all the controllers running in this server have unique addresses.

SSFController forms the base class of all new controllers. It implements functionality that is common to all sub-classed and specialized controller types, like creating an index file for the media. The intention is that any new periodic broadcasting algorithm will be created and implemented as a subclass of this controller.

SSFCHBController encapsulates our implementation of the Cautious Harmonic Broadcasting algorithm. This class also serves as an example of how to use the SSF to implement new periodic broadcasting algorithms. This controller will also be used in the experiments performed in this thesis.

SSFSegmentSource encapsulates the behavior and state management for a segment. This is the main *source* object in the frameworks server side.

SSFMPEG2TransportStreamFramer is a part of the channel and is positioned between the *source* and the *sink*, and acts like a filter. The main responsibilities for the framer is to frame data from the source in appropriate frame sizes, and to calculate the frame transmission time. This component is where we manipulate the channel bandwidth relative to the media playout speed, and is thus a key component in our transmission scheduling.

SSFRTPSink is part of the channel and is the server side sink. It is responsible for encapsulating the data into RTP packets and schedule the transmission of these using the calculations obtained from the framer component. It is the component that transmits the RTP packets.

channel_t structure is the structure used to organize and group all the channel components together. The most important components are the framer, sink, channel schedule, ports and group socket components.

ByteStreamFileSource encapsulates the primitives necessary to read an MPEG-2 TS file as a byte stream. It includes operations such as reading, seeking, and also allowing control over the block size of the read operations.

MPEG2IFrameIndexFromTransportStream uses the I-frames in an MPEG-2 TS file to calculate and create an index file for the media. This index file is later used to lookup the byte offset in a media file based on a given time instance in the media playout duration, as well as providing information on the total media duration.

6.2.4 Client side components

In this section we provide a brief overview of the most important client side components in the SSF. The use and interaction details of these components are detailed in the client implementation section below.

SSFRTPSource is the source component that receives the RTP packets, on the addresses it is listening to. After receiving packets it will call its RTP Extension Header handler before delivering the RTP packet to the registered sink. It also contains a buffer for reordering any out of order packets it receives.

SSFSegmentSink receives the media data from the source and is responsible for correct reassembly of the received segment.

clientChannel_t structure encapsulates and groups all the client channel components. The most important components are the source, sink, and the channel schedule.

This concludes the overview of the SSF framework components, and we now move on to the server and client implementation.

6.3 Server implementation

In this section we will go through the composition and execution-flow of the server side SSF, in an applied context. We describe our implementation of the cautious harmonic broadcasting algorithm and how it interacts with the general parts of the framework. This aims to provide insight into how the design goals and requirements have been realized. We believe that by describing the framework in the context of implementing our CHB algorithm, it will also be a good guide to approach implementing other periodic broadcasting stream scheduling algorithms.

6.3.1 Create the SSFServer

We start by creating a a common live555 *BasicUsageEnvironment* and *BasicTaskScheduler* so that we are able to run our server on the building blocks of the live555 library. Next we read the MSD file provided into a buffer and pass it along to our *SSFServer* constructor, along with the live555 environment we have set up. Further we need to initialize our server.

The *SSFServer* will initialize its parameters by parsing the provided MSD file content, line by line until it is completely parsed. The parsing process will also contain validations making sure any unknown lines in the MSD file are rejected, while also making sure that all the required parameters are present in the MSD file. The most important parameters read from the MSD file are the following items:

The filename and path of the media that is to be transmitted, typically the video file.

The scheduling algorithm we are going to use, this determines the type of controller we create. In this case the CHB algorithm is the one we are going to examine.

The segment division scheme , both the time based scheme, and the byte based scheme is supported. In our case we are going to examine the time based scheme.

The maximum client waiting time given in seconds, used to calculate the number of segments we require to stay within the bounds of the specified maximum client waiting time.

The *SSFServer* also creates a singleton [23] instance of the *SSFAddressFactory* class that manages the multicast IP-addresses and port numbers the controllers may request. This ensures that we do not get any addressing conflicts when creating multiple controllers for streaming different medias at the same time. In this description of the server implementation, only one media will be streamed, so addressing conflicts are not an issue, but in a more realistic running scenario the server would typically be streaming more media files at any given time.

After setting up our environment and validating the MSD file parameters, the *SSFServer* now needs to create the controller type specified in the scheduling algorithm parameter it was provided with from the MSD file. In our case the scheduling algorithm selected is CHB, and the *SSFServer* creates an instance of the *SSFCHBController* to handle the remaining steps necessary to stream the media file to clients. The newly created controller was given all the required parameters, including pointers to access the *SSFAddressFactory* singleton class. We now need to initialize the *SSFCHBController*, before it is executed.

6.3.2 Initialize the new controller

In this section we will examine the steps involved in initializing and preparing the *SSFCHBController* for execution. The scenario selected will examine the initialization of a time based segment division scheme, where no prior media index file exists. We will comment on the sections that would be handled differently in a byte based segment division scheme, and the case where a media index file is already present.

Index file creation

In our scenario we use the time based segment division scheme, and we require an index file for the media we are going to stream from our server. The index file requirement is necessary because we need to look up the TS packet number based on a given time in the media playout timeline, to obtain the corresponding byte offset within the media file.

In our implementation the logic responsible for generating the index file from a MPEG-2 TS file, is located in the *SSFController* base class. The rationale for placing the implementation logic related to creating the index file in this base class, instead of placing it in the subclass *SSFCHBController*, is that index file creation is carried out in the same way for all the different stream scheduling algorithms that may be implemented.

The steps necessary to create the index file begins by opening the input media file using a *ByteStreamFileSource*, which contains all the basic operations we need to read from the source media. We also make sure the input source will read data from the input media in 188 byte blocks, that corresponds to the size of the TS packets. Next we create an instance of the *MPEG2IFrameIndexFromTransportStream* class, and assign the input source we created as its source of media data to be indexed. As the class name implies the indexer class creates the index file using the I-Frames in the MPEG-2 TS file, and the granularity of the index file thus depends on the I-Frame frequency in the media file. The last live555 library class necessary to create our index file is an output sink, in our implementation the *FileSink* class is the most appropriate.

After creating the necessary classes and linking them together, we need to schedule the indexer to run, in our task scheduler. This is done in the normal live555 way, by calling the created file sink *startPlaying* method. We also require notification when the index file is created, and therefore register a *afterIndexFileCreation* callback function as a parameter to the output sink *startPlaying* method.

At this point we actually return from the *SSFCHBController* initialization process, back to the main server, where the task scheduler is started. This is necessary because the rest of the initialization process requires the index file in order to function. Before returning to run the task scheduler, we set up sufficient state variables, so that the initialization process can continue where it left off. This behavior is contrary to the one we face when an index file already exist for this media file. We deploy this behavior because index file creation is a lengthy and processor intensive task for large media files, and thus we reuse the index files to save system resources.

The initialization process will logically continue from this location in the sequence, wether it progresses directly, or reenters via the provided *afterIndexFileCreation* callback function we have set up.

Set up algorithm

This section explores the steps we take in order to set up our CHB algorithm controller, so that all the necessary data-structures and classes are created and initialized.

We obtain the maximum client waiting time as a parameter from the class that created the CHB controller, namely the *SSFServer*, which again obtained this parameter by parsing the MSD file. The most important use for the maximum client waiting time is in determining the segment size, and thus the number of segments we require. We calculate the number of segments by dividing the media duration, obtained from the index file, by the maximum client waiting time. The number of segments is always an integer, so small deviations from the specified maximum client waiting time may occur.

After calculating the number of segments we calculate the number of channels we require in order to stream the number of segments we have calculated. In CHB the number of channels required is *numberOfSegments - 1* because segments 2 and 3 are both streamed over the second channel, making it the only shared channel.

Next we will examine the segment creation process.

Create the segments In this paragraph we elaborate on the steps we take in order to create the segments from the media file. First we explain the initial setup, next the required steps for each segment are explained, lastly the special considerations for the last segment is described.

We start by allocating enough storage space for the calculated number of segments, and initialize our segment data structures. We have also defined a constant read block size, this illustrates the preferred data block size we obtain from each segment when requesting data from it. The data block size is a multiple of the TS packet size of 188 bytes, and as many TS packets we can fit into a network packet while avoiding fragmentation. This size depends on the underlying network infrastructure, and in our case we use *188 * 7 = 1316 bytes* as our read block size, as this allows for header data addition and still does not exceed the 1500 byte ethernet frame limit in our network.

As all segments in CHB are of equal size, we calculate the duration for each segment by dividing the media duration with the number of segments. This gives us an exact segment duration, but the actual duration for each segment will not correspond exactly to this calculated duration. The reason for this deviation is the granularity of our index file. When querying our index file for the TS packet number corresponding to a given time in the media duration, it will return the closest match, and this may lead to small differences in the segment sizes. For each of the segments we do the following set of operations, by running a loop with one iteration for each segment we need to create.

We start by calculating the segment *startTime* and *stopTime*, and use these points in time to lookup the corresponding TS packet number in our index file. Next we need to use this offset information to calculate the *startByte and stopByte* in the overall media file that corresponds to the current segment. This byte offset is obtained by multiplying the TS packet number by the TS packet size of 188. The byte offsets will later be supplied to the *SSFSegmentSource* class and will be used as bounds for reading data. We also note the TS packet number corresponding to the *stopTime* for this segment. We use this value in the next iteration of the loop to make sure we do not end up with any segments of zero size. We may end up with segments of zero size if the distance between the segments are shorter than what the index file granularity supports.

Next we create an instance of our segment implementation class, *SSFSegmentSource*, supplying it with the parameters for *segmentID*, *startByte*, *stopByte*, and the preferred read block size. After creation we also assign the segment duration in milliseconds to the *SSFSegmentSource* instance, this will be used later when we generate the SDP file for the server session. We use the segment duration in the SDP file mainly to allow the client to check deadline conformance for each segment, i.e that the segment is received on time at the client side.

The last segment we create will typically not be of the same size as the other segments, but will in most cases be slightly smaller. The exception is when our index file contains exact data for all the *startTime and stopTime* values we ask for, which is unlikely to occur.

We have now created all the segments we require and we move on to creating our framers.

Create the framers The framer is placed between the source and the sink, and acts like a filter, where it runs calculations of the presentation time for the data frames. These calculations are used by the RTP sink to schedule transmission of the data. In our design the framer and the RTP sink together make up the channel concept, and handle the responsibilities we have assigned to a channel.

We start by allocating space for the framer data-stuctures, one framer for each channel. Each of the *SSFMPEG2TransportStreamFramer* instances is assigned a bit-rate factor, used to scale the channel transmission speed relative to the media playout speed.

In CHB the first two channels are allocated as full bit-rate channels, where the transmission speed is not scaled in any way. this mean they are set up with a bit-rate factor of 1, that implies full playout speed referred to as b. The framers for channel 3 to N, where N is the number of channels, the scaling of channel i is set to b/i.

Each of the framers are assigned its initial input source, i.e the first segments in the schedule for this channel. All the channels except for channel 2, will only have one segment in its schedule in CHB. This means that channel 2 is initially assigned segment 2, while segment 3 is not yet assigned to any channel, but will be scheduled on channel 2 after segment 2 is completed.

Next we examine the RTP sink creation process.

Create the channels As we saw in the framer section, a channel is composed of both a framer an a sink. In our implementation the sink is of the class *SSFRTPSink*, and is connected to the corresponding framer.

We start by allocating space for the *channel_t* structure used to store all the required parts of the channel. For each of the channels we do the following steps in order to create and set up the channel.

We use the *SSFAddressFactory* singleton class, provided as a parameter to the *SSFCHBController* class, to obtain a multicast IP-address and an RTP/RTCP port-number pair for the channel. This endpoint information is then used to create the *Port* and *GroupSock* instances we require, and the results are stored in the *channel_t* structure.

Next we create the *SSFRTPSink* instance used to handle the RTP related parts of this channel, such as transmission scheduling and actual transmission of RTP packets on our network interface. The *SSFRTPSink* is provided with significant parameter in its constructor that allows the sink to transmit RTP packets that still have room for more data. This behavior is contrary to the standard *live555* library RTP sink implementations, which will normally wait for a RTP packet to fill up before transmitting it on the network interface. The rationale for allowing transmission of non-full RTP packets, is that we cannot allow data from different segments in the same RTP packet. This may occur in our implementation of CHB when the last part of segment 2 on channel 2 is sent ,and does not completely fill the RTP packet. In the default *live555* library RTP sink implementation the current RTP packet would then also receive a part of the initial data from segment 3 before transmission. This would break our designed use of the RTP extension headers, causing data corruption.

Next we create an RTCP instance and assign it to the channel. We currently do not handle the RTCP reports beyond the default actions of the *live555* library, but we decided to add support for RTCP reports in order to support extending our framework.

In the final part of creating and initializing our channels, we assign the RTP extension header provider callback functions, assign the corresponding initial segment to each channel, and set up the channel segment schedules. In CHB the only channel with a multi-segment schedule is channel 2.

Finally each of the channels are iterated to create the channel specific SDP lines, this includes the channel connection information along with the channel schedule.

Next we move on to examine the SDP file creation.

Generate the SDP file for the clients The final step in initializing our *SSFCHBController* is generating the customized SDP file, used by our clients to be able to receive the media.

The first part of the SDP file is RFC standardized [20] session information, media information, etc, we refer to the description provided at the beginning of this chapter for details on the SDP format. The custom attribute parts of the SDP file are created dynamically, from the controller state. The essential parameters are specified in the following list.

The number of segments are written to help the client in initial allocation of data structure sizes.

The number of channels are also written to aid the client side in its initial data structure allocations.

The segments are custom attributes used to provide metadata about the segments, describing the segmentID, segment size in the number of TS packets, and the segments playout duration in milliseconds.

The channels are described in the standard SDP *m=* type of description, but have an additional line for each channel describing the channel schedule.

This was the final step in our *SSFCHBController* initialization, we now move on tho the necessary actions required to run our controller.

6.3.3 Running the controller

The final step in the process of initializing and running the server is to set up the controller to be executed by the task scheduler and set the appropriate callbacks for the *afterSegment* handler, that is called when a segment is done playing all its data, and we need to swap input segments instead of just looping the active segment.

For each of the channels we have set up, we need to call the channel sinks *startPlaying()* function to register it in the task scheduler. We call the *startPlaying* function on each channel with the appropriate framer as the source of the media data, and a callback function to be executed upon data completion. This data completion callback is not called in the normal running channels, as the segment will handle the seeking upon reaching its stopByte position, and perform the looping or call its *afterSegment* handler if any is registered.

For channel 2 that is used to stream both segments 2 and 3, it is necessary to switch segments when the active segment is completely transmitted. This is done by registering an *afterSegment* callback function, in the segment, so instead of looping the segment, the callback function is executed, passing control from the segment itself to the controller.

The task scheduler will now start playing the different segments as soon as it is engaged.

6.3.4 Runtime server events and handlers

This section will examine some of the key events and processes that occurs when the server is running. This indicates that the task scheduler has been engaged and is processing tasks. We will examine the handling of segment switching and how we obtain and use the RTP Extension Headers on the server side.

The RTP Extension Header provider function

Each of the channels have the RTP sink responsible for encapsulating and transmitting our media data as RTP packets. Since we decided in our design to use RTP Extension Headers to carry segment metadata, we need to obtain these metadata each time a new RTP packet is created. This section will elaborate how we obtain the RTP Extension Header data, for each outgoing RTP packet, and the content and intended use for each of the metadata fields.

In our CHB implementation all the channels have a RTP Extension Header provider function registered. When a new RTP packet is created on the channel, and it has a registered RTP Extension Header provider function registered, the RTP packet will be initialized with the required RTP Extension Header present bit set. We do not request the RTP Extension Header data upon RTP packet initialization, we instead set aside space in the packet and note for later the positions where we need to insert the extension header data. The reason we postpone obtaining the extension header data is that we need to obtain the actual payload data first, as the input source segment may be changed upon our segment data request.

After we have obtained the next frame from our input source, we call our registered RTP Extension Header provider function with the following four parameters:

The active segment This is the source of our extension header data.

The profile specific data This field is used to carry the segment ID for the active segment. This provides the receiver with information on segment affinity for this RTP packet.

The size of the extension header data In our CHB implementation this size is always 1. This indicates the number of 32-bit blocks our extension header payload data will consume.

The extension header data payload The payload for the RTP Extension Header contains the segment index in the current RTP packet. The index specifies the TS packet number that begins this RTP packet media payload. This provides the receiver with enough information to be able to write the RTP payload data to the correct position in the received segment. Since we only provide the starting position, the first TS packet number in this RTP packet, it is the receivers responsibility to calculate the number of TS packets contained in this RTP packet.

This RTP Extension Header usage gives the receiver information on segment affinity, and indexing information used to reassemble the segment correctly.

The *afterSegment* and *afterCycle* callback functions

Segments that are part of a channel schedule, meaning they are being streamed on a channel that has more than one segment it is responsible for streaming, will have an assigned *afterSegment* function. The *afterSegment* function is called from within the *SSFSegmentSource* instance when it has reached the end of its data. Segments that are not part of a channel schedule, meaning the channel only streams one segment, does not have an *afterSegment* function assigned to them. When these kinds of segments are done playing all their data, they handle this internally by looping the segment, and starts reading from the segment beginning again. Next we will detail the steps carried out by the controller upon a call to the *afterSegment* function.

When the segment has reached the end of its data, and the segment is complete, it will call its registered *afterSegment* function. The *afterSegment* function evaluates the segment according to the channel schedule, and assigns the next segment scheduled to be played on the channel. If the channel schedule has completed an iteration of all the segments in its schedule, the channel *afterCycle* callback function will be called. This *afterCycle* callback is not in use for any specific action in CHB, but may be necessary in other algorithms. Also when the channel cycle is complete it will start playing from the beginning of the schedule again.

After the new segments has been assigned to the active channel, the channels framer is assigned the new segment as its input source of media data.

The channels RTP Extension Header provider function is updated with the new active segment.

The newly activated segment is assigned an *afterSegment* callback handler function, so that we again can switch the segment after it is completed.

6.4 Client implementation

In this section we will describe a client implementation, capable of receiving the media streamed by our server implementation. The main focus in this description will be on the assembly and interaction of the most important client side components, as well as important run-time events, and how they are handled. We start by describing the client creation and initialization process, and proceed with the run-time event processing.

The client implementation is a stand-alone client, and we do not perform any integration with existing media players. The rationale for making a stand-alone client implementation is that it allows us to focus on the framework components alone, as well as adding functionality to aid in our experiments and measurements described in chapter 7. The implementation does however cover the client responsibilities, and the same set of responsibilities would also have to be fulfilled when integrating with an existing media player, and thus forms a good starting point for such an integration.

6.4.1 Client creation

The first steps in the client creation is to create instances of the live555 library *BasicUsageEnvironment* and *BasicTaskScheduler* classes which allows us to rely on the live555 library management and task scheduling. After this basic initialization we need to read the SDP file generated by our server implementation into a buffer, so that is is available for parsing.

We do not describe any mechanism for transferring the SDP file from the server to the client, but note that this must be performed using a reliable transfer protocol, like HTTP running over TCP/IP.

After the SDP file is available in memory, we move on to the initialization process.

6.4.2 Client initialization

In this section we describe the most important tasks performed during client initialization. This includes parsing the SDP file describing our media session, and creating the necessary components and data-structures, enabling us to receive the media being streamed from the server.

SDP session level parsing

We begin by parsing the SDP session level parameters, and we will focus on our custom attributes. The following attributes are the key custom attributes:

The number of segments is specified by the *a=segment-count:<count>* line in the SDP file, and is used to initiate the data-structures for our *SSFSegmentSink* instances.

The number of channels is specified by the *a=channel-count:<coint>* line, and is the basis for initiating data-stuructures for *SSFRTPSource* instances.

Each segment is described by a *a=segment-description:<segment id>:<segment TS packet count / size>:(optional)<Segment playout duration in milliseconds>* line, and for each of these descriptor lines we create a *SSFSegmentSink* instance with the associated data fields assigned.

SDP session level validation

After the session level attributes are parsed and all of the segments created, we need to perform validation of the required attributes described above. This validation is performed to ensure that the SDP file was generated for our client implementation, and not a standard SDP file. If the SDP file does not contain all of the custom attributes, the

SDP file was intended for a standard media player, or has otherwise been corrupted, and the client aborts the process.

SDP media level parsing

After the SDP validation, we shift our focus to the media level SDP parameters, the lines that start with *m=*, we refer to listing 6.2 for our sample SDP file. For each of the media level SDP lines we perform the following steps in order to initiate our channels:

- Create a *clientChannel_t* structure used to store and group all the necessary components and state related to this particular channel.
- Extract the media type (video), port number, and payload type from the *m=* line, and the media connection IP multicast address from the *c=* line.
- Create the ports and group sockets for this channel based on the data obtained in the previous step.
- Create the *SSFRTPSource* component for this channel, and assign the port, and group socket component to it. We also create the RTCP instance here.
- Finally we extract the channel schedule in the *a=channel-schedule:* line, and use this to create a channel schedule data-structure, which is assigned to the channel.

We now have all the components necessary to receive streaming media from our server implementation, and we go on to linking these components together, and assigning callback functions and handlers.

6.4.3 Client channel setup

We now describe the client channel setup, where we link together the different data structures, and assign our handlers and callback functions, in order to make the client ready for execution. The following operations are performed on each channel:

- We initialize the channel schedule management variables to the schedules initial values.
- Based on the initial channel schedule, we assign the specified segment as the active segment.
- We assign the *afterSegment* callback function on the channels active segment, this function will be called by the segment when it is done receiving the entire segment.

- We register the RTP extension header handler function on the channels RTP source, which is called when a RTP packet arrives, and provides the segment metadata carried in the RTP extension header. (The metadata consists of the segment id the data belongs to, and the offset into the segment for the current data.)
- Finally we call the channel sinks *startPlaying* method, which registers the channel in the task scheduler, so that it will receive data.

We now engage the task scheduler, and begin executing our client.

6.4.4 Runtime client events and handlers

In this section we examine the most important client runtime events, and how they are handled.

RTP packet arrival and RTP extension header processing

When a RTP packet arrives on a channel RTP source, it calls the registered RTP extension Header handler. The RTP extension header handler takes the following four parameters:

- The channel the packet arrived on
- The RTP extension header specific data:
- The profile specific data used to carry the segment id the data belongs to.
- The size of the RTP extension header data in 32-bit blocks. This is always 1 in our implementation.
- The RTP extension header payload carrying the index of the first TS packet in this RTP packet, used by the segment.

We first check that the RTP packet that has arrived belongs to the active segment sink on this channel. If it does not, we suspend the current segment and call the assigned*afterSegment* handler, which will assign the correct segment to the channel according to the channel schedule. This may occur if we start the client in the middle of a server schedule, and thus have different schedule progression status in the client and server endpoints. We will examine the details of the *afterSegment* handler in a few moments.

After we have verified that the correct segment is assigned to the channel, we set the index carried in the RTP extension header in the segment so it will receive the data in the correct position relative to the entire segment. This is also a good time to perform any statistics collection, and print any status information we might want to display during testing, like the progress of the different segments.

We now move on to examine the details of the channel schedule advances and segment completion handling.

Segment completion and channel schedule advancement

When a segment is complete, i.e all the data belonging to the segment is received, the *afterSegment* callback function is called.

This function is also called if we start receiving data in the middle of a server schedule and the wrong segment sink is currently active on the channel. This condition is detected in the RTP extension header handler as described in the previous section. This means we need to "hot-swap" the segment sink, but we already have data in the RTP source buffer, ready to be copied into the segment sink registered. This drove the need to change the RTP source to support switching sinks while in progress of delivering data to the active segment sink that requested the data. To accomplish this we needed to make some changes in the live555 library itself, because it does not allow hot-swapping of the sink while it is in the process of delivering the received RTP packet to it.

The *afterSegment* function has one parameter when called from within the segment or from the RTP Extension Header handler, this parameter is the the channel this segment is currently active on. We proceed by describing the *afterSegment* operations, which involves examining the channel and session schedule.

We first check if we have another segment scheduled on this channel cycle, and if there is another segment in the channel schedule we perform the following operations:

- We first check if the next segment on the schedule is complete, if it is we skip to the next segment in the schedule, and starts this list again.
- We set the segment that was next in the schedule as the active segment on the channel.
- We set up the *afterSegment* callback function in the segment that we assign to the channel.
- We assign the RTP extension header handler function to the channel, passing in the correct parameters.
- Finally we register the newly assigned segment in the task scheduler by calling its *startPlaying* function, and return from the handler function.

If we do not have any segments ahead in this channel cycle, we check if we have another channel cycle that we need to execute. This will never happen in the CHB algorithm, because all channels have only one cycle. But if it happens we start playing the first segment in the new cycle, by doing the same steps described in the listing directly above this description.

If we are at the end of the channel cycle, we have reached the end of the channel schedules, and we check our entire channel schedule for incomplete segments:

- We check the segments in our schedule for completeness, starting at the beginning of the channel schedule.
- If we find any unfinished segment we start playing this segment by assigning it to the channel and repeating the steps mentioned above, when assigning a new active segment.

If we have received all segments scheduled on this channel we perform the following operations:

- We shut down this channel and its RTP source and RTCP instance.
- We check if there exist any overall segment that is not complete, if any exist we return to the scheduler and the remaining channels keep receiving.
- If this was the last segment to complete, we call our completion handler, *afterSession* detailed in the following section.

6.4.5 Session completion

When all the segments are completed, the *afterSession* handler is called. This function rearrange all the segments, so that they all start at the beginning of a file, and merges all the segments into a full media, so that we can use a binary diff tool to examine the reception result. It also prints some helpful statistics, like average bandwidth consumption for each segment, bytes received for each segment, the transfer time for each segment and the total duration of media reception, used for determining on time delivery.

6.5 Summary

In this section we have detailed our stream scheduling framework, in the context of the CHB stream scheduling algorithm. We have examined how our design requirements have been realized and will now move on to examining the performance of our implementation through a series of experiments.

Chapter 7

Experiments

In this chapter we perform a series of experiments using our CHB implementation, as well as some tests designed to test the SSF transmission scheduling mechanisms. The purpose of the experiments is to examine the performance and operation of our implementation, using both constant bit-rate (CBR) media, and compressed media with a variable bit-rate (VBR).

7.1 Test environment

This section describes our test environment including the testbed network, media files used in the experiments, and our selection of tools and techniques.

7.1.1 Media description

The media used throughout our experiments is based on the open source movie *Elephants dream* [24]. The original media file has been transcoded into two test media files, both with an average bit-rate of 1400 kbps and a duration of 654 seconds. The two test media files are transcoded using the FFmpeg [25] tools, into a CBR and a VBR encoded media. The CBR encoded media is not fully constant bit-rate due to the MPEG-2 TS encoding, but as close to CBR as the FFmpeg tools could produce.

7.1.2 Testbed

The testbed is set up using three computers, as well as a management station used to control the experiments. The three computers are a server, used to serve the media running our CHB implementation, the client used to receive the streamed media over IP multicast, and a router and network emulator. We have chosen not to inflict any network packet loss or additional delays, this is due to our focus on testing our framework and CHB implementation for correctness, and perform accurate measurements. The testbed diagram is shown in figure 7.1.

The network is divided into separate IP subnets for the client and the server computers, connected by the router computer using different network interfaces for the different subnets. The router is running the XORP [26] open source routing platform, configured with IGMPv3 [21] and PIM-SM [22] multicast management and forwarding protocols. All the traffic between the client and server computers goes through the router, and the router is thus also set up as a monitor and emulator. The monitor is used to capture network traffic between the client and server computers.

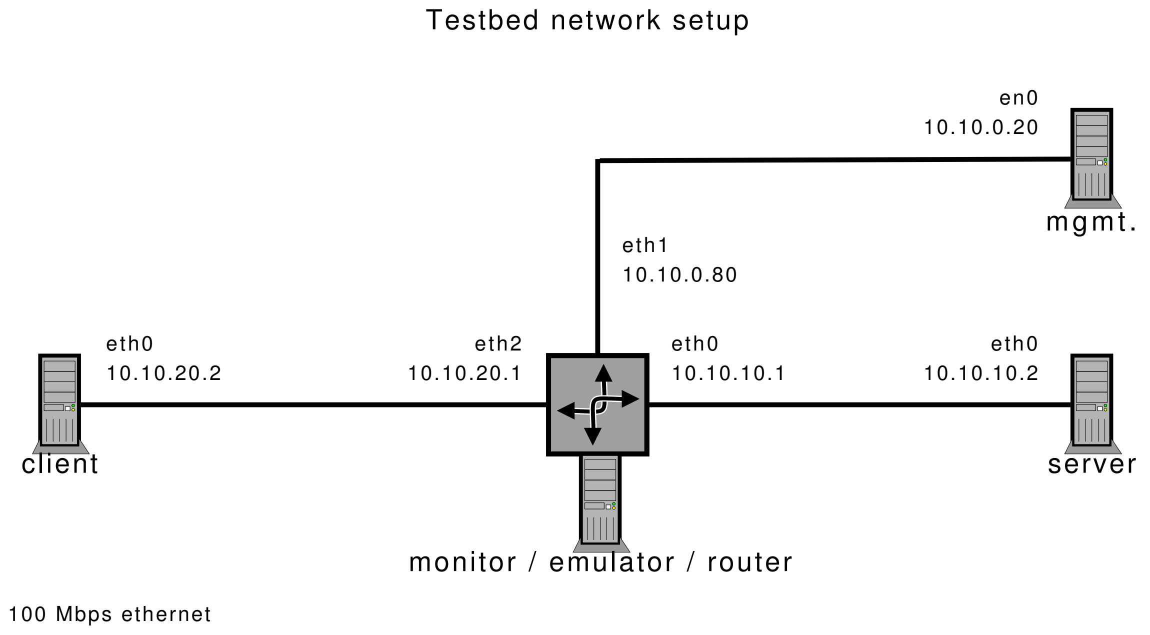

Figure 7.1: Testbed network

7.1.3 Tools and measurements techniques

The monitor computer is set up to use tcpdump for traffic capture, which are saved to log files on disk storage. During some of the experiments the disk IO rate at the server and client computers can become high, and by performing the traffic capture on the monitor computer we ensure that the capture does not interfere with the implementation performance. The monitor also have separate network interfaces for the client and the server computers, allowing more complex traffic capture operations.

The SSF client implementation is also equipped with functionality to gather statistics, and capture aggregated data during operations, like the time it used to download each segment, the average bit-rate for each segment, which again can be used to determine if all the data arrived on-time for consumption.

The experiments are controlled by batch and python scripts, and python scripts are used to process the collected data, and generate gnuplot data files.

7.2 Scheduling performance

In the following sections we are performing a series of experiments in order to test the stream scheduling framework mechanisms and our CHB implementation. Most of the tests use a client waiting time set to 60 seconds, and are all performed using both the VBR and CBR media files described in the previous section. All the tests related to our CHB implementation are performed using the time based segmentation scheme.

7.2.1 Startup delay

Description

The CHB stream scheduling algorithm assumes that all the channels start transmitting the different segments at the same instance in time. This assumption is not possible in our implementation, as we perform our experiments on a single CPU computer, making all our instructions execute in sequence. For this reason we expect a startup delay between the different channels in our implementation, and we want to measure the delays, in order to examine their significance.

The CHB algorithm also assumes that when segment *N* is consumed by the client, segment *N+1* is downloaded and ready for consumption. This means that the transmission scheduling for the different segments needs to operate in a way that supports this assumption. Because we expect a startup delay between our streams, we need to keep this assumption supported to the best of our abilities. In our CHB implementation the transmission scheduling mechanism schedules the segments in the reverse order, meaning that the first RTP packet of segment *N*, is scheduled for transmission after the first RTP packet of segment *N+1*. By doing this reverse order scheduling of segments, we ensure that our startup delay does not impact the assumption made by CHB, because the first segment to be consumed by the client, is the last segment in the transmission schedule.

It is also interesting to examine how the startup delay manifests itself when we vary the number of channels, and if it remains predictable across different channel numbers.

The measurements are performed by capturing the network traffic from the streaming server, and plotting the delays.

Results

In figure 7.2 we see 9 streams plotted in reverse order, and the stream delay. The first stream is not extended as a line in the plot, because this is our source of the first RTP packet, and is what the other streams are measured against. We observer a linearly growing startup delay as the stream number advances, this is in accordance with our expectations for a sequential transmission scheduling approach. The values for the different startup delays, as well as a calculated average are available in table 7.1. We see that the average startup delay, when adding another stream, is close to 81 microseconds, and have a small impact on the overall scheduling.

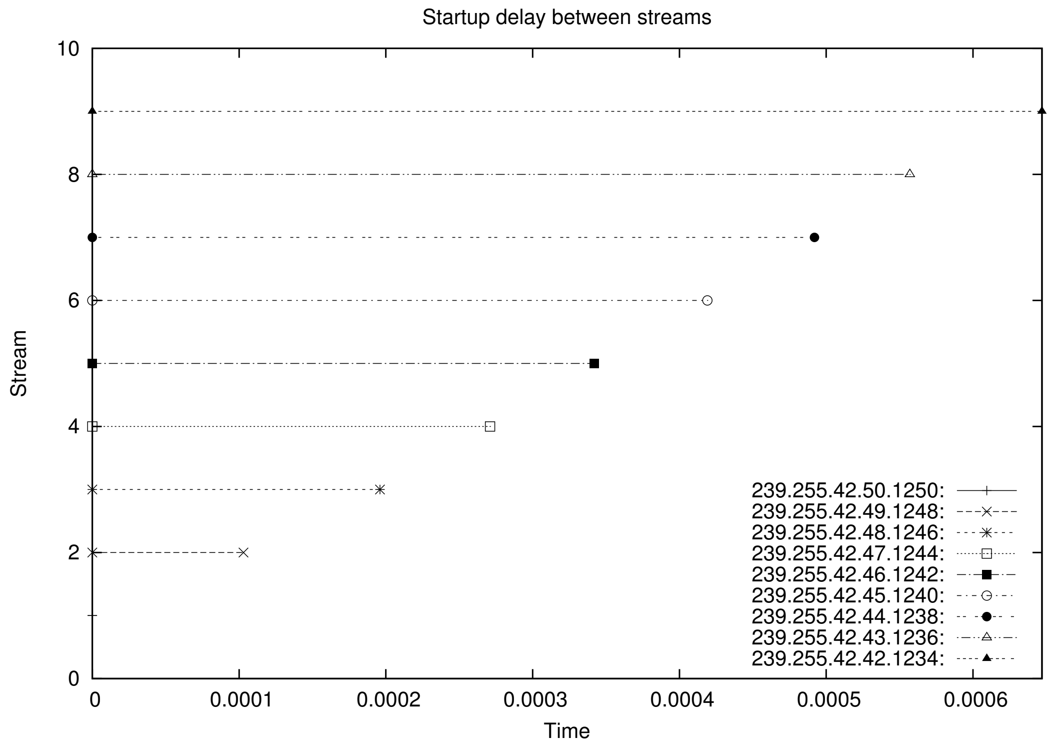

Figure 7.2: Startup delay for 9 streams

We also wanted to examine if the startup delay remains predictable when increasing the number of streams that is scheduled for transmission. In figure 7.3 we have plotted the startup delays for 42 different streams, and the delays also here follows a linear pattern. The only observable difference is between stream 37 and 38, where the startup delay appears slightly larger than between the other streams.

The cause for this extra delay may be related to disk IO scheduling or other operating system related delays. The difference is not as significant, that we wish to examine it any further.

We have also performed the same startup delay measurement using 325 different streams, the results are plotted in figure 7.4. Here we observe the same linear increase in startup delay as in the other figures. As far as our measurements have explored, the startup

Startup delay for 9 streams		
Stream	Delay (sec)	Difference from previous stream (sec)
1	0.000000	N/A
2	0.000103	0.000103
3	0.000196	0.000093
4	0.000271	0.000075
5	0.000342	0.000071
6	0.000419	0.000077
7	0.000492	0.000073
8	0.000557	0.000065
9	0.000647	0.000090
Average between streams: 0.000080875 (sec)		

Table 7.1: Startup delay for 9 streams

delay pattern remains predictable, by growing in a linear way as more streams are added.

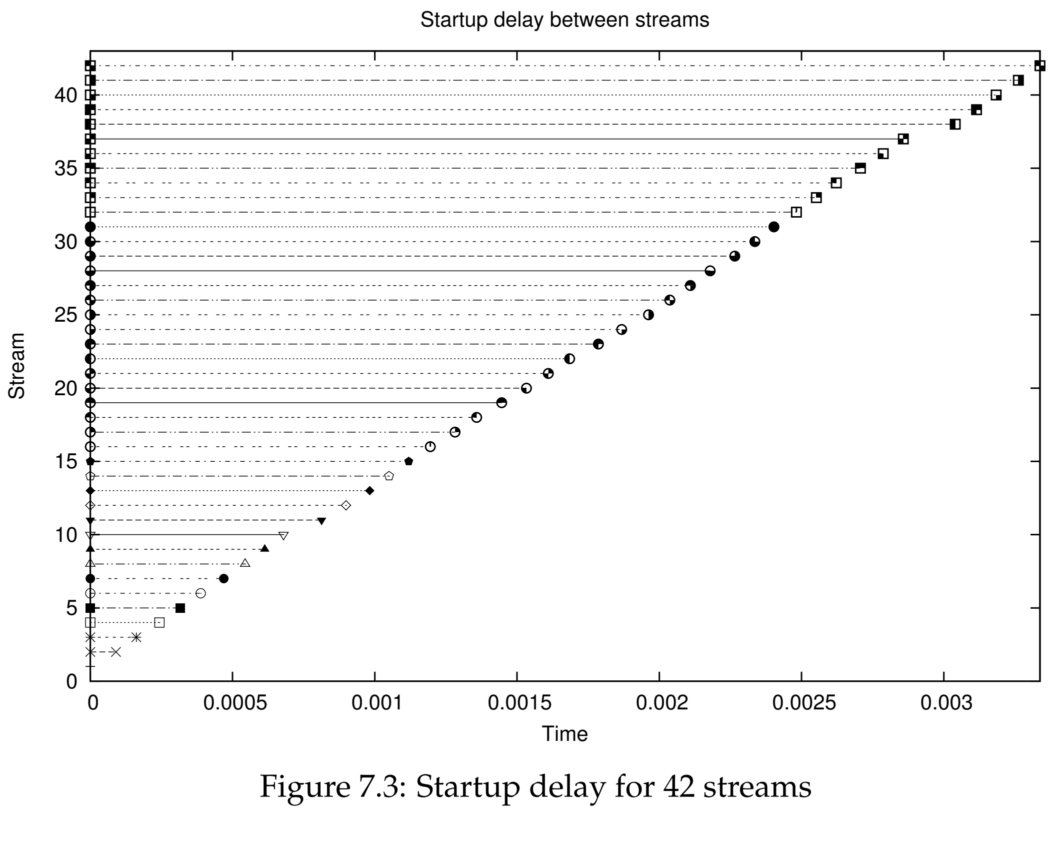

Figure 7.3: Startup delay for 42 streams

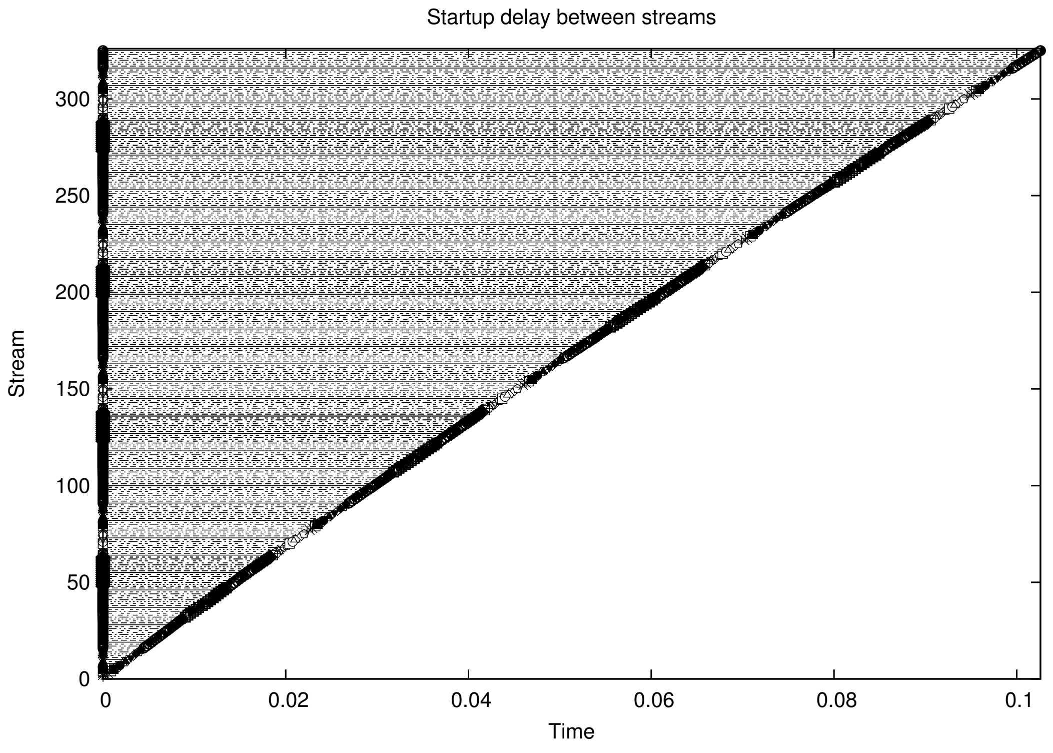

Figure 7.4: Startup delay for 325 streams

7.2.2 Transmission scheduling

Description

One of the key parts of our stream scheduling framework is the ability to correctly scale the transmission speed for the segments according to the stream scheduling algorithm. Therefore we think it is interesting to test this mechanism separately. We will examine the transmission duration for both a CBR and a VBR based media, in order to see if there is any differences in handling the two different media encodings.

For us to measure the time it takes in a comparative way, we have implemented a modified CHB algorithm, where all the segments consists of the entire media. The desired client waiting time is set to 60 seconds, resulting in 10 segments. Using our modified CHB algorithm, where each segment is equal to the entire media, makes it well suited for comparing the calculated transmission time and the measured transmission time for each segment.

The measurements are performed in the client application receiving the segments. The client notes the time of the first part of a segment, and notes the time when the segment is completely downloaded and ready for consumption. This is done for each of the 10 segments separately.

Full CBR media 60 sec client waiting time									
Segment length ms	SegID	Channel slow-down	Calc. trans-mission time	Meas. trans-mission time	Diff ms	Diff %	With heuris-tic	Diff ms	Diff %
653414	1	1.00	653414	653452	-38	-0.01	651170	2244	0.34
653414	2	1.00	653414	653452	-38	-0.01	651169	2245	0.34
653414	3	1.00	653414	653468	-54	-0.01	651191	2223	0.34
653414	4	3.00	1960242	1960342	-100	-0.01	1953511	6731	0.34
653414	5	4.00	2613656	2613793	-137	-0.01	2604658	8998	0.34
653414	6	5.00	3267070	3267254	-184	-0.01	3255843	11227	0.34
653414	7	6.00	3920484	3920692	-208	-0.01	3906977	13507	0.34
653414	8	7.00	4573898	4574148	-250	-0.01	4558144	15754	0.34
653414	9	8.00	5227312	5227599	-287	-0.01	5209310	18002	0.34
653414	10	9.00	5880726	5881068	-342	-0.01	5860458	20268	0.34

Table 7.2: Transmission duration for CHB media

Results

In table 7.2 we see the transmission data for each of the segments in the CBR based media. The table shows the segment duration, segment ID, channel slow-down factor according to the CHB algorithm, as well as the calculated and measured transmission durations. The deviation between the calculated transmission duration and the measured duration are also provided in milliseconds and in percentages. We will return to the *with heuristics* columns towards the end of this section.

We observe that the measured transmission duration is 0.01 % lower than the calculated transmission duration, for all the segments. This delay is consistent across all the segments, and is well within the bounds of the client waiting time of 60 seconds.

The first three segments are transmitted at the media playback rate, and no rate manipulations are performed on the channels transmitting these segments, as this is the rate the live555 library is designed to operate according to. These three first segments deviate by the same percentage as the segments that are transmitted over channels with rate manipulation. We therefore conclude that the manipulations performed on the channels operate as expected. The CBR media being transferred is not fully constant bit-rate, as mentioned in the media description of this chapter, and we believe this is the reason for the slightly longer measured transmission durations.

In table 7.3 we see the same measurements performed for the VBR media, and the differences are clearly more significant, as well as more variable. Although the VBR media delays are more significant, all the segments are still transmitted within the bounds of the client waiting time of 60 seconds.

Full VBR media 60 sec client waiting time									
Segment length ms	SegID	Channel slow-down	Calc. trans-mission time	Meas. trans-mission time	Diff ms	Diff %	With heuris-tic	Diff ms	Diff %
653414	1	1.00	653414	654764	-1350	-0.21	652502	912	0.14
653414	2	1.00	653414	654764	-1350	-0.21	652502	912	0.14
653414	3	1.00	653414	654895	-1481	-0.23	652632	782	0.12
653414	4	3.00	1960242	1964422	-4180	-0.21	1957680	2562	0.13
653414	5	4.00	2613656	2619257	-5601	-0.21	2610283	3373	0.13
653414	6	5.00	3267070	3274009	-6939	-0.21	3262806	4264	0.13
653414	7	6.00	3920484	3929041	-8557	-0.22	3915389	5095	0.13
653414	8	7.00	4573898	4583923	-10025	-0.22	4567941	5957	0.13
653414	9	8.00	5227312	5238786	-11474	-0.22	5220757	6555	0.13
653414	10	9.00	5880726	5893633	-12907	-0.22	5873369	7357	0.13

Table 7.3: Transmission duration VBR media

Discussion

As we observed in table 7.3, the measured transmission time for VBR media deviates more than for the CBR based media. We believe this is caused my the mechanisms in live555 responsible for calculating the transmission scheduling.

The transmission duration for the VBR medias last segment is almost 13 seconds, while this is within the client waiting time of 60 seconds, it is still a significant delay. The client waiting time is directly linked to the number of segments, and with a reduced client waiting time the number of segments would increase. This would imply that the transmission scheduling would no longer be able to send a VBR media segment on time for consumption, when lowering the client waiting time further. For lower client waiting time, assuming that the deviation rate at approximately 0.21 % would remain at its rate, the scheduling would fail. We will further examine the deadline conformance in a following section.

The *with heuristics* column in table 7.3 shows the same measurements, but with a heuristic constant added to the live555 scheduling mechanism. This heuristic constant involves adding an increase to the duration estimate performed on transport stream packets within live555. In table 7.3 and 7.2, we have added a 0.0035 % constant increase into the live555 duration estimation process, the results are shown in the *with heuristics* columns. In the VBR media case, this heuristic constant ensures that media will always arrive on time for consumption, at the expense of a slightly increased bandwidth consumption. The same constant is added to the CBR media table, to illustrate its impact on CBR media.

In experiments conducted, but not reported here, we observe that the live555 scheduling estimation is less accurate for smaller segments than for larger segments. This implies that the heuristic constant needs to be adaptive. We propose further research into establishing a mechanism for an adaptive heuristic constant, to guarantee on-time delivery for any media type and segment size.

Next we will examine the deadline conformance when client waiting time varies.

7.2.3 Client waiting time variation impact

Description

In the previous section, we examined the the transmission scheduling using a customized CHB implementation, designed for channel transmission scheduling tests. In this section we observed that our transmission scheduling mechanism experienced a delay rate that implies reduced accuracy when the segment size decreased. This decreased segment size is directly linked to an increasing number of segments, as a result of reducing the client waiting time.

These observations raise some concern, and we want to measure the impact of the client waiting time below 60 seconds.

When transferring our media file, we have a deadline for the on-time delivery of the entire media equal to the *media duration + client waiting time*. Using our 654 second media files, we proceed to measure the deadline conformance for varying client waiting time.

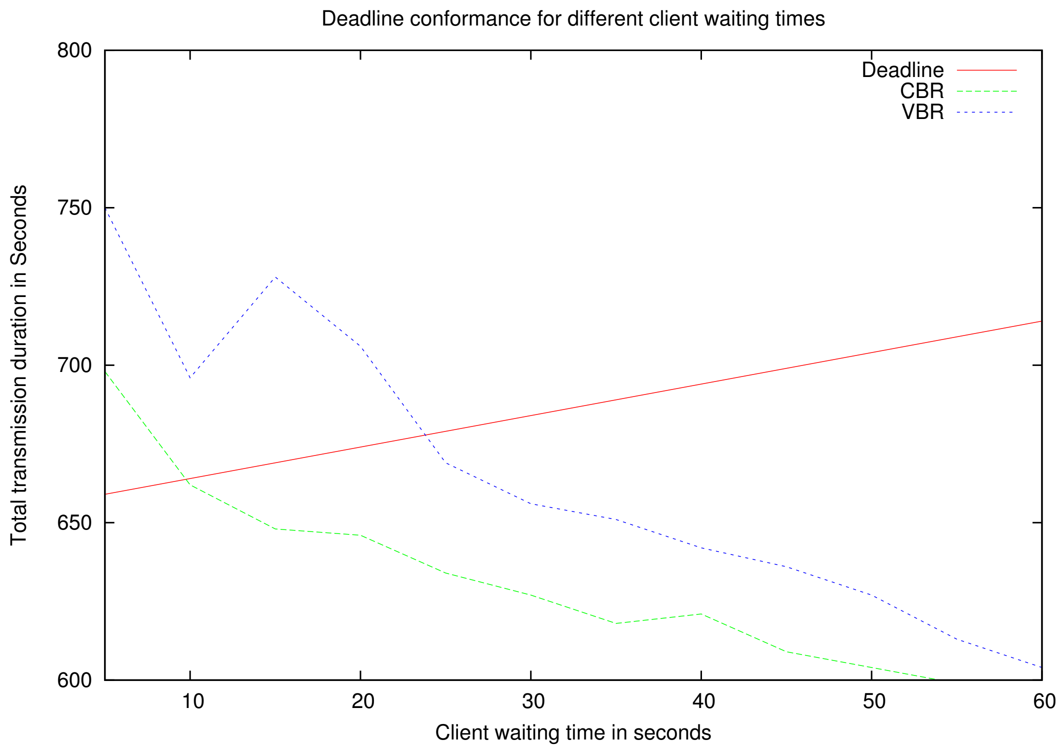

Figure 7.5: Deadline conformance for varying client waiting time

Results

In figure 7.5 we see the calculated deadline, for different client waiting times. We observe that the deadline times increase linearly as the client waiting time increases.

Further we observe that for lower client waiting times, both the CBR and VBR media fails to stay below the deadline requirement. We also observe that the CBR media performs better then the VBR media. And these observations verifies the observations in the previous section.

The CBR media is able to deliver the entire media on-time for client waiting time values greater than 10 seconds, while the VBR media does not meet the on-time requirement before the client waiting time is increased to 25 seconds. These results leads to our restatement, in that further research is necessary in order to achieve a better transmission scheduling mechanism, for small segments, and more so for small VBR based segments.

We now move on to examining the individual segment progressions.

7.2.4 Media transmission

Description

Our implementation of the CHB stream scheduling algorithm is responsible for delivering segments to its client on-time before consumption. In this section we want to examine the segment progression, and on-time delivery. The experiment is performed with both a CBR and a VBR based media, and we are interested in examining the transmission progress for these two media types, looking at the segments over time.

The experiment is performed for one iteration of the media, where the client receives the segments from the beginning of the iteration. The media is 654 seconds, and we have a 60 second client waiting time. The media is split into 10 segments of equal duration, and streamed over 9 channels.

The measurements are performed using tcpdump for traffic capture, and python scripts for data processing and plot generation.

Results

Figure 7.6 shows the segment progression for the CBR media, where the segment progress is plotted as time progresses. The streams in the figure represents the 9 CHB channels, where stream 1 is responsible for segments 1, stream 2 is responsible for segments 2 and 3. The segment progress over time takes the form of a fan, and as shown in figure 7.6, the fan shape is very closely matched. This makes it clear that the media is a constant bit-rate media, because the progress is growing evenly as time progresses.

The angle of the line representing a stream, represents the stream speed, the higher the speed is, the steeper is the plotted line. As is seen by examining stream 1, reaching 100 % after approximately 60 seconds, while stream 9 reaching 100 % after approximately 600 seconds.

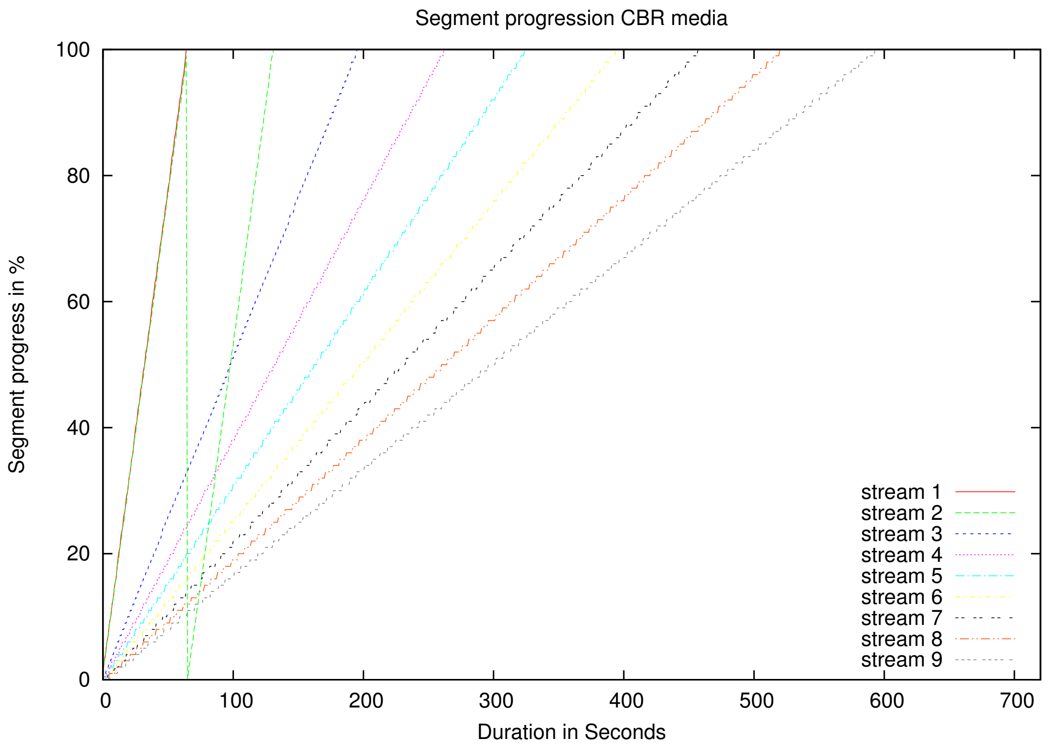

Figure 7.6: Segment progression for CBR media

The first three segments of the media are transmitted at the media playout rate, this is clear when examining stream 1 and 2, which is responsible for streaming these segments. We observe that stream 1 and 2 follow the same progression. Stream 2 however, is also responsible for transmitting segment 3 in addition to segment 2, this is done in sequence. We observe a drop back to 0 % for stream 2, as soon as it has completed the transmission of segment 2. After the drop, we observe that stream 2 again progresses from start to finish, this is the transmission of segment 3.

In figure 7.7 we see the same plot for the VBR media. Here we observer more variation in the progression lines, as is natural due to the varying bit-rate of the media. In the progress lines for stream 7 and 8, we observer that up to 400 seconds into the duration, the progress for stream 8 is ahead of stream 7, but in near the end the streams arrive in order. This variation is a clear consequence of the varying bit-rate, and is to be expected.

The scheduling still ensures that the segments for both CBR and VBR media arrive in order and on-time for consumption. In the next section we will examine the bandwidth consumption of our implementation.

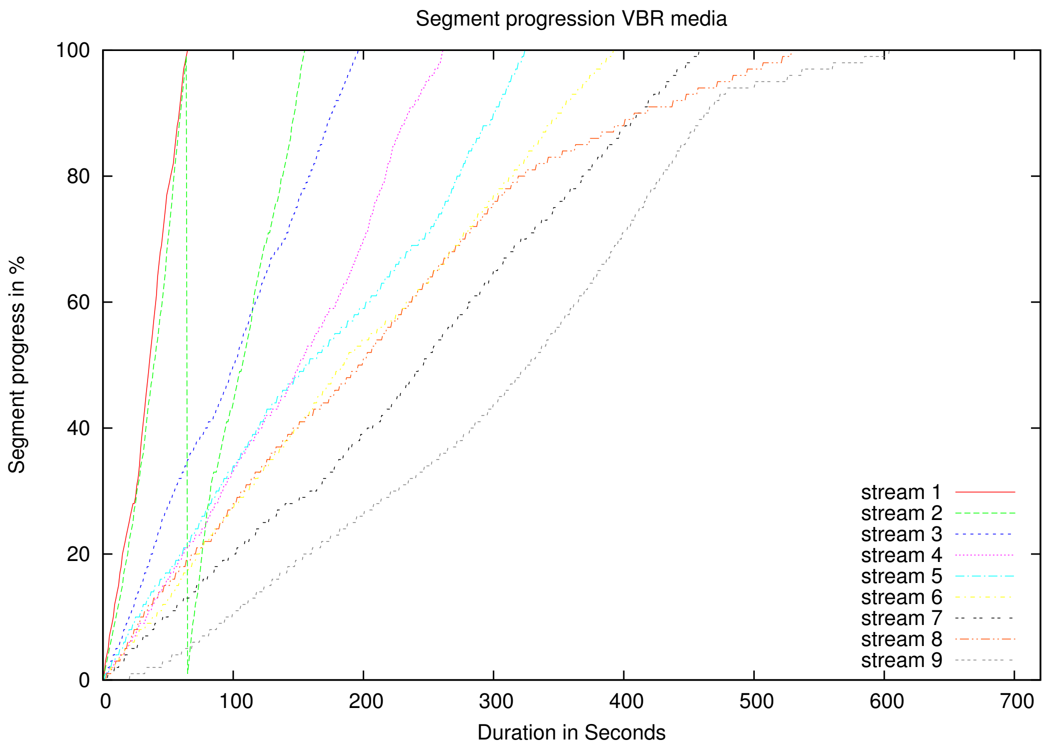

Figure 7.7: Segment progression for VBR media

7.3 Bandwidth consumption

Description

We are interested in measuring the bandwidth consumption required for our CHB implementation, and to compare these to the algorithm theory. For this experiment we use our 1400 kbps media files, and measure the average bandwidth used by each segment transfer.

We perform our measurements for both the CBR and the VBR media files. The measurement is performed by the measurements code built into our client implementation. The client measures the average bandwidth consumed by each of the 10 segments individually.

Results

In table 7.4 we see the theoretical bandwidth calculations, according to the CHB algorithm, along with the measured results. The results are shown for each segment, and the difference between the measured and calculated values are also provided. At the bottom of the table the total average bandwidth consumption is shown.

We observe that table 7.4 indicates a very close relation between the theoretical and

Average bandwidth consumption CBR			
Segment	Calculated avg. kbps	Measured avg. kbps	Difference
1	1400	1405	5
2	1400	1420	20
3	1400	1412	12
4	467	471	4
5	350	351	1
6	280	281	1
7	233	234	1
8	200	201	1
9	175	175	0
10	156	156	0
Total bandwidth for all segments			
Sum:	6061	6106	45

Table 7.4: Average and total bandwidth consumption for 10 segments CBR media

measured results, an the total difference in bandwidth consumption is 45 kbps. We observe that the measured average bandwidth is slightly higher than the calculated average bandwidth, but the differences are not alarming.

In table 7.5 we see the same measurements performed for the VBR media, while the theoretical calculations remain the same. In the VBR measurements we observe a significantly higher difference in the total bandwidth, compared to the CBR measurements. We also observe that the variations are more significant, where segments vary both above and below the calculated average. The average bandwidth consumed by segment 9 is lower than that of segment 10, and is contrary to the bandwidth reduction steps defined by CHB.

The VBR media, is performing to our expectations, as it is variable and scheduled using an adaptive scheduling mechanism. The most interesting observation is that the total average bandwidth does not deviate to much from the theoretical calculations. Next we will briefly comment on the issue of video quality.

7.4 Video quality

In our work we have not employed any particular techniques designed for analyzing video quality at the receiver. Our focus has been on testing our stream scheduling framework mechanisms and CHB implementation, and the operations and correctness for these parts. The tests performed in order to ensure media integrity, has been using a binary diff tool, that compares the source and resulting media file. This method of verifying the received and reassembled media file to the source is strict, as it requires the files to be binary equal and has been useful for detecting segment reassembly errors in the early implementation phase.

Average bandwidth consumption VBR			
Segment	Calculated avg. kbps	Measured avg. kbps	Difference
1	1400	1428	28
2	1400	1792	392
3	1400	1305	-95
4	467	374	-93
5	350	307	-43
6	280	265	-15
7	233	263	30
8	200	248	48
9	175	126	-49
10	156	167	11
Total bandwidth for all segments			
Sum:	6061	6275	214

Table 7.5: Average and total bandwidth consumption for 10 segments VBR media

7.5 Summary

In this chapter we have performed a series of experiments of our stream scheduling framework through our CHB implementation. We have demonstrated a working prototype implementation of the CHB algorithm, and determined that the performance is sufficient for both CBR and VBR based media.

We have examined the scheduling performance by examining the startup delay, transmission scheduling, on-time delivery conformance, segment progression, as well as the CHB bandwidth consumption. We find that the scheduling performance is not sufficient for small segments based on variable bit-rate. And suggest further research into the estimation and transmission scheduling mechanisms, where one possibility would be adding adaptive heuristic constants. The scheduling performance is not able to deliver all the data on time for low values of the client waiting time, but for values beyond 25 seconds, the on-time delivery succeeds. This property is affected by the media form, and CBR based media has shown a better deadline conformance than VBR based media.

In measuring the total average bandwidth consumption in our CHB implementation, we found that is closely matched that of the theoretical model in the CHB proposal. The measured values for CBR based media was closer to the theoretical model than the VBR based media.

Chapter 8

Conclusion

In this thesis we have argued that the current unicast delivery schemes for video on-demand are unable to scale when the number of consumers are rising. Valuable network resources are not being used in an optimal way, where all the consumers watching the same movie, all use exclusive video streams transmitting the same content.

A multicast infrastructure is not available in todays global Internet, but as demand for more efficient network utilization keeps rising, in parallel with growing use of multimedia content and a growing number of users, internet service providers are gaining incentives to implement multicast support in their autonomous systems. This multicast infrastructure within an ISPs domain, is required for establishing a scalable multicast based video on-demand service, and we also argue that this is the most realistic short term scenario for the deployment of a scalable video on-demand service.

We have examined different stream scheduling techniques, and chosen to focus on the periodic broadcasting class of solutions, due to its scalability, as well as predictable resource consumption, regardless of the number of consumers. The most promising algorithm, considering its demands for server bandwidth, client waiting time, client access bandwidth, client buffer requirements, as well as complexity, is the Harmonic variant named cautious harmonic broadcasting (CHB). The CHB algorithm have been criticized for its high requirement for channels, translating into a high number of IP multicast addresses, but our scope of applicability is running this algorithm inside of an internet service providers autonomous system, and thus will not be unmanageable.

Based on these considerations we have implemented and measured an IP multicast based periodic broadcasting stream scheduling technique, and will now go on to summarize our main contributions, followed by our suggestions for future work in this context.

8.1 Summary and contribution

The main contributions of this thesis is the implementation of the cautious harmonic broadcasting algorithm, as well as a stream scheduling framework (SSF), that supports implementing periodic broadcasting class algorithms. By implementing a SSF we support implementing other promising periodic broadcasting algorithms in our future work, in a more consistent approach. The client side SSF implementation is algorithm agnostic, as long as the server side algorithm can be expressed through our schedule description approach. The client is served descriptions of each segment, along with descriptions of each channel. For each of the channels we attach a channel schedule stating how the client will receive and assemble the segments, this loose coupling between segments and channels is the rationale for claiming an algorithm agnostic client.

The SSF is built by extending the live555 streaming media library, enabling future integration into existing media-players like VLC and MPlayer. The live555 library did not support the use of RTP extension headers, and such support have accordingly been implemented, to ensure the support for more complex scheduling, like shared channels.

The implementation have been tested through running experiments, and measuring the performance and operation of the CHB algorithm, using both constant bit-rate and variable bit-rate media. We also implemented a specialized version of the CHB algorithm, where all the segments were equal and consisted of the entire media. This special implementation was used to measure the correctness of our transmission scheduling in a comparative manner, and helped us realize the issues in scheduling variable bit-rate media using the same approach as scheduling the constant bit-rate media.

We conclude that the implementation of periodic broadcasting algorithms are realistic and feasible, and that the CHB algorithm performs well in most of our experiments. However we also suggest that more research is necessary in the area of transmission scheduling calculations, and especially for small media segments, and even more so for the ones based on a variable bit-rate media. We therefore move on to our suggestions for future work.

8.2 Future Work

In our measurements of our CHB implementation, we have seen that we are unable to guarantee on-time delivery when the segments transmitted are small and have a short duration. The mechanism responsible for transmission scheduling in the live555 library, is adaptive and uses the transmission history for a particular program inside the transport stream, as well as the current TS packet, when it estimates the transmission time. We believe that for perfect CBR media, the transmission scheduling mechanism works as intended, and is able to deliver media on-time. When the media bit-rate is variable however, the current transmission scheduling technique is not able to calculate and schedule the transmission times correctly for small segments. For this reason we suggest that this mechanism should be the subject of further research, in order to support on-time delivery of variable bit-rate media segments of any size.

Another subject we suggest for further research are the mechanisms causing the linearly growing startup delay. We believe that the startup delays will remain in any sequential processing system, but the time between the different streams initial packet, could be mitigated by preloading the first part of each segment into memory buffers, avoiding time consuming disk IO operations to affect the startup delays.

Finally we would like to see other periodic broadcasting algorithms implemented using our SSF, and perform similar experiments and measurements on these algorithms. After gaining insight into the performance of different periodic broadcasting techniques, we suggest moving on to integrate the SSF client into and existing media-player like VLC or MPlayer.

Bibliography

[1] Carsten Griwodz and Pål Halvorsen. Distribution systems (lecture notes). *http://www.uio.no/studier/emner/matnat/ifi/INF5071/h08/undervisningsmateriale/06-distribution-1.pdf*, October 2008. [Online; Accessed 06-March-2009].

[2] J. F. Paris, S. W. Carter, and D. D. E. Long. Efficient broadcasting protocols for video on demand. In *Proc. Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 127–132, 1998.

[3] Ross Finlayson. Live555 streaming media. *http://www.live555.com*, March 2009. [Online; Accessed 06-March-2009].

[4] Videolan Project. Videolan media player. *http://www.videolan.org*, March 2009. [Online; Accessed 06-March-2009].

[5] MPlayer Team. Mplayer. *http://www.mplayerhq.hu*, February 2009. [Online; Accessed 06-March-2009].

[6] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *MULTIMEDIA '94: Proceedings of the second ACM international conference on Multimedia*, pages 15–23, New York, NY, USA, 1994. ACM.

[7] Leana Golubchik, John C. S. Lui, and Richard Muntz. Reducing i/o demand in video-on-demand storage servers. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 1995. ACM.

[8] 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 Syst.*, 4(3):140–155, 1996.

[9] Charu Aggarwal, Joel Wolf, and Philip S. Yu. On optimal piggyback merging policies for video-on-demand systems. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 200–209, New York, NY, USA, 1996. ACM.

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

[11] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid. *Multimedia Systems*, Volume 4(4):197 – 208, 08 1996.

[12] Kien A. Hua and Simon Sheu. Skyscraper broadcasting: a new broadcasting scheme for metropolitan video-on-demand systems. *SIGCOMM Comput. Commun. Rev.*, 27(4):89–100, 1997.

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

[14] Kien A. Hua, Ying Cai, and Simon Sheu. Patching: a multicast technique for true video-on-demand services. In *MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia*, pages 191–200, New York, NY, USA, 1998. ACM.

[15] S. W. Carter and D. D. E. Long. Improving video-on-demand server efficiency through stream tapping. In *Proc. Sixth International Conference on Computer Communications and Networks*, pages 200–207, 1997.

[16] D. Eager, M. Vernon, and J. Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. *IEEE Transactions on Knowledge and Data Engineering*, 13(5):742–757, 2001.

[17] Cyrus Dara Vesuna and Jehan-Francois Paris. An empirical study of harmonic broadcasting protocols. *Computación y Sistemas [1405-5546]*, 7(3):156–166, 2004.

[18] Tonje Jystad Fredrikson. Scheduling of data streams over a multicast protocol. Master's thesis, University of Oslo, December 2008.

[19] Rtp - rfc3550, 2003.

[20] Handley, Jacobson, and Perkins. Sdp: Session description protocol, July 2006.

[21] Cain, Deering, Kouvelas, Fenner, and Thyagarajan. Internet group management protocol, version 3. rfc 3376., October 2002.

[22] Fenner, Handley, Holbrook, and Kouvelas. Protocol independent multicast - sparse mode (pim-sm). rfc4601., August 2006.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : elements of reusable object-oriented software*. Addison-Wesley, 1994.

[24] Project Orange Blender Foundation. Elepants dream. *http://www.elephantsdream.org/*, October 2008. [Online; Accessed 06-March-2009].

[25] The-FFmpeg-Project. Ffmpeg documentation. *http://www.ffmpeg.org/ffmpeg-doc.html*, March 2009. [Online; Accessed 19-March-2009].

[26] The XORP team. Xorp. *http://www.xorp.org*, January 2009. [Online; Accessed 06-March-2009].