Potential of Quality-Adaptive Streaming of Layer-Encoded Video over a Wireless Network

by Cuong Huu Truong

Master Thesis

The Department of Informatics,
University of Oslo

November 2005

Table of Contents

Acknowledgments

I wish to express a sincere gratitude to my advisor Carsten Griwodz and my co-advisor Svetlana Boudko for their support and patience all the way through the process. During the research of this thesis they provided me with many good advices and essential information, which helped me a lot in achieving the goal of this thesis.

Also I would like to thank the Norwegian Computing Center (Norsk Regnesentral) for providing me with the necessary equipments and a space in their laboratory for doing my research.

I would also like to thank my family and friends for their encouragement and support.

Cuong Huu Truong
Oslo, November 2005

Abstract

The aim of this thesis is to find and implement an optimal solution for efficient utilization of the varying bandwidth of a wireless network when streaming a video, between a streaming server and a mobile wireless device. The video used in the research is in a layer-encoded format, which makes it possible to achieve quality-adaptive streaming through priority dropping of video layers. The research is based on an existing streaming software called Qstream. This software supports quality-adaptive streaming, by making use of the layer-encoded video format SPEG for streaming and an algorithm known as Priority-Progress Streaming (PPS) for priority data dropping. This thesis contributes by providing an improvement code, which is added to this software to make the quality-adaptation work in a network with intensely varying bandwidth. The challenge also lies in the task of verifying how efficient the improved system is when handling different degrees of bandwidth variations

List of Figures

Chapter 1: Introduction

Through the years, video and audio have rapidly become an essential part of the Internet. As the interest for instant access to continuous media was growing, technology was improved to overcome the fact that multimedia files have to be fully downloaded before viewing is possible. Streaming was developed to support this instant access feature.
However, achieving full utilization of streaming is not an easy task due to a number of technical problems. Network resources are probably the one of primary concern, as streaming in a satisfying way mostly depends on a good network bandwidth, which costs are expensive and slow to improve for wide areas. The following two sections, 1.1 and 1.2, provide a deeper insight into the meaning of streaming and the technical problems related to it. Section 1.3 gives an overview of the goal of this thesis. Finally, section 1.4 provides a brief description of the structure of this thesis.

1.1 Streaming

Streaming is a term used to describe the process of transmitting multimedia data from a sender to a receiver over the Internet, or other kinds of network, for instant viewing.
This is a precondition for live communication (telephony, video conferencing) and will improve user satisfaction in on-demand services. It is also favourable in the case of playing back multimedia files stored on remote machines.

The benefit is that a receiver doesn't need to have access to the entire multimedia content before the playback can begin. The content will usually be played as soon as it arrives at the receiver. This is the reason why live viewing is possible, as streaming allows playback to occur in real-time. Satisfaction in using streaming depends on continuous playout. Once the playback of the multimedia content begins, it should proceed according to the original timing of the content. The receiver must get data from the sender in time, or else jitter in playback will occur.

In some streaming cases the receiver may have a buffer to store some future data. It is meant as a precaution against network problems that might result in data not arriving in time. Buffering is possible because transmission speed and playback speed are to a very large degree independent, which is explained in section 1.3. The receiver will be able to play the video/audio while it receives and buffer the later parts concurrently. This is typical when streaming a pre-recorded multimedia file from a remote sender.

However, this is not always the case due to higher real-time demand in certain kinds of streaming. In a live communication for instance, it is not possible to buffer future data as these data might not have been captured yet. With network problems present, the receiver will likely have to accept jitter in this kind of presentation.

Streaming supports the three kinds of network traffic, unicast-, broadcast- and multicast-traffic. In a unicast streaming, the multimedia content is transmitted separately from the sender to all the receivers that request it. When broadcasting, a single copy of the multimedia content is sent to all receivers on the network. In general, both these methods waste bandwidth when the same data needs to be sent to only a portion of the receivers on the network. Unicast wastes bandwidth by sending multiple copies, while broadcast wastes bandwidth by sending to the whole network whether or not the multimedia content is wanted. Multicast was introduced to solve this problem as its strategy is to send a single copy of the data only to those receivers who request it.

1.2 Problems Related to Streaming

The problems that might arise when streaming are caused by limitations of the fundamental resources: processors, storage and network. These limitations may affect the performance of the streaming in the sense that continuous viewing is interrupted by errors like delay and jitter. The most crucial resource today, in regard to such failures, is perhaps the network, where the available network bandwidth plays a significant role.

Since wide-area bandwidth costs are expensive and slow to improve, the primary challenge is to find ways to deliver video in a most efficient manner at low bandwidth costs. To reduce bandwidth costs, video compression and video distribution techniques have been developed.

Video compression is basically a technique to reduce the size of a video file, but still maintain a good and acceptable video quality in comparing to the original one. Thus, the compressed video file requires less bandwidth to transmit. Different compression formats have been developed through the years. Among them is one named Moving Picture Experts Group (MPEG), which is of interest to this thesis and is elaborated in section 2.3.1 of chapter 2. Most compression formats also have the ability to carry video with variable data rate.

On the distribution side, improvements in speed and cost have been made to basic networking technologies such as link types, switchers, routers, etc. Techniques like caching and multicasting have also been taken into account to achieve efficient distribution of video content.

Apart from the transmission cost problem, there is another issue for streaming that needs to be resolved. That is the handling of variable video and network rates. The main purpose of streaming is to deliver video across the network with proper timing, so that it is displayed at the receiver at the proper rate and without interruption. To be able to do this, the sender application is required to transmit the video in a most efficient way, with the variable video and network rates taken into consideration. Quality-adaptive approaches to streaming have been developed to solve this problem. As compression controls the rate of the video, these approaches have the task of adjusting the compression ratio of the video adaptively, so that timeliness of video playout is

maintained. According to how the network bandwidth is, the quality-adaptive approach will attempt to match the rate of the video to the rate of the network to achieve as efficient streaming as possible.

1.3 The Goal of this Thesis

The goal of this thesis is to determine how to deliver video to a mobile device, through a wireless network with varying bandwidth, in a most efficient manner. It should be noted that the wireless network is simulated in this thesis, but the idea is to develop a code that will also work in a real wireless network with varying bandwidth.

A quality-adaptive streaming approach is required for the research as well as a scalable layer-encoded video format that is rate-adjustable. The quality-adaptive streaming algorithm Priority-Progress Streaming (PPS) and the scalable video format SPEG (scalable version of MPEG) are the two chosen candidates for this thesis. These are further explained in sections 2.4 and 2.3 of chapter 2, respectively.

This thesis is based on unicast streaming over the Internet protocol Transmission Control Protocol (TCP), and the multimedia content used for streaming is a pre-recorded SPEG file stored on the sender machine. The issues about streaming over TCP are covered in section 2.1.2 of chapter 2.

Qstream is a software that includes a quality-adaptive streaming system. It makes use of the PPS algorithm and is specifically developed for streaming over TCP. Improvement of this software is the primary interest of the research, as the goal is to make the quality of the streaming video adapt gracefully to the varying bandwidth condition caused by a wireless network. The details about Qstream is further elaborated in sections 2.6 of chapter 2 and 3.1 of chapter 3.

The wireless network is simulated by using a network emulator, and thus different network bandwidth scenarios can be created to be used in tests, to verify the efficiency of the quality-adaptive streaming in different circumstances. The wireless network simulation is explained in section 2.2.2 of chapter 2.

A scalable video stream indicates a stream that can be divided into several video layers. The details are covered in section 2.3.2 of chapter 2. Thus, it's possible to adjust the amount of video layers that the streaming server is allowed to transmit, according to how the network condition is. Qstream has the ability to divide the video stream into smaller time intervals called adaptation windows, which is described in section 2.4 of chapter 2. Each window's amount of layers, which defines the video quality, is independent of the other windows. Therefore, an adaptation window that consists of a small amount of video layers will result in a video with lower data rate, within the time interval that is covered by the window, compared to another adaptation window with more layers. It follows that the transmission speed of the video stream can differ from the playback

speed, as the transmission time of the adaptation windows depends on how many video layers the windows contain, and also on the condition of the network.

A concrete example will explain this more clearly. Assume that the maximum layers a video stream can be divided into is L, and among all the adaptation windows to be transmitted, there are two in the spotlight called A and B. Thus, each of the window contains L video layers that are possible to transmit.

If the quality-adaptation algorithm decides that L_1 layers of adaptation window A and L_2 layers of adaptation window B are to be transmitted, then the transmission time for adaptation window A is shorter than for adaptation window B if $L_1 < L$, $L_1 < L_2 \leq L$, and the network bandwidth is equally good when transmitting the two adaptation windows.

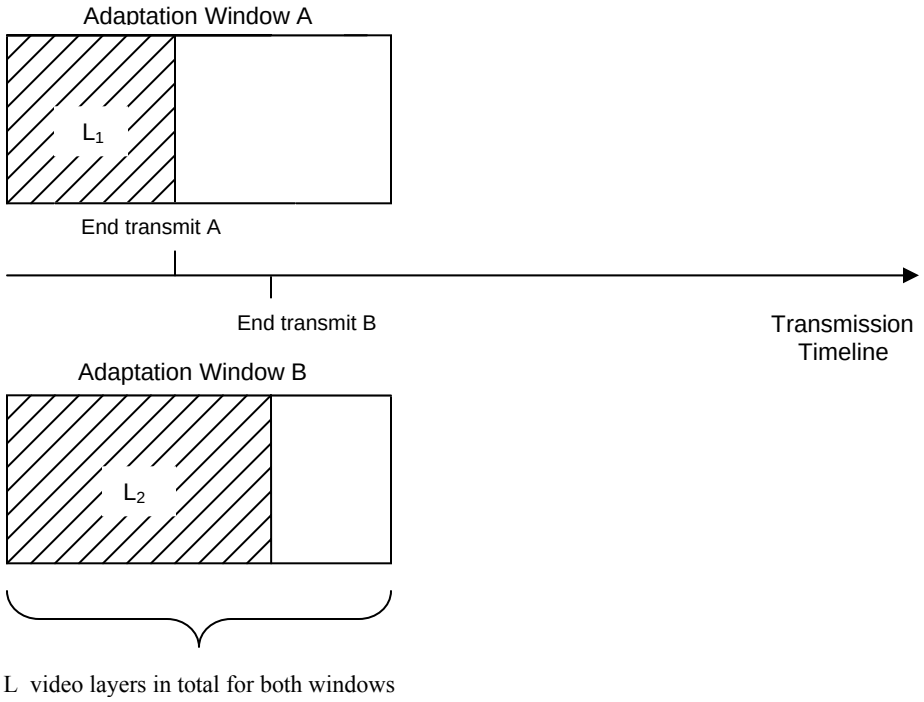


Figure 1: Independent Transmission Time

Apparently this indicates that by decreasing the number of layers to be transmitted for a number of adaptation windows, the transmission time might get ahead of the playback time. The time difference is called the workahead time. This is the same as to say that the receiver is buffering some future data. Since the transmission is ahead of time, the receiver gets data that is not supposed to be displayed yet. These data are stored in a temporary buffer until the time for decode and display arrives.

The buffering of future data is an important assumption for handling a streaming session over a wireless network with varying bandwidth. In a network with unpredictable

bandwidth, the streaming system must be able to adapt to the condition of the network at all time, by estimating the balance point between workahead/buffering and video quality. This means that less layers for each adaptation window leads to reduced video quality, but more workahead/buffering. Buffering provides support in the sense that the playout doesn't stop immediately if the connectivity suddenly is gone. How long the playout can keep going, depends on how much video left there is in the buffer. On the other hand, more layers for the adaptation windows give better video quality, but at the expense of workahead/buffering. This trade-off theory is elaborated in section 3.2.3 of chapter 3.

To summarize, streaming over a unreliable wireless network with varying bandwidth needs to be controlled and quality-adapted. The main goal of this thesis is to develop an algorithm on the server side that makes the streaming server aware of the condition of the network at all time throughout a streaming session. Based on this awareness, the algorithm makes it possible for the streaming server to adapt the quality of the video stream according to the network condition. The adaptation is based on the issues discussed above, which consists of estimating the amount of video layers to transmit for each adaptation window, and the amount of workahead time to acquire at different times.

1.4 Thesis Structure

Chapter 2 provides a detailed description of the relevant background materials that this thesis is based on. The focus is put on the materials that are mentioned in the previous section.

In chapter 3, a detailed analysis is performed on the Qstream software to reveal the areas that need to be improved in order to achieve the goal of this thesis. A proposal of an improvement code is also introduced and discussed.

Chapter 4 provides an insight into the way the improvement code is implemented, which is based on the proposal made in chapter 3. The improvement code uses the C programming language, since Qstream is based on this language. However, to simplify and make it easier to understand, the code presented in this chapter is written in pseudo-code.

In chapter 5, the improvement code is tested and evaluated. The test part is based on a number of test cases (streaming sessions) that are performed on the code. The goal is to investigate if the improvement code solves the issues introduced in this thesis efficiently enough.

Chapter 6 is the final chapter of this thesis, which consists of a conclusion and ideas about future work.

Chapter 2: Background Materials

This chapter provides an introduction to the background materials that are relevant for this thesis, which is based on a number of sources ([1], [2], [3], [5], [6], [7], [8], [9] [10], [11], [12], [13]). Section 2.1 describes the streaming technique in practice. Section 2.2 and 2.3 cover the issues of wireless network and scalable video format, respectively. Section 2.4 and 2.5 introduce two algorithms, Priority-Progress Streaming (PPS) and Priority Mapping, which are fundamental for achieving quality-adaptive multimedia streaming. The last section of this chapter gives an overview of the Qstream software that is used and further developed in this thesis.

2.1 Streaming in Practice

Today there are several internet protocols available for streaming data, TCP, UDP, RTP, MMS and HTTP.

User Datagram Protocol (UDP) is probably the most preferable protocol for streaming, and the following section will provide further details of this protocol.

Transmission Control Protocol (TCP) has been considered less suitable for streaming, but in recent years there have been arguments against this claim [18]. Streaming softwares were also developed to prove that TCP might not be as bad as it's claimed to be. One example is the Qstream software which is used in this thesis. As this thesis is based on TCP streaming, the issues about TCP streaming are further elaborated in section 2.1.2.

Microsoft introduced Microsoft Media Server (MMS) as the primary server protocol of their media technologies. MMS includes both Microsoft Media Server protocol/UDP (MMSU) and Microsoft Media Server protocol/TCP (MMST) as subsets to explicitly request the stream to use UDP or TCP respectively. This protocol has both a data delivery mechanism to ensure that packets reach the receiver and a control mechanism to handle client requests such as Play/Stop.

Hyper Text Transport Protocol (HTTP) is the slowest of the protocols and is used by Internet Web Servers. HTTP is a well known protocol used everyday by people who browse the Internet. This protocol has the ability to simulate streaming by using a method called progressive download, and it is great for short contents. As the multimedia content is in downloading progress, the receiver computer will start playing the video/audio while it keeps downloading it concurrently. This will make it look like a real streaming, but in reality it's just a normal downloading process. The receiver must support this feature, or else the simulated streaming will not work.

2.1.1 Streaming with UDP

User Datagram Protocol (UDP) provides a way for applications to send encapsulated IP datagrams. The transmit is possible without having to establish a connection, and UDP is therefore defined as a connectionless protocol. UDP transmits packets which consist of an 8-byte header followed by a payload. The header contains the source and destination ports, which helps the transport layer to deliver the packet to the right destination. UDP does not support flow control, error control (FEC, etc.) or retransmission upon receipt of a bad segment. All of that is up to the user processes. However, retransmission is generally considered bad for streaming because it adds latency at the application layer. So the fact that UDP is missing this feature, has been one of the reasons why it's favourable for streaming. A protocol widely used for streaming which runs on top of UDP, is the real-time protocol (RTP). Before sending a file into the network for streaming, it has to be split into smaller packets. The packets are typically encapsulated with special header fields that include sequence numbers and timestamps. Usually RTP is chosed to serve this purpose.

RTP, defined in RFC 3550, is a standard used for transporting common formats such as PCM, GSM, MP3 for sound and MPEG and H.263 for video. What the sender side actually does is that it encapsulates a media segment within an RTP packet. The media segment along with the RTP header form the RTP packet. This packet is further encapsulated in a UDP segment which will be handed to IP (Network layer). The receiving side extracts the RTP packet from the UDP segment. From the RTP packet it will extract the media segment and use the header fields to properly decode and play back the segment with a media player. However, RTP does not provide any mechanism to ensure timely delivery of data or provide other quality-of-service (QoS) guarantees for the client application, and the delivery of packets to the application can also be out-of-order.

The RTP header consists of the following important header fields:

- **Payload type** - This field is 7 bits long. For an audio stream, the field is used to indicate the type of audio encoding that is being used, for example PCM, adaptive delta modulation, linear predictive encoding, etc. For a video stream, the field indicates the type of video encoding, for example JPEG, MPEG-1, MPEG-2, H.261, etc. The space for payload types is limited, so only very common video and audio encodings are assigned static (permanent) types, such as those described above. On the other hand, dynamic payload types are not assigned in the RTP profile. They are dynamically assigned, and the meaning is carried by external means. They map an RTP payload type to an audio and video encoding for the duration of a session. Different members of a session could, but usually not, use different mappings. Dynamic payload types use the range 96 to 127 while static payload types use range below 90.

- **Sequence number** - This field is 16 bits long. The sequence number increments by one for each RTP packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence.

- **Timestamp** - This field is 32 bits long. The receiver can use timestamps in order to remove packet jitter introduced in the network and to provide synchronous playout of the media file. This timestamp is derived from a sampling clock at the sender.

- **Synchronization source identifier (SSRC)** - This field is 32 bits long. SSRC is a number used to identify which packets that belong to the same RTP stream, and this number is randomly assigned by the source when the new stream is started.

In addition to RTP there are also two other protocols defined, the RTP Control Protocol (RTCP) and the real-time streaming protocol (RTSP).

As the name indicates, RTCP packets are control packets. These are sent periodically and contain sender and/or receiver reports that announce statistics useful to the application. A sender generates a sender report for each RTP stream that it is transmitting, while a receiver generates a reception report for each RTP stream that it receives. The sender/receiver aggregates its report into a single RTCP packet, and this packet is sent into the multicast tree that connects all the session's participants.

The real-time streaming protocol (RTSP) is a signalling protocol. The following control actions are possible, pause/resume, fast-forward, rewind and repositioning of playback. The protocol is based on a set of request and response messages between the client and the server. It is similar to the HTTP protocol where all request and response messages are in ASCII text. The client employs standardized methods (SETUP, PLAY, PAUSE, etc.), and the server responds with standardized reply codes. The following example shows a client (C) requesting for playback of an audio file by sending a "PLAY"-RTSP message, and the server (S) that responds with an "OK"-RTSP message:

C: PLAY rtsp://audio.example.com/twister/audio.en/lofi RTSP/1.0
Range: npt=0-
Cseq: 2
Session: 4231

S: RTSP/1.0 200 OK
Cseq: 2
Session: 4231

2.1.2 Streaming with TCP

Transmission Control Protocol (TCP) was considered unsuitable for streaming due to its two basic mechanisms, packet retransmissions and congestion control [18] .

TCP was designed to provide a reliable end-to-end byte stream over an unrealiable internetwork. An internetwork may have different topologies, bandwidths, delays, packet sizes, etc. This led to the design of TCP which could dynamically adapt to the conditions of the internetwork and to handle various kinds of failures.

TCP is a connection-oriented protocol which provides a connection-oriented service. For TCP service to be obtained, a connection must be explicitly established between instances of TCP on the sending machine and the receiving machine. Each machine supporting this protocol has a TCP transport entity that manages TCP streams and interfaces to the IP layer. This entity will accept user data streams from local processes and break them up into pieces not exceeding 64 KB. Each piece will be sent as a separate IP datagram. When the datagram arrives at the receiving machine, it is given to the TCP entity which is responsible for reconstructing the original byte stream from the received datagrams.

The IP layer does not give any guarantee that datagrams will be delivered properly, so it is up to TCP to retransmit them when necessary. When TCP sends out data, it requires an acknowledgment (ACK) from the receiver in return. If the acknowledgment arrives several times or doesn't arrive at all, TCP must retransmit the data.

TCP has an implemented function known as the fast retransmit algorithm which deals with retransmission. On the sender side TCP will count the ACKs for a sent datagram. If the ACK for datagram N is received three times at the sender, it will assume that the sent datagram N+1 is lost. The sender will then retransmit datagram N+1. Also it is the reponsibility of TCP to reassemble datagrams into messages in the proper sequence if the datagrams arrive in wrong order.

In a streaming situation this is considered unacceptable, since it will introduce end-to-end latency. The claim is that re-sending is not appropriate in regard to the real-time nature of video, because the resent data would arrive too late at the receiver for display [18] . Another problem that might occur with retransmission is its potential to limit the effectiveness of end-to-end feedback mechanisms.

The other drawback with TCP streaming is the congestion control mechanism. Congestion occurs when a network is offered more data than it can handle. A possible way to solve this problem is to refrain from injecting a new data packet into the network until an old one is delivered. TCP achieves this goal by dynamically manipulating the window size of a congestion window. This window will help exploring how much traffic the network can handle, and it will react upon a congestion. When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use on the connection. After sending one maximum segment, hopefully an

acknowledgment will come back. If the acknowledgment arrives before timeout, the sender will add one segment's worth of bytes to the congestion window to make it two maximum size segments and sends two segments. When a congestion window is n segments and all n are acknowledged, the window is increased by the byte count corresponding to n segments.

All TCP implementations support an algorithm called slow start. The idea of the algorithm is that if bursts of packets of size like 1024, 2048 or 4096 bytes work fine but a burst of 8192 bytes gives a timeout, then the congestion window should be set to 4096 to avoid congestion. The point is that whenever a congestion occurs, the congestion window will be set to half its size. When the transmission begins, slow start will be used to determine what the network can handle. The slow start algorithm will stop when a threshold (initally 64KB) is reached, and from that point on successful transmissions grow the congestion window linearly (by one maximum segment for each burst). When a timeout occurs, the threshold will be set to half of the current congestion window. The window is reset to one maximum segment, and the transmission will continue using the slow start algorithm again until the new threshold is reached. So basically this congestion algorithm probes available bandwidth, through deliberate manipulation of the transmission rate. When viewed over shorter time-scales, the transmission rates form a sawtooth shape. This shape indicates abrupt transmission rate changes, which unfortunately might impede efficient streaming.

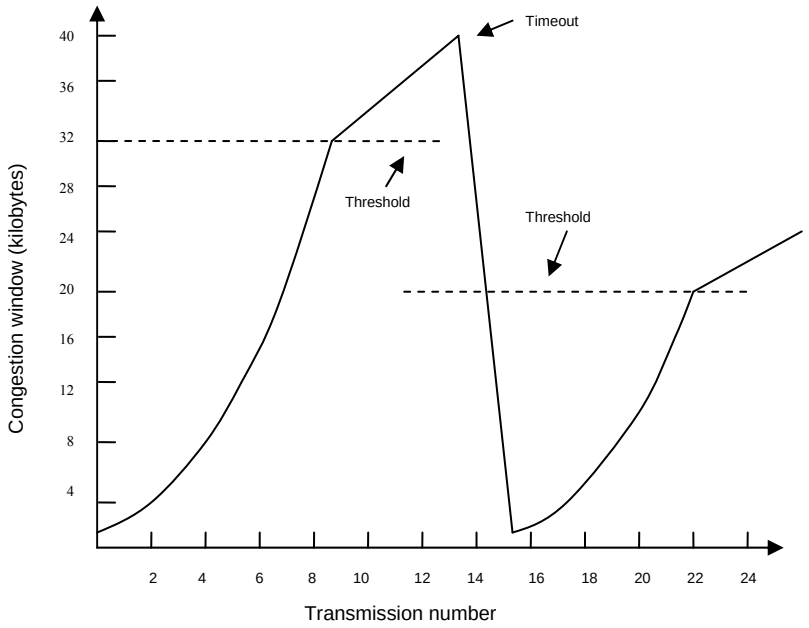


Figure 2: TCP Congestion Algorithm [8]

Figure 2 shows how the TCP congestion algorithm works. The maximum segment size in use here is 1024 bytes. Initially, the congestion window was 64 KB. But because a timeout occurred, the threshold is set to 32 KB and the congestion window to 1 KB for transmission 0 here.

A way to resolve the problems introduced by TCP, is to employ buffering at the receiver application to smooth out the rate change. Protection from sudden rate reductions will be achieved by borrowing some current bandwidth to transmit future data and buffer these at the receiver. It is also a favourable solution to use a scalable video format for this purpose, since the video can be divided into different layers. When time or bandwidth is critical, the less important layers of the video is dropped. Scalable video is explained in section 2.3.2.

2.2 Wireless Network

This section provides some background materials about the wireless network standard in use today, along with a description of how to create a simulated wireless network which is necessary for the research of this thesis.

2.2.1 IEEE 802.11 WLAN

This section provides a short introduction into the IEEE 802.11 WLAN (Wireless Local Area Network). This was the first international standard for WLANs that was adopted back in 1997.

The main difference from wired networks is that wireless networks make use of the air link instead of wires, which could either be the radio or infrared link between WLAN transmitters and receivers. The mobility provided here is an important feature and gives users the opportunity to move around freely with their laptops for instance. Since the data on a WLAN is broadcasted for everybody to hear, the IEEE 802.11 standard has provided a cryptographic mechanism in the protocol to protect the data being sent through the air.

The IEEE 802.11 architecture consists of the following components:

- **The Station (STA)** - This is the most basic component of a wireless network. It's a device that has the functionality of the 802.11 protocol and has the ability to connect to the wireless medium. It consists of a MAC (Medium Access Control) and a PHY (Physical Layer) which is explained more later. The station may be mobile, portable or stationary, and it also supports station services such as authentication, deauthentication, privacy and data delivery. A station could be a laptop PC or a handheld device, and they are usually referred to as the network adapter or network interface card (NIC).

- **The Basic Service Set (BSS)** - This is known as the basic building block of an 802.11 wireless LAN, and it's defined as a group of any number of stations. When all the stations in the BSS are mobile and not connected to a wired network, the BSS is called an independent BSS (IBSS). In an IBSS all stations can communicate directly with other stations under the condition that they are within

range of each other. When the BSS includes an access point (AP), it is no longer independent and is called an infrastructure BSS, usually referred to simply as a BSS. The difference now is that the stations do not communicate directly with each other, but go via the AP.

- **The Access Point (AP)** - The AP provides the local relay function for the BSS and the connection to a wired LAN if there is any. As told in the previous section, a station in an infrastructure BSS doesn't communicate directly with another station. Instead the communication is first sent to the AP and then forwarded from the AP to the other mobile station, aka. data being relayed between the mobile stations by the AP. One major advantage about this is that the AP can buffer data frames for mobile stations. So when these data frames are requested by another mobile station and the source station is in power saving mode, the AP can provide the station with the requested data frames from the buffer, without having to 'wake up' the source station. That way mobile stations in power saving state can remain in such condition for longer periods.

- **The Wireless Medium** - The IEEE 802.11 standard defined three physical (PHY) layers which are an infrared (IR) baseband PHY, a frequency hopping spread spectrum (FHSS) radio in the 2.4 GHz band and a direct sequence spread spectrum (DSSS) radio also in the 2.4 GHz band.

- **The Distribution System (DS)** - The DS could be defined as a mechanism by which an access point communicates with another access point to exchange data frames for stations in their BSSs, forward frames to follow mobile stations from one BSS to another, and exchange frames with a wired network. The requirements of the DS is that it must provide certain distribution services. There are no restrictions on the implementation of the DS, and it can be referred to as an abstract medium.

- **The Extended Service Set (ESS)** - An ESS is a set of infrastructure BSSs, where APs communicate among themselves to forward traffic from one BSS to another and to facilitate the movement of mobile stations from one BSS to another.

The IEEE 802.11 architecture has defined nine services which are divided into two groups, station services and distribution services.

Station services:

- **Authentication** - Provides a mechanism for a station to identify another station. Without such proof of identity, a station is not allowed to use the WLAN for data delivery.

- **Deauthentication** - This is used to eliminate a previously authorized user from any further use of the network. Once a station is de-authenticated, it can not access the WLAN without performing the authentication function again.

- **Privacy** - This mechanism is supposed to protect the data as it traverses the wireless medium. The level of security of this protection is equally good as that of a wired network. The privacy service is an encryption algorithm based on the 802.11 Wired Equivalent Privacy (WEP) algorithm.

- **Data delivery** - This provides a reliable delivery of data frames from the MAC in one station to the MAC in one or more other stations.

Distribution services:

- **Association** - A logical connection between a mobile station and an AP is required before a station can send data through the AP onto the distribution system. This is also known as an association between a mobile station and an access point.

- **Reassociation** - This enables a station to change its current association with an access point to be able to associate with a new access point. The station can provide information to the new AP, so that it can contact the previous AP to obtain frames that may be waiting there for delivery to the mobile station, or other relevant information.

- **Disassociation** - This is used to make a mobile station eliminate its association to an access point. The mobile station can also use this service to inform an access point that it no longer needs the services of WLAN. When a station becomes disassociated, it must go through the association process to be able to communicate with an access point again.

- **Distribution** - This is the primary service used by a 802.11 station. A mobile station uses the distribution service every time it sends MAC frames across the distribution system. This service provides the information to determine the proper destination BSS for the MAC frame.

- **Integration** - This service connects the IEEE 802.11 WLAN to other LANs, including one or more wired LANs, or other IEEE 802.11 WLANs. It is also capable of translating IEEE 802.11 frames to frames that may traverse another network, and vice versa.

Medium Access Control (MAC)

The 802.11 MAC layer provides the functionality to allow reliable data delivery for the upper layers over the noisy, unreliable wireless media. Another function it provides is a

fair controlled access to the shared wireless medium. A third function is to protect the data that it delivers, and the MAC layer does this by providing a privacy service that's been mentioned earlier, Wireless Equivalent Privacy (WEP). This layer also implements a frame exchange protocol to allow the source of a frame to determine when the frame has been successfully received at the destination.

Physical Layer (PHY)
This layer is the interface between the MAC and the wireless media where data frames are being transmitted and received. The PHY provides three functions. The first one is an interface with the upper MAC layer for transmission and reception of data. The second function is that the PHY uses signal carrier and spread spectrum modulation to transmit data frames over the media. And thirdly, the PHY provides a carrier sense indication back to the MAC to verify activity on the media.

2.2.2 Simulation of a Wireless Network

Since a real-world wireless network is not completely predictable, a simulated wireless network is necessary for this thesis.

The Linux network emulator Netem is used to emulate the variable bandwidth of a wireless network. This emulator supports a range of queuing disciplines, where the first-in first-out (FIFO) discipline is the one of interest. By queuing it means to manipulate the way in which data is sent. In FIFO, the data packets are placed in a single queue and are served in the same order they were placed.

In addition, the Token Bucket Filter (TBF) algorithm, which is also supported by Netem, is used to control the amount of outgoing data packets. It consists of a buffer (bucket) that is constantly filled by some virtual pieces of information called tokens, at a specific rate (token rate). Each generated token collects a certain amount of bytes from the data queue and is then deleted from the bucket. The collected bytes of data are then allowed to be transmitted. If the bucket becomes empty of tokens, then the arriving data must wait for more tokens to be generated before they can be transmitted.

The TBF algorithm allows saving, up to the maximum size of the bucket, n. This property means that bursts of up to n bytes can be sent at once, allowing some burstiness in the output stream and giving faster responses to sudden bursts of input.

The relation between tokens and data packets gives three possible scenarios:

- The data arrives in TBF at a rate that is equal to the rate of tokens being generated. In this case each incoming byte gets a token and passes the queue without delay.

- The data arrives in TBF at a rate that is smaller than the token rate. Only a part of the tokens are taken by the incoming data. The number of tokens eventually

accumulates up to the bucket size. The unused tokens can then be used to send data at a speed that's exceeding the standard token rate, in case short data bursts occur.

- The data arrives in TBF at a rate bigger than the token rate. This means that the bucket will soon be devoid of tokens, which causes the TBF to throttle itself for a while. This is called an 'overlimit situation'. If data keeps coming in, they will start to get dropped.

To emulate a wireless network with a highly varying bandwidth, a simple script can be written to vary the token rate at different times throughout a streaming session.

2.3 Scalable Video Format

This section introduces the meaning of scalable video. The scalable video format SPEG is a modification of a compressed video format called MPEG and is used in the research of this thesis. Section 2.3.1 provides a description of the MPEG format. Section 2.3.2 describes scalable video in general, while section 2.3.3 focuses on the SPEG format.

2.3.1 The MPEG Video Format

Moving Picture Experts Group (MPEG) is the name of a family of standards used for coding audio-visual information in a digital compressed format. With its sophisticated compression techniques, the video quality achieved is equally good compared to other coding formats, but at lower file sizes which is a major advantage.

The MPEG family of standards include MPEG-1, MPEG-2 and MPEG-4, formally known as ISO/IEC-11172, ISO/IEC-13818 and ISO/IEC-14496. The MPEG-2 format is an improvement of the MPEG-1 format with higher picture resolution and data rate. Because of this, MPEG-2 requires more space than MPEG-1 when storing a video file of equal running time.

An MPEG video stream basically consists of consecutive picture frames which define the motion picture. There are three different picture frames: I (Intra) -frames, P (Predicted) -frames and B (Bidirectional) -frames. I-frames are complete pictures that can be decoded without needing any other information. It is similar to a JPEG still image. This type of frame requires the most storage space compared to the other two types. P-frames are predictions from the previous reference frames, which could be I-frames or P-frames. The idea is to 'borrow' parts of the reference frame that are common with the current frame. It can for instance be a macroblock in the previous I-frame that hasn't moved an inch since then, so there is no need to recreate that block for this frame. The format makes use of a motion vector to derive these common parts, which are called “predictive coded” macroblocks. Parts that are not possible to borrow from the reference frame must be

encoded as I-frames, also known as “intra coded” macroblocks. So basically a P-frame consists of “intra coded” and “predictive coded” macroblocks. An estimation of the size of a P-frame is about 30-50 % of an I-frame. B-frames are also predictions from other reference frames. The difference here is that these can be predictions from both previous and later frames. This is possible due to the fact that the encoder already has access to the later frames at the start of encoding of the frame. The size of a B-frame is estimated to be around 50 % of a P-frame. A collection of consecutive frames in MPEG is known as a GOP (Group Of Pictures). An MPEG videostream is therefore built from a row of GOPs. One GOP usually corresponds to about 0.5 - 1 second of video length and consists of a combination of the three types of video frames described above. The first frame in a GOP must be an I-frame. Because of this the MPEG video stream will be easy to edit. Corruptions in the stream can be skipped by searching to the next I-frame, and it will also be possible to perform "random access" on the video stream.

2.3.2 Scalable Video

The purpose of scalable video is to make the video stream adaptable to different conditions of server and client applications, in addition to a varying bandwidth of a unreliable wired/wireless network. Scalable video can achieve this adaptability by splitting the video stream into different layers.

The lowest layer is called the base layer and has the lowest acceptable video quality. This layer is the minimum requirement that must be transmitted when streaming the video. Apart from this, there is also one or more enhancement layers. To achieve a better picture quality, these layers can be added to the base layer when possible, but at the cost of higher data rate. The amount of enhancement layers to be used will depend on the streaming application and the network capacity.

The following primary methods of scalable coding are explained in details below:

- SNR (signal-to-noise ratio) scalability
- Temporal scalability
- Spatial scalability

Each of these methods has their own way to create the base layer and the additional enhancement layer(s).

SNR (signal-to-noise ratio) scalability
This coding technique is based on the DCT (Discrete Cosine Transform) encoding. The DCT can transform a signal or image from the spatial domain to the frequency domain. As a result we get a set of frequency coefficients which measure how fast intensities of an image are changing. It should be noted that the frequency coefficients measure the difference between two neighbour pixels. The frequency coefficients can be runned through a process called quantization. The main goal of this process is to transform near-zero coefficients into zeroes. These zero coefficients represent the high frequency area. In

other words, high frequency data have been removed. Because the human eye is less sensitive to high frequency information, we can remove this without actually getting any visible loss. It is possible to adjust the level of quantization. By increasing the quantization factor, more data will be removed, and as a result we will get a reduced picture quality. In SNR scaling this is how the base layer is created. The raw video data is DCT encoded and then quantized with a large factor which will result in large amounts of data being removed. The enhancement layer is made by first running Inverse DCT on the quantized base layer. This data will then be subtracted from the original data, and the outcome will be DCT encoded once again with a lower quantization factor. This is illustrated in figure 3. The idea here is that if an enhancement layer manages to arrive at the client, then it will be added to the base layer before running Inverse DCT.

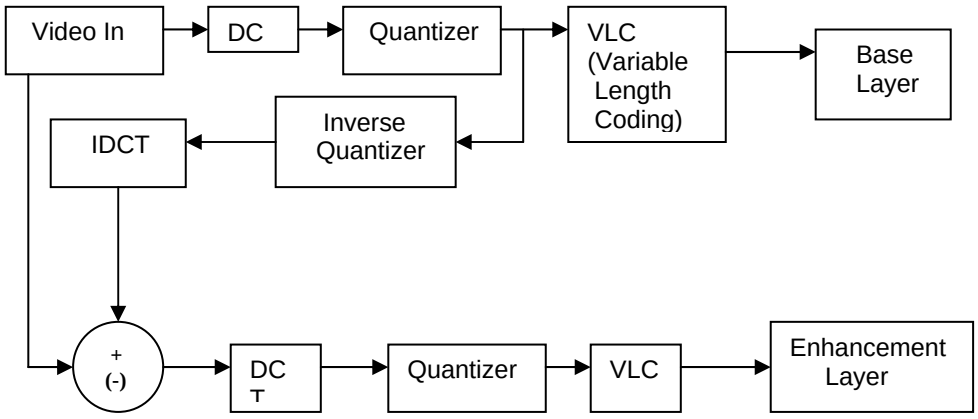


Figure 3: SNR Scalable Coding [5]

Temporal scalability

Frame rate is defined as frames per second (fps), which is the number of video frames being displayed per second. High frame rate means smoother playback of a video stream, while low frame rate results in choppy playback. The normal playback framerate is 25-30 fps. Temporal scalability is based on the manipulation of the frame rate of a video stream. The purpose here is to create a base layer video stream with low frame rate but with a minimum acceptable picture quality. The enhancement layers will be added to the base layer when possible to achieve a video stream with higher frame rate. The idea is to remove B-frames from the video stream, so that the base layer will only consist of I-frames and P-frames. It's also possible to remove P-frames if necessary. The removed frames will then become the enhancement layer(s).

Spatial scalability

In this coding technique we work at the pixel level of a video frame. The base layer consists of downsampled frames of the original images where we code less pixels. To create the enhancement layer we subtract the base layer pixels from all the pixels of a frame. If the enhancement layer manages to arrive at the client, the layers are added (DCT decoded) together to create higher resolution images.

2.3.3 SPEG – A Modification to MPEG

As mentioned, SPEG is the scalable video format used in this thesis, which is a modification to MPEG and introduces scalability in the transmission rate of a video stream. SPEG was implemented because there was no freely available implementations of layered extensions for existing video standards (MPEG-2, MPEG-4). SPEG combines temporal and SNR scalability which improves the granularity of scalability.

One thing to notice is that SPEG can be derived from the different MPEG standards, and still maintain the different standards' properties, such as picture resolution and the range of capable data rate. It follows that a SPEG file (S1) converted from an MPEG-1 source has video layers of smaller sizes than the layers of a SPEG file (S2) converted from an MPEG-2 source. This means that when streaming S2, a higher network bandwidth is required to achieve satisfying playback, compared to when streaming S1.

It should be noted that the Qstream software (section 2.6), that is used and further developed in this thesis, operates with 16 video layers in total. It divides the SPEG video stream into smaller time intervals in which each interval has 1 base layer and 15 enhancement layers. These intervals are known as adaptation window. This is explained in section 2.4. Section 4.3.6 of chapter 4 further elaborates on how the video layers are used and referred to.

2.4 Priority-Progress Streaming (PPS)

Priority-Progress Streaming (PPS) is a streaming algorithm which has the ability to adapt to the rate decisions of a TCP congestion control mechanism. The basic idea is that higher prioritized data packets are transmitted before those with lower priorities. The PPS algorithm also defines how to manage timing and priorities simultaneously.

To realize the idea of prioritized data packets, a scalable video format like SPEG has been taken into account. Since the video stream can be split into many layers, it's possible to apply priorities to the different video layers. The base layer will get the highest priority, while the enhancement layers will be marked with lower priorities. The layers are represented by units called **Application Data Units (ADU)**.

By using an algorithm called priority mapping (explained in section 2.5), the ADUs are grouped into units called **Streaming Data Units (SDU)**. The ADUs with the same timestamp will become part of the same SDU. The SDUs are then marked with priorities according to the ADUs they contain, and they are placed into a so-called **adaptation window**.

An adaptation window is meant to represent a specific time interval of the streaming video, as the PPS algorithm subdivides the timeline of the video into a sequence of time intervals using the SDU timestamps. Therefore, an adaptation window contains all the

SDUs with timestamps within its time interval. The relationship between ADUs, SDUs and adaptation windows is illustrated in figure 4. The SDUs of an adaptation window are processed by priority. The idea is that by the end of the transmission timeline of an adaptation window, all the SDUs within the window that haven't been transmitted are discarded. Recall that Qstream operates with 16 video layers in total. That is, each adaptation window has 1 base layer and 15 enhancement layers. These layers are represented by the SDUs. The SDUs of the enhancement layers are of lower priority than the SDUs of the base layer. The different levels of enhancement layers are also sorted by priority.

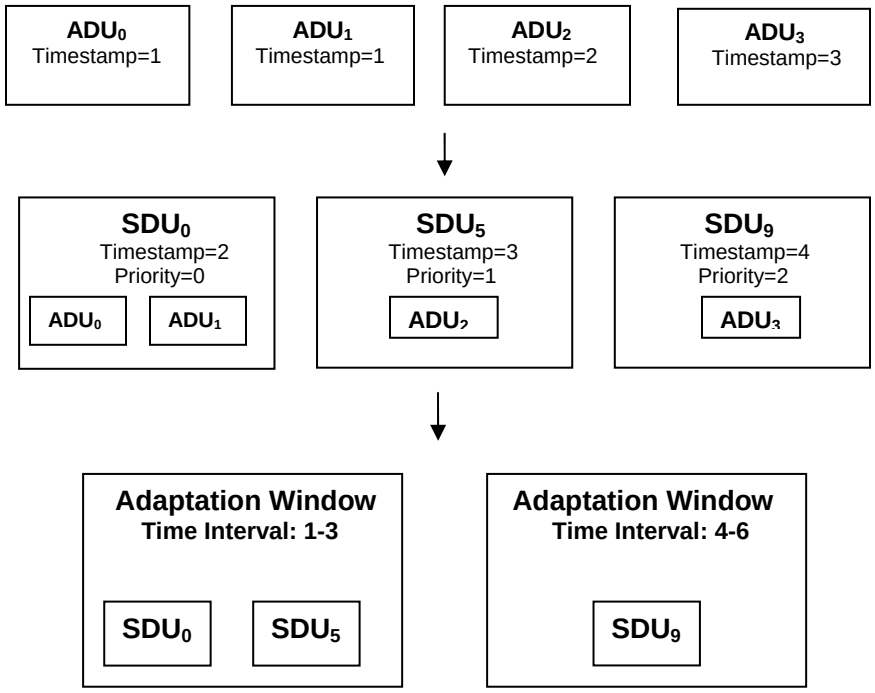


Figure 4: The Relationship Between ADU, SDU and Adaptation Windows

Based on the SDU timestamp labels, PPS can regulate the progress of the video stream to ensure that the receiver can achieve proper playback timing. The PPS algorithm consists of three subcomponents, the upstream buffer, the downstream buffer, and the progress regulator.

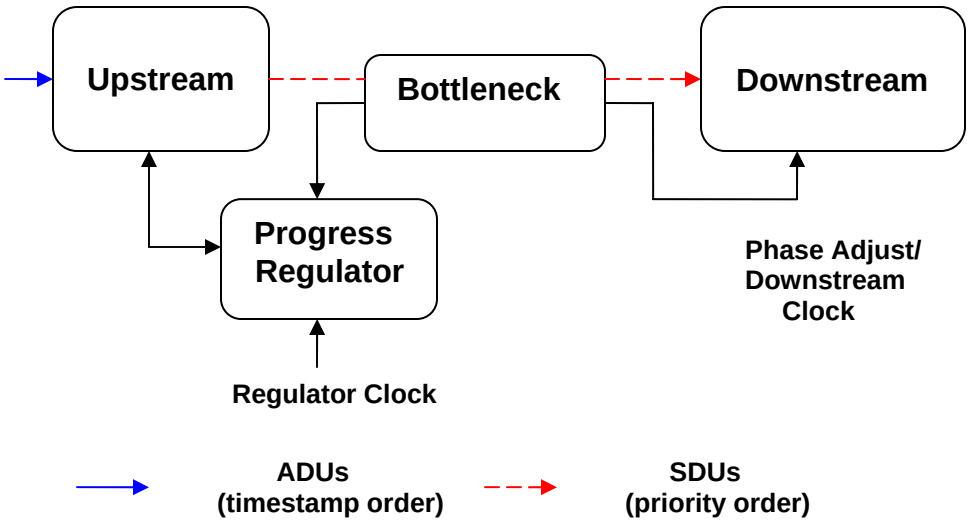


Figure 5: PPS Conceptual Architecture [17]

The upstream buffer admits SDUs within the time boundaries of an adaptation window. Time boundaries are chosen by the progress regulator, which is also responsible for advancing the window forward. This will trigger unsent SDUs from the old window position to be expired and dropped, and the window is then populated with SDUs from the new position. The SDUs flow in priority-order from the upstream buffer through the bottleneck (for example the TCP transport) to the downstream buffer, where the transmission rate is controlled by the bottleneck. Upon arrival of data, the downstream buffer will collect ADUs contained in SDUs and re-order them to their original timestamp order. The contents are then passed on for decoding and display. In the case of late arrival of SDUs, the process regulator will adjust the phase between the regulator clock and the downstream clock, in an attempt to prevent late SDUs in the future. As the bottleneck can have a varying bandwidth in certain cases, like in a wireless network, the downstream buffer may not always be able to receive all the SDUs. It will receive as many SDUs as the bottleneck allows, and the rest which are of lower priority will be dropped at the server. This method of prioritized dropping will adapt the video quality to match the network conditions between the sender and the receiver.

2.5 Priority Mapping

The previous section briefly introduced the algorithm called priority mapping. The priority mapper used in this thesis is the one included in the Qstream software. The details about this software is covered in section 2.6 and section 3.1 of chapter 3. A priority-mapper assigns priorities to the units of a media stream, so that priority drop yields the most graceful degradation, as appropriate to the viewing scenario. The mapper used by Qstream is depicted in figure 6.

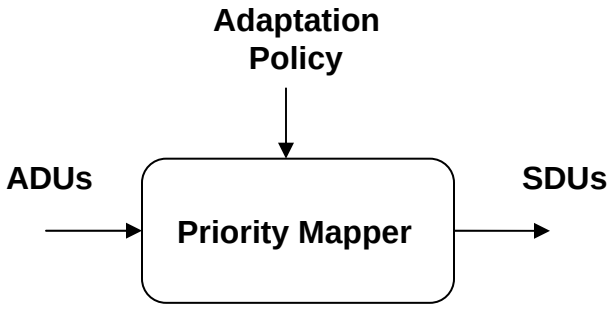


Figure 6: Priority mapping [17]

The inputs are ADUs and the quality adaptation policy. The output of the mapper is a sequence of SDUs. Each SDU contains a subset of the input ADUs, a timestamp and a priority computed by the mapper algorithm.

The adaptation policy consists of utility functions where users can specify their preferences. Figure 7 shows the general form of a utility function.

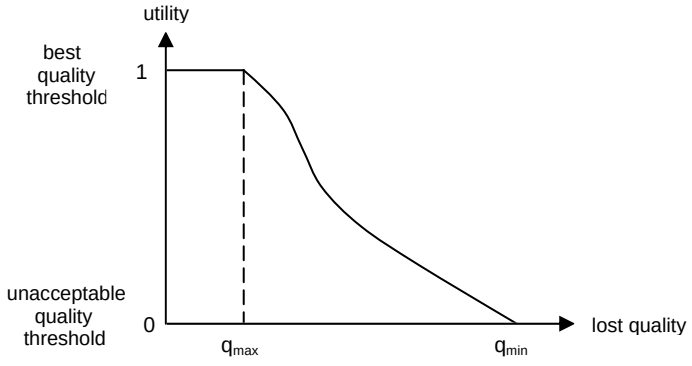


Figure 7: Utility function [17]

The horizontal axis describes an objective measure of lost quality, while the vertical axis describes the subjective utility of a presentation at each quality level. The region between the q-max and q-min thresholds is where a presentation has acceptable quality. The q-max threshold indicates a point where the quality of the presentation is as good as perfect, while the q-min threshold marks a point where lost quality has reached beneath an acceptable level. In the case of priority mapping for SPEG, the adaptation policy contains two utility functions, one for spatial quality and one for temporal quality.

The mapping algorithm subdivides the timeline of the video stream into intervals called mapping windows and prioritizes the ADUs within each window separately, which is done in two phases.

In the first phase, the ADUs are partially ordered according to a "drop before" relationship. This means that base layer ADUs should not be dropped before their corresponding enhancement layer ADUs. This kind of ordering constraint represent hard dependency rules, in that they simply reflect SPEG semantics. There are also soft dependency rules which ensure that frame dropping is spaced as evenly as possible. For example, if half the frames are to be dropped, then it is better to drop every other frame rather than clustered dropping such as keeping even GOPs and dropping odd GOPs.

In the second phase, the adaptation policy is used to refine the partial ordering from the first phase, generating the prioritized SDUs. The algorithm works through an iterative process of elimination of ADUs. For each iteration a set of candidate ADUs (initially all ADUs from the mapping window), that are still in the set of unprioritized ADUs, is considered. The mapper computes, for each of these candidate ADUs and quality dimension (spatial and temporal in SPEG), the presentation quality that would result if the candidate ADU was dropped. For the temporal quality dimension, the mapper computes the frame rate. For the spatial quality dimension, the spatial level is computed. The utility functions are used to convert the computed quality values to corresponding utilities. The candidate ADU that has the highest utility is selected as the next victim, as that ADU will have the smallest impact on utility when dropped next. The priority value for the victim ADU is a linear (inverse) fitting of the utility into the range of priority values. The iterations stop when all ADUs have been assigned a priority. Once the mapper has assigned priorities to all of the ADUs in a map window, it groups them into SDUs. In this mapping algorithm, there is one SDU per priority level, which contains all the ADUs that ended up with the same priority. Another main attribute of an SDU is its timestamp. The SDUs are all set to have the same timestamp as the first video frame in the whole map window. All the ADUs in a map window are grouped into a single set of SDUs, distinguished by priority, but sharing the same timestamp.

2.6 Overview of Qstream

Qstream is one of the softwares that makes use of the PPS protocol. It consists of several components. Among them is a component called Qvid, which is the video streaming system that supports quality-adaptive streaming over TCP, and is based on the notion of priority data dropping. It uses the scalable video format SPEG for streaming.

Qvid is actually a collection of several programs, and these are the most significant ones for this thesis:

- **StreamServ** - The main functions of this program are video retrieval, priority mapping and PPS transmission. The video retrieval is either a stored SPEG file, or a live video capture from a webcam that's being encoded to SPEG in real time. It should be mentioned that this thesis doesn't include the latter part.

- **StreamPlay** - This player represents the receiver side of the PPS protocol. It takes care of video decoding and display, and also defines the usual functions of a video player.

- **FileServ** - This program is responsible for checking that a requested video bitstream and index files are available for streaming. It also prepares the requested media file for StreamServ to fetch.

2.6.1 Collaboration of the Three Programs

In this thesis Qstream is configured to work in a unicast streaming mode, as figure 8 shows. The architecture basically consists of two nodes, the upstream node and the downstream node. The upstream node contains the two programs FileServ and StreamServ, while the downstream node consists of StreamPlay. FileServ and StreamServ were initially two separated programs, but were later merged into a single, dual-threaded program for an ease of use purpose.

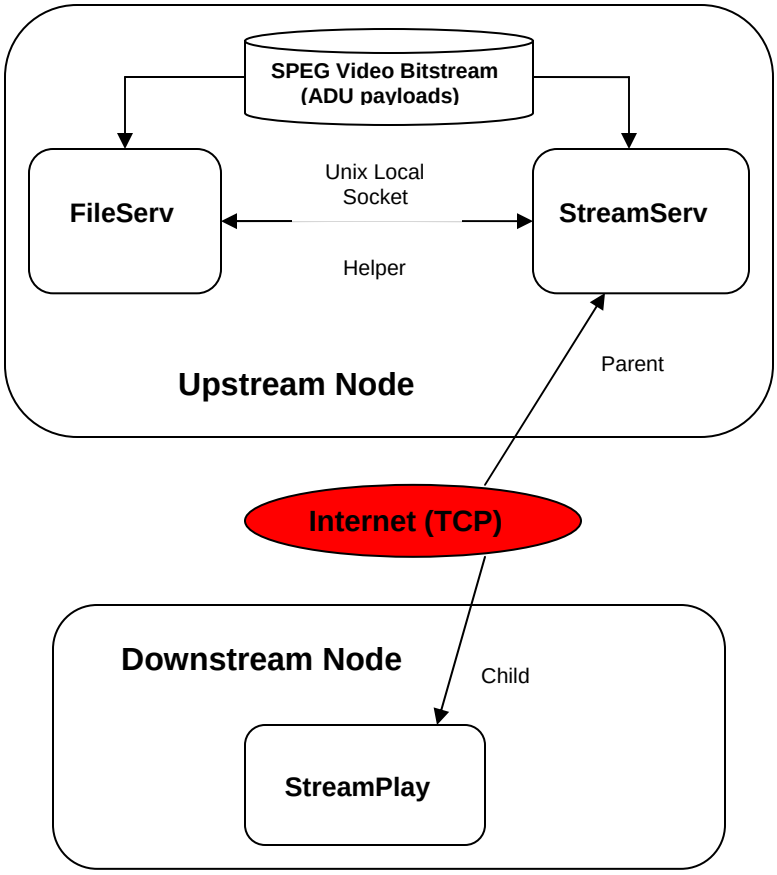


Figure 8: Qstream architecture for unicast mode [17]

This section provides an overview of how the three programs work together to achieve streaming of media files over TCP using the PPS algorithm.

At the starting point, both StreamServ (the streaming server) and FileServ (the file server) put themselves in a state to accept requests. When StreamPlay (the streaming client) starts, it gets the name of the video to request from the command line input. It then attempts to connect to StreamServ. If it succeeds, then StreamServ responds with an acceptance message.

StreamServ and StreamPlay must also each initialize a PPS session to handle the streaming session. The PPS session of StreamServ takes care of everything that happens with the streaming session at the server side, while the PPS session of StreamPlay is responsible for all that happens, within the same streaming session, at the client side.

It should be noted that the different parts of the system use message passing to communicate with each other. After the client is successfully connected to StreamServ, it forwards the video request to StreamServ by wrapping the request inside a predefined message shell. StreamServ gets the request and wraps it in a new message which it

forwards to FileServ. Upon receiving the message, FileServ tries to find the requested video. Details about the message passing protocol used are covered in section 2.6.2.

The next step for StreamServ is to initialize an adaptation window. As mentioned earlier, an adaptation window represents a fixed timeline in the streaming video. Thus, it requires several adaptation windows to cover the whole timeline of the streaming video.

After initializing an adaptation window, StreamServ contacts FileServ to retrieve the range of video data that falls within the interval of the window. The window is responsible for this range of video data in the sense that it contains a pointer to the data. It should be noted that as soon as an adaptation window is ready to be transmitted, StreamServ initializes and prepares the next adaptation window in the video timeline.

When the range of data is retrieved, StreamServ runs the adaptation window through the mapping function, which is described in section 2.5. A deadline is computed for the adaptation window to determine when the transmission of the window should start. When the time comes for the adaptation window to be transmitted, StreamServ sends a window start message to the client to indicate that the streaming is about to begin. The client will initialize its stream clock and be prepared for the adaptation window to arrive. StreamServ also schedules a timeout for the adaptation window to mark when the window expires. If the timeout fires before the transmission of the entire adaptation window contents complete, then the algorithm proceeds to drop unsent data for the window. If an adaptation window finishes before its expiry timeout, then StreamServ has to cancel the timeout scheduled for that specific window.

As mentioned, StreamServ prepares the next adaptation window when the current window is ready for transmission. The preparation of the next adaptation window is then done concurrently with the transmission of the current one. If the preparation of the new window finishes before its transmission time, then it is scheduled with a deadline as indication of when to start the transmission. However, if the deadline happens to have already past by the time the preparation finishes, the scheduler starts the transmission of the window as soon as possible.

On the client side a play window is created when the client receives the window start message from StreamServ. The play window is initiated with the contents of the window start message, and later it is used to store the data that arrive. That is, one play window is created for each adaptation window arriving. The expiration time of the play window is also being computed and put into a scheduling scheme. When the window expires, it is put into a scheduling queue for decoding and display.

2.6.2 GAIO and QSF

Another component of Qstream is qsf. This is actually a collection of modules that collectively makes up the Quasar Streaming Framework library (libqsf). This library is used by the rest of the Qstream software and contains two main modules, GNU Asynchronous IO (GAIO) and Quasar Streaming Framework (QSF). These two modules realize the idea of a reactive programming model, which basically means an event-based programming model.

An event is a notification of either a result of a requested IO operation being ready, an expiration of a scheduled deadline or an available time slot to do a computation. Only one event is active at a time. Once invoked, it is allowed to execute to completion. This means that a long running computation should be spread across multiple events. IO and computation events are prioritized, and an application can dynamically set these priorities according to its own needs.

The GAIO library provides the core Application Programming Interface (API) for reactive programming in Qstream. It offers services that make it possible to schedule events for execution immediately or at a given time deadline. It also contains a GAIO event dispatcher which is the core of the application's state machine. The applications (StreamServ and StreamPlay) call functions of GAIO and QSF, providing an event-handler callback parameter (in the form of a C function pointer) to be invoked when the requested action is complete (the IO has completed or the deadline expired). The GAIO event dispatcher can schedule the handlers for the mix of IO completion events, deadline events and immediate events. It ensures that executions of event handlers are atomic, which means that every time an event handler is dispatched, it is allowed to run to completion before another handler can be dispatched.

An event handler of a specified priority can be scheduled for execution as soon as possible. For example, if computations are running by scheduling and dispatching a long running loop of separate events, the loop may be interrupted if another higher priority event occurs, such as an IO completion or a deadline expiry. It's also possible to allow the application to schedule an event handler for execution at a set deadline time by using a timer primitive. The deadline can be specified as an absolute time, or time relative to the current time. When the deadline expires, the event dispatcher will execute the handler as soon as possible.

GAIO also provides a tool called Worst Case Execution Time (WCET). As the name indicates, this tool measures the duration of event handlers that are dispatched, with the purpose of estimating if the running time of the handlers affect the overall timeliness of an application.

Qstream includes a second library called QSF (Quasar Streaming Framework) which provides services that are more specialized to network streaming applications. QSF is a higher level API built on top of GAIO for message passing protocols, including

connection establishment for clients and servers (initiate and accept connections), message creation helpers for clients, and message dispatching for servers.

A set of routines for message formatting for logging and debugging are also implemented to help understanding the dynamics of the program executions. QSF also provides support for using OS real-time scheduling. However real-time running might be dangerous, as it could lead to live-locks which could crash the whole system. QSF solves this by creating a watchdog process that can detect and kill the application if a live-lock occurs. QSF was developed with the intention of directly supporting the message oriented style of the PPS protocol. It provides a generic API for message oriented protocols, of which PPS is one instance. The goal is to provide a simple API for sending and receiving messages.

All messages share a generic message header. The length and type fields contain the essential information necessary to implement message oriented communication over a (TCP style) reliable, byte-stream session. The length field indicates the size in bytes of the message body that follows the header. The type field indicates what kind of message that is contained in the body, and it's application specific. The magic field is for debugging purposes, like to detect corruption of the basic framing of messages. StreamServ, StreamPlay and FileServ make use of this message passing protocol to communicate with each other during a streaming session. To distinguish between the different connections and functions of these programs, naming conventions have been introduced. StremServ's connection to FileServ is called helper. Whenever StreamServ wants to send a message to FileServ, it passes the message via functions with the word helper as part of name indication. StreamServ's downstream connection to StreamPlay is called child, and StreamPlay's upstream connection to StreamServ is called parent. The messages exchanged between StreamServ and StreamPlay comprise the PPS protocol. Each PPS session begins with StreamPlay establishing a transport level connection to StreamServ, which in turn establishes a connection to Fileserv. From then on, the PPS session consists of a sequence of application level messages exchanged across the transport connections.

Chapter 3: The Analysis

This chapter is divided into two main sections of analysis. The first section is an elaboration of section 2.6.1 of chapter 2, which provides a deeper insight into the code of Qstream. The second section proposes improvements to the existing code.

3.1 Part I – A Closer Look at Qstream

This first part of the analysis presents the server and client algorithms for PPS in more depth. The focus is put on the various C-coded functions of StreamServ and StreamPlay. Sections 3.1.3 – 3.1.6 present the streaming scenario in four phases, which are explained in details with references to the functions called in the different steps. These four phases include start-up of programs, creation and initiation of data structures, requesting and fetching of a media file, preparation of adaptation windows, and the actual streaming process. Section 3.1.1 describes the naming conventions used by the functions of StreamServ and StreamPlay, while section 3.1.2 provides a description of the most relevant data structures used by these two programs. Section 3.1.7 describes an important strategy of Qstream, which is known as work conservation. Finally, the last section of this chapter describes an option included in the PPS algorithm, which is known as window scaling.

3.1.1 Naming Conventions

Each of the programs (StreamServ, StreamPlay and FileServ) contains several functions which have a particular naming convention. Each function name begins with a prefix that identifies which Qstream program or to which library the function belongs. The *ss* prefix is for functions in the StreamServ program, *sp* is for functions in StreamPlay, and *fs* is for functions in FileServ.

The qsf prefix is used for QSF library functions. The second component of the name, as mentioned in section 2.6.2 of chapter 2, refers to the object on which the function acts or was triggered by. These are the words *helper*, *child* and *parent* which refer to StreamServ's connection to FileServ, StreamServ's downstream connection to Stream-Play, and StreamPlay's upstream connection to StreamServ, respectively. The remaining suffix of the name describes the action performed by the handler.

3.1.2 Data Structures

There are two main data structures in the StreamServ program, one for state related to a PPS session and the other for individual adaptation windows. These are shown in figure 9 and figure 10.

The ServPpsSession object is allocated for each PPS session (section 2.6.1 of chapter 2). As the names *child* and *helper* are references to StreamPlay and FileServ respectively, the *child_session* and *helper_session* fields are handles used to exchange messages with the programs via QSF.

The next three fields show the different queues for adaptation windows that each session maintains. As mentioned in section 2.6.1 of chapter 2, the adaptation windows go through different steps in the preparation stage. The *recv_windows* queue holds adaptation windows as they are initialized with the video data range fetched from storage. The *mapped_windows* queue holds adaptation windows that have been prioritized but are not ready for transmission yet. The *xmit_windows* queue holds adaptation windows that the mapper has finished prioritized and are eligible for transmission.

The *workahead_limit* field is used for configuration of a work conserving mode. In this mode, the transmission of an adaptation window is allowed to start immediately if bandwidth was enough for the previous window to finish before its deadline. However, this is allowed only up to the configurable *workahead_limit* of the session. It should be noted that this *workahead_limit* variable is an essential part in the research of this thesis. More details are covered in section 3.1.7.

The *map_session* field is a pointer to a MapSession object that contains state for the mapper algorithm. The variables *expand_end*, *shrink_start* and *growth_rate* are used to update the size of adaptation windows for an option of PPS called **window scaling** (section 3.1.8).

```
ServPpsSession {
    QsfSession      child_session;      //Handle for TCP connection downstream
    QsfSession      helper_session;     //Handle for local connection to FileServ
    Queue           recv_windows;       //Windows captured/fetched from storage
    Queue           mapped_windows;     //Windows mapped beyond workahead limit
    Queue           xmit_windows;       //Adaptation windows ready to stream
    StreamHeader    stream_header;      //Contains video duration, fps, etc.
    Integer         video_fd;           //File descriptor for video data
    Time            session_origin;     //Base time of regulator clock
    Time            phase_offset;       //Worst case transport latency
    Time            workahead_limit;    //For work conserving mode
    MapSession      mapper_session;     //Mapper specific session state
    Time            expand_end;         //When window sizes stop growing
    Time            shrink_start;       //When window sizes start shrinking
    Time            prev_vid_end;       //Video end position of latest window
    Float           growth_rate;        //How fast the windows grow/shrink
    Boolean         serv_ready;         //First window ready for transmit
    Boolean         child_ready;        //Child has sent start request
}
```

Figure 9: The PPS Session Object for StreamServ [17]

The ServPpsWindow object is allocated per adaptation window (section 2.4 of chapter 2). The *vid_start* and *vid_end* fields delimit the position of this window within the video timeline. The *xmit_start* and *xmit_end* fields are the window's position within the transmission timeline.

The *start_timeout* field stores a handle to a scheduled GAIO event (section 2.6.2 of chapter 2), which in this case is the event that enqueues the window for transmission. This field can be manipulated by the *workahead_limit* variable in the work conserving mode, as a decrease of value in the *start_timeout* field will make the window start its transmission phase earlier than originally planned (step 5 of phase III, section 3.1.5). On the other hand, the *xmit_timeout* field is used to issue an GAIO event that will trigger the transmission phase to stop for the current window. In this case, recall from section 2.4 of chapter 2 that unsent SDUs are dropped, and the next window is allowed to start.

The *fetch_done* flag is used to track whether FileServ has fetched the contents. When the contents from FileServ are received, the *adus* field will store these unprioritized ADUs. The mapper algorithm will consume the content of adus, and then prioritize and group the ADUs transforming them into SDUs (sections 2.4 and 2.5 of chapter 2). The mapper will proceed to insert the SDUs into a heap data structure, which is pointed to by the *sdus* field.

```
ServPpsWindow {
    Time        vid_start;      //Start position in video timeline
    Time        vid_end;        //End position in video timeline
    Time        xmit_start;     //Start position in transmit timeline
    Time        xmit_end;       //End position in transmit timeline
    Time        xmit_deadline;  //End position in absolute time
    Boolean     fetch_done;     //Window has arrived
    Queue       adus;           //Window contents before mapping (time order)
    Heap        sdus;           //Window contents after mapping (priority order)
    Timeout     start_timeout;  //Handle to timeout scheduled to start transmit
    Timeout     xmit_timeout;   //Handle to timeout scheduled to stop transmit
}
```

Figure 10: The Adaptation Window Object for StreamServ [17]

```
PpsSdu {
    Time        timestamp;      //Derived from map window
    Integer     priority;       //Assigned by mapper
    Integer     num_adus;       //PpsAdus follow, then ADU payloads
    PpsAdu      adus[num_adus]; //Index ADU payloads
    Bytes       payloads[];     //ADU payloads (variable length)
}

PpsAdu {
    FileOffset  offset;
    Integer     length;
}
```

Figure 11: SDU and ADU objects [17]

The data structures for the SDU and ADU objects can be seen in figure 11. An SDU contains the timestamp and priority, along with a group of ADUs. The *adus* field is an

array with pointers to the ADU objects, which describe the logical location of the ADUs within the video bitstream.

Similar to StreamServ, there are two main data structures in StreamPlay, a per-session object called PlayPpsSession and a per-adaptation window object called PlayPpsWindow, shown in figure 12 and figure 13.

A PlayPpsSession object is allocated for each active PPS session (section 2.6.1 of chapter 2). The *parent_session* field is a handle to the network socket to StreamServ corresponding to the PPS session. The *session_origin* field contains the start time of the transmission phase for this session and is used as the basis for converting time of day values to the transmission and display timelines. The *slack* field is used to mark, for each arriving SDU, the remaining time before the deadline for the adaptation window currently in transmission phase. A negative slack value means that the last SDU received was late, and the value is then used to maintain the correct phase offset between the StreamServ and StreamPlay clocks.

The *xmit_window* field is a pointer to a PlayPpsWindow object (seen in figure 13) which corresponds to the current adaptation window of the transmission phase. The *decode_windows* field is a queue of PlayPpsWindow objects for adaptation window(s) in the process of decoding. In the work conserving mode, this queue makes it possible for the transmission phase to work more than one window ahead of the decode and display phase.

```
PlayPpsSession {
    QsfSession      parent_session;     //Handle for TCP connection upstream
    StreamHeader    stream_header;      //Contains video duration, fps, etc.
    Time            session_origin;     //Base time of regulator clock
    Time            slack;              //How early did last SDU arrive?
    PlayPpsWindow   xmit_window;        //Window in transmission
    Queue           decode_windows;     //Adaptation window in decode/display
}
```

Figure 12: The PPS Session Object for StreamPlay [17]

The PlayPpsWindow object is instantiated for each adaptation window in the video timeline. The *vid_start* and *vid_end* fields delimit the position of the window in the video timeline. The *xmit_start* field indicates when the window must begin its transmission phase, while the *xmit_end* field contains the time at which the decode/display phase for the window must start. The *xmit_timeout* field is a handle to a scheduled timeout, which can also be used to cancel the timeout in the event that processing of the window completes prior to the timeout deadline.

The *num_base_sdus* and *num_sdus* field are used to detect conditions such as when the base layer is complete and when the entire window is complete. The *adus* field is a handle to the heap data structure that is used to sort the contents of SDUs from priority order back to the original time order.

```
PlayPpsWindow {
    Time        vid_start;          //Start position in video timeline
    Time        vid_end;            //End position in video timeline
    Time        xmit_start;         //When should xmit start
    Time        xmit_end;           //When should xmit end
    Timeout     xmit_timeout;       //Handle for cancellation
    Boolean     decode_started;     //Has decode already started?
    Integer     sdu_count;          //How many SDUs so far
    Integer     num_base_sdus;      //How many SDUs in base layer
    Heap        adus;               //Window contents (time order)
}
```

Figure 13: The Adaptation Window Object for StreamPlay[17]

A field that both the ServPpsSession and PlayPpsSession objects share in common, is the *stream_header* field which points to a StreamHeader object that contains configuration information from the video. The StreamHeader object is shown in figure 14.

```
StreamHeader {
    VideoRate   video_rate;         //Fps, timecode settings
    Time        duration;           //Total duration of stream
    Integer     h_size;             //Horizontal resolution
    Integer     v_size;             //Vertical resolution
    Time        preroll_duration;
}
```

Figure 14: The StreamHeader Object [17]

When StreamServ receives this object from FileServ, it passes it forward to StreamPlay. The resolution information (*h_size* and *v_size* fields) is used by StreamPlay to initialize the display window during startup. The *preroll_duration* variable is used to inform the PPS of the smallest feasible adaptation window duration. The *video_rate* and *duration* fields indicate the framerate and running time of the video respectively. Both StreamServ and StreamPlay need to have this information.

3.1.3 Phase I – The Setup

This phase involves the start-up of the three programs and the process of how they connect to each other. The necessary data objects will be created and initiated, like the ServPpsSession and PlayPpsSession objects. This is depicted in figure 15.

1) When StreamServ starts up, it creates a thread that triggers FileServ to start running as well. The two programs will then put themselves in a ready state for further requests.

2) When StreamPlay starts up, its main function calls *sp_stats_ready* to get the name of the requested media file by parsing a list of names built from the input command. This function then calls *sp_parent_connect*, passing the name as an argument. *Sp_parent_connect* creates and initiates the necessary data objects, like the PlayPpsSession object. The name of the requested file is stored in the PlayPpsSession object for later use. Then by calling qsf_connect, StreamPlay initiates a connection to StreamServ.

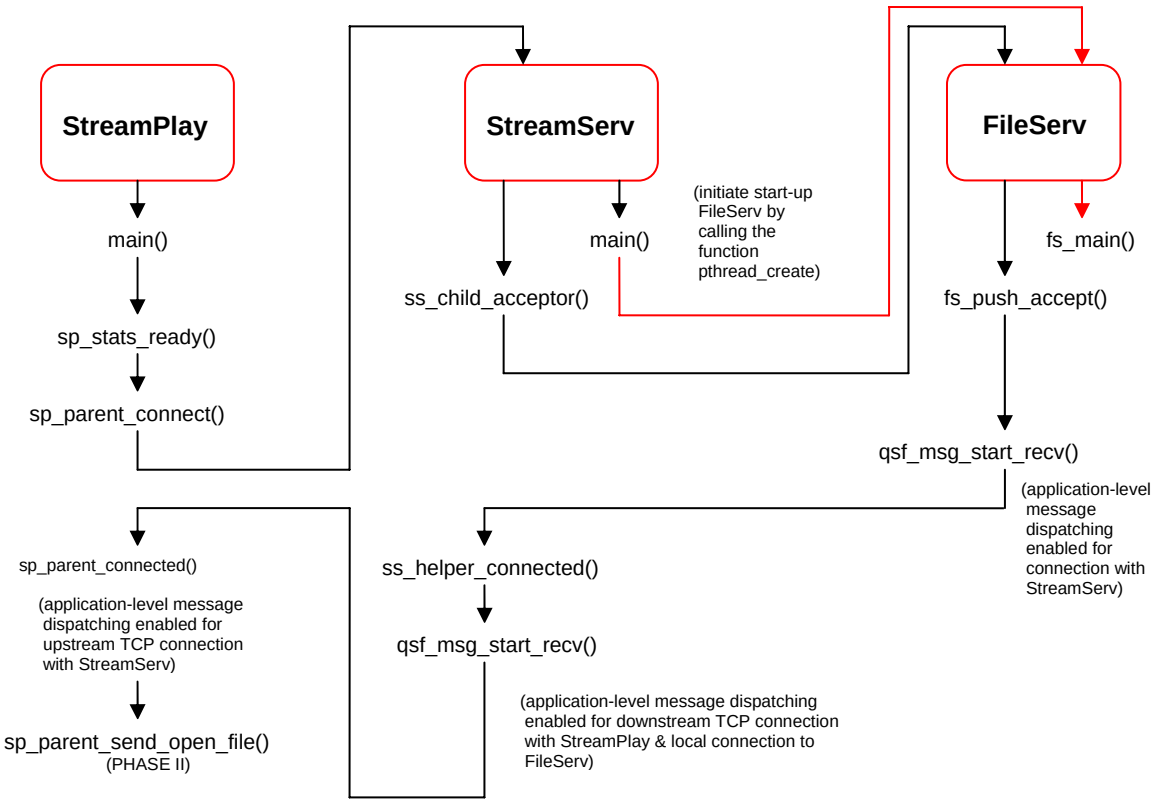


Figure 15: Phase I

3) The function *ss_child_acceptor* of StreamServ is called when the new connection with StreamPlay has been accepted. This function creates a ServPpsSession object, and then it initiates a local connection to FileServ.

4) FileServ's function *fs_push_accept* is called when the new connection with StreamServ has been accepted. FileServ then calls the function *qsf_msg_ start_recv* to enable the application-level message dispatching for incoming messages. FileServ then responds to StreamServ, reporting that the connection has been accepted.

5) When the response from FileServ arrives at StreamServ, the function *ss_helper_ connected* is called. This function calls *qsf_msg_start_recv* twice, to enable the application-level message dispatching for both the downstream TCP connection to StreamPlay and the local connection to FileServ.

6) It follows that StreamPlay gets the similar kind of response from StreamServ, reporting that the connection to StreamServ has been accepted. StreamPlay's function *sp_parent_connected* is called to start message dispatching for incoming messages on the connection to the parent (StreamServ). It then calls the function *sp_parent_send_open_file* which is the starting point of phase II.

The code of this phase will remain unchanged in the further research of this thesis, as it's not a part of the code that needs to be improved.

3.1.4 Phase II – The File Request

This phase will describe how the file request process is handled by the three programs and is depicted in figure 16.

1) StreamPlay's function *sp_parent_send_open_file* prepares a qsf message with all the necessary information about the file to be requested, like the name of the file and the length of the name. The qsf message is then sent to StreamServ via the application-level message passing protocol enabled in phase I, also discussed in earlier sections.

2) StreamServ's function *ss_child_recv_open_file* is called upon receiving the qsf message from StreamPlay. This function simply forwards the message to FileServ.

3) Upon receiving the message, FileServ's function *fs_push_open_file* is called. This function performs a look-up for the requested file. If the look-up succeeds, then the function prepares a StreamHeader object (refer to data structure section) with relevant information about the requested file, that it sends back to StreamServ via a qsf message.

4) The function *ss_helper_recv_open_file* of StreamServ is called upon the arrival of the response message from FileServ. This message is then forwarded to StreamPlay, as it includes the StreamHeader object which StreamPlay needs to initialize its display. The function *ss_win_prep_first* will then be called by *ss_helper_recv_open_file*, with the starting time point of the video passed as an argument.

5) The purpose of the function *ss_win_prep_first* is to allocate a ServPpsWindow object for the first adaptation window. It initializes all the necessary variables of the window object and puts the new window in the *recv_windows* queue, where it will stay until the preparation is complete. The function *ss_helper_send_read_range* is then called to initiate FileServ to locate and fetch the contents of the window from storage. This marks the entrance to phase III.

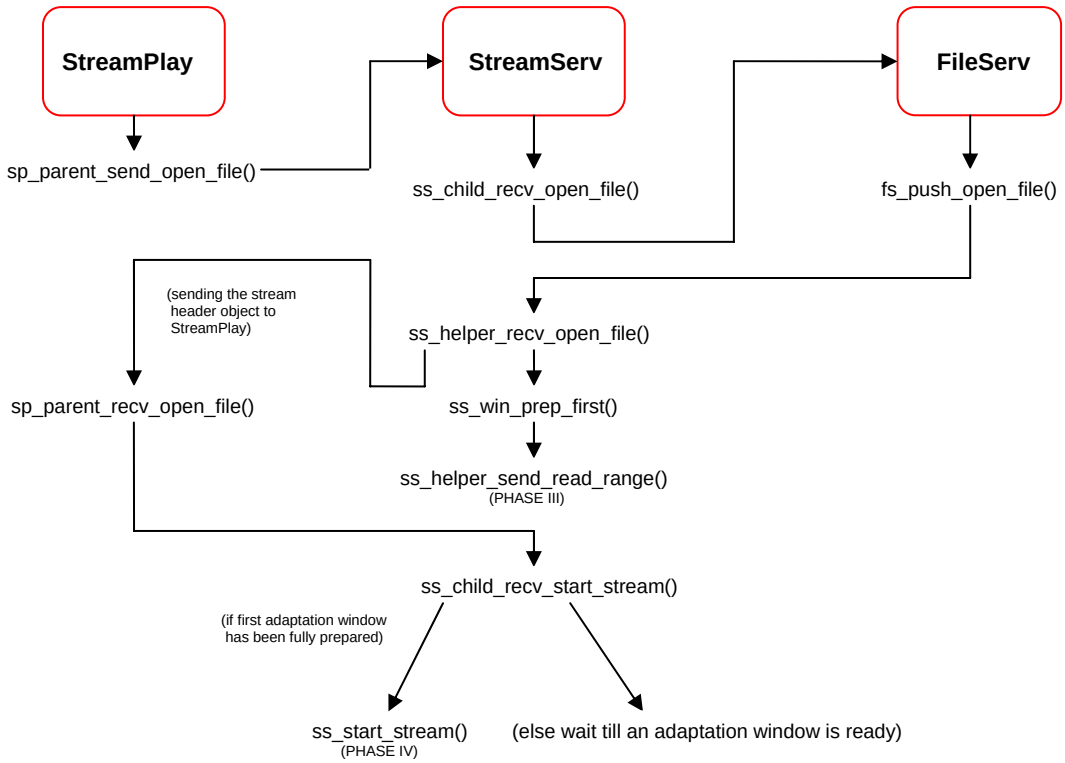


Figure 16: Phase II

6) When StreamPlay receives the message from StreamServ, the function *sp_parent_recv_open_file* is called. The StreamHeader object is extracted, and it is used to

initialize play-out, such as duration of the stream, width and height of the video, and frame rate. The function then sends a qsf message to StreamServ to indicate that the player is ready to commence streaming.

7) The message from *sp_parent_recv_open_file* triggers StreamServ's function *ss_child_recv_start_stream* to be called. This is an indication that the child (StreamPlay) is ready to receive the first adaptation window. The function *ss_child_recv_start_stream* tests if the first adaptation window has been fully prepared. If so, then the function *ss_start_stream* is called, which starts phase IV. This assumes that phase III has finished, which is explained in the next section.

The code of this phase will remain unchanged in the further research of this thesis, as it's not a part of the code that needs to be improved.

3.1.5 Phase III – The File Fetching and Window Preparation

This phase provides a description of how the requested file is fetched and prepared for streaming, and it is depicted in figure 17. It should be noted that the processes of this phase is repeated for each new adaptation window.

1) When StreamServ has finished preparing an adaptation window, as explained in phase II, it will call the function *ss_helper_send_read_range*. This function prepares a qsf message which includes the starting and ending time points of the window. The message is sent to FileServ to locate and retrieve all the ADUs (from the requested media file) that fall within the time points specified.

2) When the message arrives at FileServ, it triggers the function *fs_push_range_get*. This function fetches the requested range of the media file and prepares a reply message with necessary information, that it sends back to StreamServ.

2) The response message from FileServ triggers the function *ss_helper_recv_read_range* of StreamServ. The message is parsed by the function to get the content, which is an array of ADU descriptors for the ADUs that fall within the range requested. These are stored for later processing by the priority mapper. The window is then put into a scheduling queue, where the mapper function *ss_map_adus* is scheduled to be run as soon as no other higher priority events are pending.

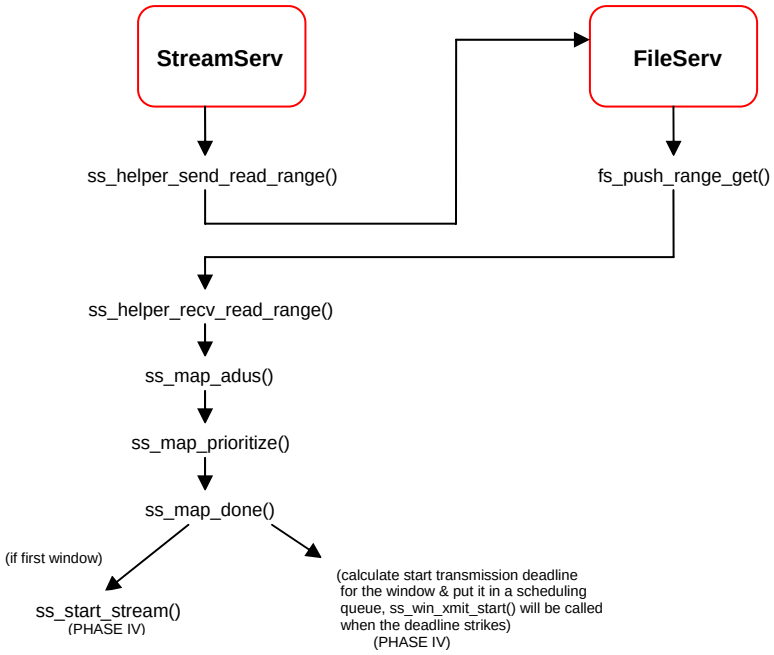


Figure 17: Phase III

4) When the mapping event for the window is dispatched, the function *ss_map_adus* is called. This one will further call another function named *ss_map_prioritize* to prioritize the contents of the adaptation window, according to the mapper algorithm described in section 2.5. Recall that the adaptation window consists of a sequence of one or more smaller intervals called map windows. The mapper is scheduled to process the ADUs in all the map windows until the entire adaptation window is empty.

5) When the entire adaptation window has been prioritized, *ss_map_adus* calls the function *ss_map_done*. The window is then moved from the *recv_wins* queue to the *mapped_wins* queue. If the current window is the very first adaptation window in the stream, then *ss_map_done* proceeds to call the function *ss_start_stream*, which marks the entrance to phase IV. For all other windows other than the first, the deadline at which transmission of the windows should start is computed. If there is a workahead time available, then it is subtracted from the transmission start deadline of a window also. This makes it possible for a window to start its transmission phase earlier than planned. The window is put into a scheduling queue, where the function *ss_win_xmit_start* will be dispatched at the deadline. If the deadline happens to have already past, then the event dispatcher will call *ss_win_xmit_start* as soon as possible. The *ss_map_done* function also checks whether the next adaptation window is available and ready for mapping. If so, an event is scheduled to invoke the mapper.

The code of this phase will remain unchanged in the further research of this thesis, as it's not a part of the code that needs to be improved.

3.1.6 Phase IV – The Transmission

This final phase describes the process of transmitting an adaptation window. It should be noted that when an adaptation window is starting its transmission phase, the preparation of the next window is being done concurrently (phase III). The following presentation is depicted in figure 18.

1) The call of the function *ss_start_stream* marks the start of this phase. This function can be dispatched in two different ways. If StreamServ is not ready with the preparation of the first adaptation window by the time the child (StreamPlay) is ready to receive this window, then *ss_map_done* will be responsible for calling *ss_start_stream* when the preparation is finished (phase III). However, if the child is not ready by the time StreamServ is finished preparing the first adaptation window, then *ss_map_done* will not call ss_start_stream. Instead the function *ss_child_recv_start_stream*, which is dispatched when the child is ready, calls the function *ss_start_stream* (phase II). That is, the main task of *ss_start_stream* is to initiate the transmission phase for the first adaptation window in the stream. It calls the function *ss_child_send_stream_start*, which purpose is to send a message to StreamPlay reporting that the streaming is about to begin. *Ss_start_stream* also calls the function *ss_win_xmit_start* to begin the transmission.

2) When StreamPlay receives the streaming-start message from StreamServ, the function *sp_parent_recv_stream_start* is dispatched. The client side stream clock is initialized, and StreamPlay is then prepared to receive the first adaptation window.

3) On the server side, *ss_win_xmit_start* has been called. The current adaptation window is moved from the *mapped_wins* queue to the *xmit_wins* queue. Thus, the window is ready to be transmitted. A timeout is scheduled to mark the expiration of the window. If the timeout fires before the transmission of the entire window contents complete, then the algorithm will proceed to drop unsent SDUs for this window. For the first adaptation window, *ss_win_xmit_start* calls the function *ss_child_send_win_start* to begin the transmission. Then the *ss_win_prep_next* function is called to initiate preparation of the next adaptation window in the video timeline. It calls *ss_helper_send_read_range*, which repeats the processes of phase III for the next window. This preparation will then proceed concurrently with the transmission of the current window.

4) The function *ss_child_send_win_start* initiates a window start message and sends it to StreamPlay before it starts the transmission. The window start message contains infos such as the number of SDUs expected for this particular adaptation

window, the start time of the video stream and the end times of the video and the transmission. Then the function initiates another message to be sent downstream to StreamPlay, containing a notification that the function ss*_child_send_sdu_head* is called whenever StreamPlay is ready to receive the adaptation window.

5) The SDU data structure consists of a header and a payload. The header part of the SDU contains an array of ADU descriptors which specifies offset and length of each ADU within the video bitstream, and the payload consists of the ADUs that have been grouped together. StreamServ starts by sending the SDU header first. This is done by the function *ss_child_send_sdu_head*. This function then proceeds to call the *ss_child_send_adu* function to transmit the first ADU of the SDU. *Ss_child_send_adu* repeatedly calls itself to transmit the next ADU in the same SDU, until the SDU is empty. After finishing one SDU, the function *ss_sdu_next* is called.

6) The function *ss_sdu_next* is called to transmit the next SDU in the adaptation window. If there are still SDUs left, then *ss_child_send_sdu_head* is called to begin transmission of the next SDU. The pick of a SDU depends on the priority order. However, if the heap in which the SDUs are stored is empty or the current adaptation window is expired, then the function *ss_win_done* is called. It should be noted that an adaptation window is allowed to ignore its expiration deadline, if the expiration occurs before the base layer is finished transmitting. The transmission will end as soon as the base layer is transmitted.

7) The function *ss_win_done* checks if the current window finished before its expiry timeout occured. If that's the case, then the window's timeout is cancelled. If there's any adaptation windows left in the *xmit_wins* queue, then *ss_win_done* calls *ss_child_send_win_start* to initiate transmission of the next adaptation window in the queue. However, if the current window turns out to be the last one in the stream, then an end of stream message is sent downstream by calling the function *ss_child_eof*.

8) Upon receiving a window start message from *ss_child_send_win_start*, StreamPlay's function *sp_parent_recv_win_start* is dispatched. This indicates that a new adaptation window is about to arrive. Apart from the first adaptation window, this is also the normal indication that the previous adaptation window is done transmitting, and that it should be committed to decode and display. In this case, the function *sp_win_xmit_done* will be called after this step. A new instance of PlayPpsWindow is created for the new adaptation window to arrive, and it's initialized according to the contents of the window start message, such as the number of SDUs expected and the time value for transmission end, so that StreamPlay can compute an expected deadline for this new window. After the window start message, SDUs for this new window will start to arrive.

9) The function *sp_parent_recv_sdu* is responsible for receiving the incoming SDUs. A counter in the PlayPpsWindow object tracks the number of SDUs which have arrived. The purpose is to check whether the base layer of the window is complete or whether all of its SDUs have arrived. The ADUs contained within the SDU are entered into a heap, which has the effect of sorting all of the ADUs for the window back to their original time-order. If the current window receive all the SDUs before its expiration deadline, then *sp_parent_recv_sdu* calls the function *sp_win_xmit_done*. However, if the current window expires before being able to receive all the SDUs, then *sp_parent_recv_sdu* calls the function *sp_win_decode_start* as soon as the base layer is received.

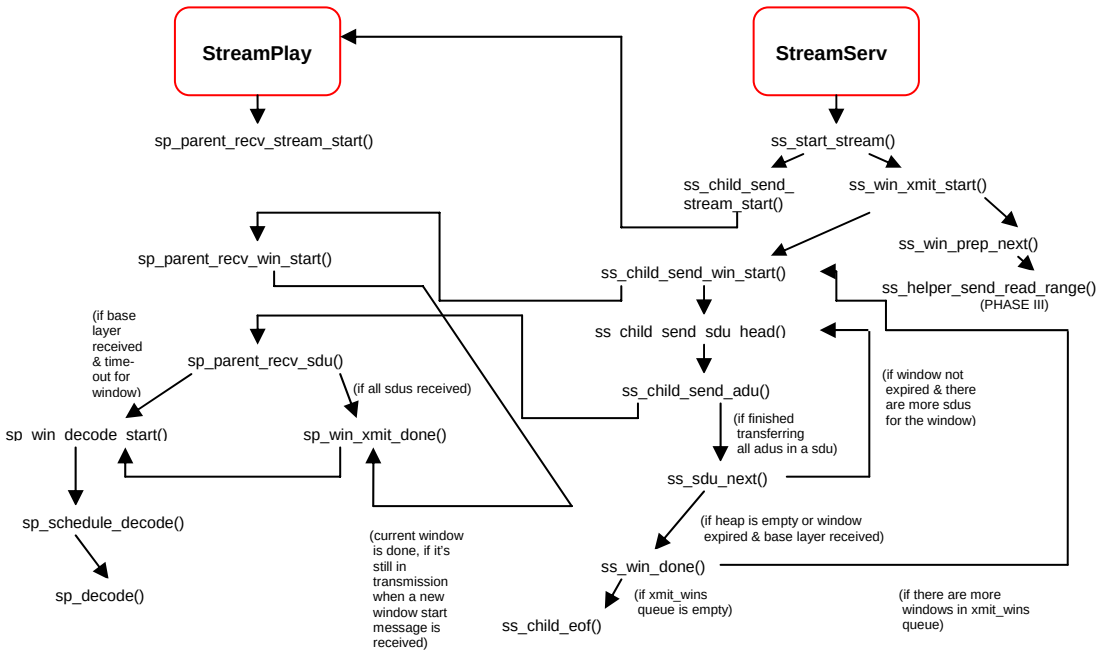


Figure 18: Phase IV

10) The function *sp_win_xmit_done* is called whenever a transmission window needs to be retired. It's either because the window has finished transmitting, or the window has reached its expiration deadline. In the first case, the function cancels the window's expiry timeout. The function also checks whether there were late SDUs in this window. If so, a phase adjust message is sent to StreamServ, which tells StreamServ how much it should adjust its clock in order to avoid late SDUs in future windows. An alternative way to do this is to adjust the display clock. The function *sp_win_decode_start* is then called by *sp_win_xmit_done*.

11) The function *sp_win_decode_start* is the starting point for decoding and displaying of the retired transmission window. It further calls the function *sp_schedule_decode*, that leads to the *sp_decode* function which is responsible for the decoding process. This process will not be further discussed, as it's not relevant to the thesis.

It seems reasonable to apply a quality-adaptation algorithm each time an SDU of an adaptation window is transmitted. According to the condition of the network, a consideration is made to either transmit the next SDU of the window or to skip the rest of it. This indicates that step 6 is one important insertion point for the improvement code. Step 4 is extended with a call of an initiation function which is explained in section 4.3.8 of chapter 4.

Step 5 needs to be extended in order to register the number of bytes that is written to TCP. This is necessary for a bandwidth prediction to work as section 3.2.2 shows, and section 4.1 of chapter 4 presents the variable *bytecount* in the ServPpsSession object that takes care of this registration. Since the function *ss_child_send_adu* goes through an iteration to transmit the next ADU of the same SDU, the previous ADU is considered successfully written to TCP whenever a new ADU is to be transmitted. Thus, the number of bytes for the previous ADU is added to the variable. It should be noted that the byte amounts of the SDU header parts, transferred in the function *ss_child_send_sdu_head*, are also added to the same variable.

The function *ss_win_done* mentioned in step 7 also needs to be extended with a little condition that is triggered whenever there is a need to ensure that the next adaptation window starts its transmission phase as soon as possible. A function is then called to handle this case, which is further explained in section 4.3.7 of chapter 4.

Section 4.4 of chapter 4 gives a detailed overview and illustration of all these changes.

3.1.7 Work Conserving Strategy

The work conserving strategy can be claimed to be a fundamental assumption for developing the improvement code. It was implemented to support quality-adaptive streaming over TCP. TCP's congesion control causes abrupt transmission rate changes, as discussed in section 2.1.2 of chapter 2, which might impede efficient streaming. This strategy helps smoothing the rate change out, by employing buffering at the client application.

If the transmission of an adaptation window finishes earlier than its estimated deadline, then this strategy makes it possible for StreamServ to advance immediately to the next adaptation window, discarding that window's original deadline for transmission start. However, in a non work conserving strategy this is not the case. StreamServ instead waits for the start deadline of the next window, which leads to a temporarily network transmission pause.

Qstream handles the work conserving strategy by adjusting the start deadline of an adaptation window according to a time variable called *workahead_limit*. This variable is part of the ServPpsSession object.

If StreamServ manages to finish the transmission of a window at time T_1 when the actual deadline of that window is T_2, where $T_1 < T_2$, then the difference $T_2 - T_1$ forms the workahead time T_3. This computed workahead time T_3 is then subtracted from the start deadline of the next adaptation window. As a result the next window starts its transmission phase T_3 time earlier than the original start deadline.

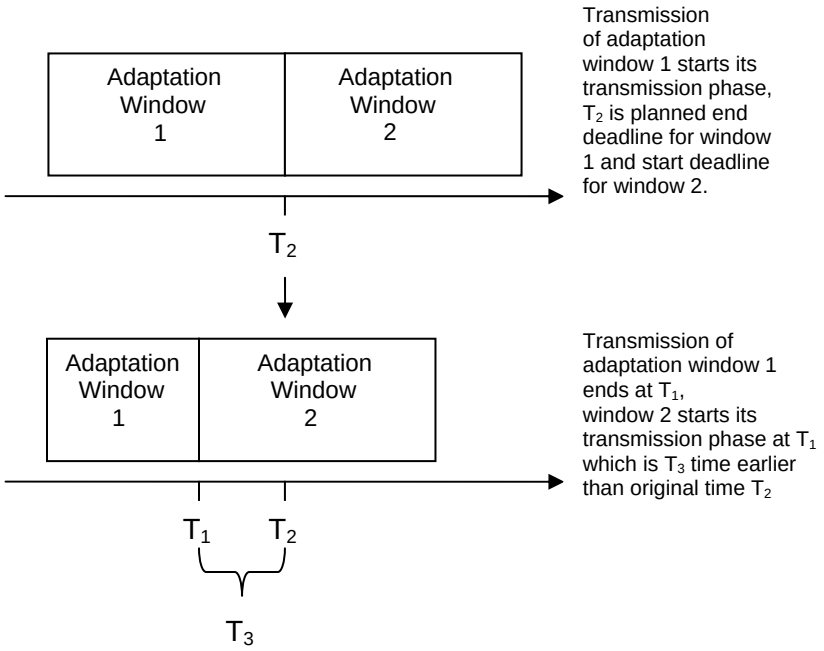


Figure 19: Ilustration of Work Conservation

The transmission timeline then advances faster than the real-time rate of the video. As a consequence, StreamPlay receives some video data earlier than it needs them. As mentioned in section 1.3 of chapter 1, this is good in the sense that StreamPlay can store these data in its buffer/queue, as a protection against future rate reductions caused by either TCP or other network failures.

This strategy is an essential part of the implementation of the improvement code, because it makes it possible for StreamPlay to buffer future data as a precaution against sudden loss of network connectivity. The usefulness of this strategy is further elaborated in section 3.2.

3.1.8 Window Scaling

The PPS algorithm includes an option called window scaling, which means that the size of adaptation windows is adjustable during a streaming process. The window starts out minimal to minimize the startup latency. Then the window durations grow with each new adaptation window as the stream plays. The adaptation windows grow by a rate of *growth_rate*, which is a variable that is specified in the ServPpsSession object. There are two other variables in addition, which are the time variables *expand_end* and *shrink_start*. The *expand_end* variable determines when the windows are to stop growing, which means that the window duration has reached the maximum allowed size. The next adaptation windows are initiated with this maximum size until the *shrink_start* time is reached. The *shrink_start* variable indicates that the next consecutive windows are to shrink according to a rate of 1 / *growth_rate*. Figure 20 provides an illustration of the window scaling.

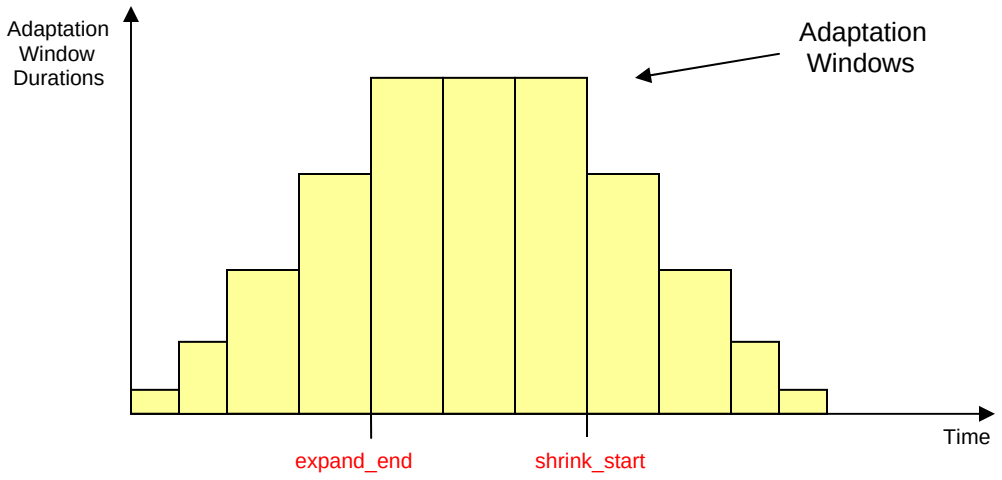


Figure 20: Window Scaling

Latency is a general problem that often occurs in a streaming session. All streaming algorithms buffer some data, which in turn add some latency to the overall end-to-end latency between the sender and the receiver. In the case of PPS, shorter adaptation windows will reduce its contribution to end-to-end latency. In PPS, the duration of the adaptation windows plays a significant role in achieving consistent video quality. It is a goal to adapt the video quality with least noticeable effect to the user, and this is done by making fewer quality changes, which in the case of PPS can be achieved by increasing the duration of the adaptation windows. Thus, window scaling provides a way to yield a balance between latency and consistency.

In this thesis, the improvement code is based on adaptation windows, whose size are of fixed duration. Section 3.2.2 of chapter 3 describes that bandwidth predictions are necessary for the improvement code to work. These predictions are based on bandwidth measurements of a past condition of the network, in which a prediction is done once for each new adaptation window. Based on these predictions, StreamServ is aware of the condition of the network at all time. To make sure the information of the network condition is up to date, the predictions should be made as regularly as possible. Thus, it's reasonable to use a small, fixed size of adaptation window for this purpose. Another reason for using fixed size adaptation windows, is because of the usage of a low-pass filter for bandwidth measurement, which is described in section 3.2.4.

When the window scaling option is turned off, the adaptation windows have a fixed duration of 2 seconds. That is, each adaptation window has a maximum transmission time of 2 seconds in the original implementation of Qstream.

3.2 Part II - Ways of Improvement

This part of the analysis identifies the improvements that can be made, in order for Qstream to be able to handle a streaming session over a wireless network with varying bandwidth. The focus is put on the parts of the code that need closer analyzing, and a proposal of an improvement code is discussed and analyzed.

3.2.1 A Way to Confront the Wireless Network

As discussed in section 1.3 of chapter 1, the main goal of this thesis is to improve the code of Qstream in a way that it can handle a streaming session gracefully over a wireless network with bandwidth that varies intensely over a longer period.

The wireless environment can be quite noisy and unreliable at times. Thus, it might lead to major changes in the available bandwidth over longer time intervals. This will unfortunately affect the performance of the streaming system.

A critical point are the expiration deadlines of adaptation windows. If the network bandwidth falls below a useful level over a longer period of time, the transmission of an adaptation window will eventually not be able to catch up with its scheduled transmission time. The current implementation of Qstream assumes a continuously stable and good network bandwidth. For this reason, StreamServ always sends all the SDUs of the different video layers, of an adaptation window, to StreamPlay without any further restrictions. This is acceptable if the network bandwidth never drops to an unusable level at any time.

In the case of a bandwidth drop, a timeout can occur for an adaptation window if its transmission phase begins too late. The solution to this problem of the current Qstream is to only send the base layer SDUs of the window and then move on to the next window in the transmission queue. Another way of getting a timeout is when a window exceeds its transmission time. The unsent SDUs for the remaining video layers are dropped, and the next window in the queue begins its transmission phase.

In a network with an unpredictable and varying bandwidth, the algorithm described above is not appropriate for achieving a smooth and satisfying playback of a streaming video over a longer time-scale. If the bandwidth varies intensely over a longer period of time, the quality of the streaming video is also changing in a similar intense way. In the worst case, only the base layer of the adaptation windows is transmitted, and some windows can even be dropped completely due to timeout. This is unacceptable, as thoroughly bad or highly varying video quality is not pleasant for the human eye. A test is performed in chapter 5 to verify this.

The work conserving strategy introduced in the first part of this analysis is appropriate for solving the problem regarding wireless network streaming. Since this strategy allows

StreamServ to work ahead of time, it is possible for StreamServ to transmit more data to StreamPlay than required, at a moment in which the network bandwidth is good enough. StreamPlay stores these future data in its buffer. Thus, the buffering of extra data at StreamPlay makes up for the possible loss of data transmission which might occur when the network connectivity is getting bad. When the network connectivity is partially or completely lost, and StreamServ has some workahead time to spare, then the situation is not critical for the playout at StreamPlay, as long as the loss doesn't exceed the workahead time.

Although the work conserving strategy solves the issue of connectivity loss, there is need for an algorithm that knows how to make use of the workahead time in an efficient way. This is where the improvement code comes into the picture. The improvement code must assist StreamServ in computing how much workahead time that should be put aside in different circumstances in a fair manner. This depends on some kind of information about the condition of the network and how much data it can handle. A way to achieve this is by implementing an algorithm to predict the network bandwidth. This is elaborated and discussed in the next section.

3.2.2 Prediction of the Network Bandwidth

The predicted network bandwidth is an essential piece of information that StreamServ must have in order to estimate how much contents of the adaptation windows it should transmit. A real prediction of the future bandwidth is impossible, and thus the implemented prediction must be based on bandwidth measurements of the past condition of the network.

A reasonable way to measure the bandwidth is by computing the number of bytes written to TCP per second by StreamServ, for every pre-defined time interval with start and end points that are fixed. The chosen time interval for this computation is the duration of the transmission of an adaptation window. For each new adaptation window to be transmitted, StreamServ compares the values of the last two intervals to predict if the network bandwidth is going to be good or bad. That way a decrease in value indicates a prediction in which the network bandwidth is falling. When the value eventually increases again, it indicates a prediction in which the bandwidth is increasing. If the network bandwidth is stable and good over a longer period of time, then the number of bytes written per second for each time interval should be about the same. The byte calculation is possible because StreamServ transmits the video parts by writing the bytestream to the downstream TCP socket using the system call write(). Upon successful completion, this function returns the number of bytes sent.

3.2.3 The Trade-Off between Quality and Workahead

With the bandwidth information in place, the next challenge is to figure out how StreamServ should transmit the adaptation windows according to this information.

One thing to notice is that if less SDUs of a window are transmitted, then more workahead time is gained. The minimum SDUs that have to be transmitted per window, are those corresponding to the base layer of a window. If the bandwidth prediction indicates that the network connectivity is getting bad, then StreamServ must be able to estimate how many video layers it can afford to transmit for a particular window to achieve a certain level of acceptable quality, and drop the rest to avoid unnecessary delays. The most it can drop are all the enhancement video layers for that window. However, if the bandwidth prediction indicates that the network connectivity is good, then StreamServ has two choices. Either StreamServ should transmit more video layers for the coming windows to increase the video quality, or it should transmit a small amount and gain some workahead time as precaution against possible network failures instead. In other words, there is a trade-off between video quality and workahead time.

The more workahead time StreamServ saves up, the better is the protection against playback failures at StreamPlay which is caused by unstable network condition. The drawback is that the video quality of the playback might be quite bad over a longer period of time, since most of the enhancement layers for a number of adaptation windows have been dropped. On the other hand, if StreamServ chooses to transmit more layers for each window and put less workahead time aside, then the video quality of the playback is better, assuming the transmission is successful. However, the drawback in this case is the workahead time saved up, which might be too small to cover a longer period of bad connectivity. This could result in a sudden, drastic loss of quality while playing the video, or in the worst case the playback stops, due to late arrival of SDUs.

The challenge lies in the task of finding a balance point for the trade-off between video quality and workahead time. This should depend on how StreamServ interprets the predicted bandwidth information. In general, StreamServ should treat a prediction of bad connectivity more seriously than a prediction of good connectivity, as precaution is better than taking risks. A prediction of good connectivity, which turns out to be true, doesn't necessarily mean that the bandwidth of the wireless network stays good for a longer period. Although this assumption is important, precaution must not dominate completely at the expense of video quality (throughout the streaming session) either.

An idea could be to take precaution against connectivity loss by transmitting the least amount of video layers possible, until a certain amount of workahead time is reached, and then proceed to increase video layers to transmit for the next adaptation windows.

It's also favourable that the video quality doesn't change too rapidly whenever the prediction indicates either good or bad connectivity, as fewer quality changes is generally desirable from a viewer's perspective. This means that the increasing/decreasing of video layers should be done in a slow manner.

3.2.4 Proposal of Improvement Code

This section introduces the functions that ought to be implemented on the server side, to solve the issues discussed in the previous sections of this chapter.

- **Bandwidth prediction** - As discussed in section 3.2.2, prediction of the network bandwidth is necessary. There is need for a function that can make a calculation to determine the prediction. This function is called once for each new adaptation window that is entering the transmission phase. The calculation is based on two values, which is the number of bytes written to TCP per second for the two latest time intervals. In other words, these two values are bandwidth samples computed at the two latest time intervals.

  An appropriate way to interpret a prediction is by computing and returning the result as a percentage of either gain or loss. If x is the value of the latest bandwidth sample, and y is the value of the previous bandwidth sample, then there's two possible outcomes:

  1.) **Loss** - The x value is smaller than the y value. That means the percentage of loss is ((y - x) / y) * 100.
  2.) **Gain** - The x value is larger than the y value. That means the percentage of gain is ((x - y) / y) * 100.

  However, further investigation on the computations above shows that the achieved loss and gain values are not quite trustworthy when viewed over a short time scale. The problem is that the amount of time used to transmit the number of bytes for consecutive time intervals is not uniform, when the samples are computed based on the short pre-defined time intervals. This leads to inaccurate single bandwidth samples, and thus, the consequence is unreliable loss or gain values computed from bandwidth samples of the two latest consecutive time intervals. To solve this issue, a low-pass filter is taken into account.

$$b_k = \frac{2\tau - \Delta k}{2\tau + \Delta k} b_{k-1} + \Delta k \frac{x + y}{2\tau + \Delta k} \quad [14] \qquad \text{(a)}$$

  This filter is employed to average sampled measurements and to obtain the low-frequency components of the available bandwidth. Thus, single measurements are less significant, and the weight is put on the development over a longer period of time.

  The value b_k is the filtered measurement of the available bandwidth, while b_{k-1} is the previous measurement. Δk is the time between the bandwidth samples x and y. In other words, Δk corresponds to the time in which the total number of bytes is written for the latest time interval, which corresponds to the previous transmitted adaptation window. In order to satisfy the Nyquist-Shannon sampling Theorem,

Δk must be less than $\tau/2$ [14]. τ must not be less than 2 * window duration. Since the window duration is 2 seconds, the chosen value of τ is 5. An extra second is added as a precaution against delays that might be caused by sudden bandwidth drops. If it happens that $\Delta k > \tau/2$, then Δk is set to $\tau/2$ and x is set to 0 for the computation of b_k.

- **Written bytes update** – This function has the task of updating the two variables used by the 'bandwidth prediction' function to compute the predicted percentage value. These two variables correspond to the number of bytes written to TCP per second for the the last two time intervals, in which the start and end points of the intervals are fixed. These are the bandwidth samples x and y mentioned above in the description of the 'bandwidth prediction' function. Since the 'bandwidth prediction' function is called each time a new adaptation window is entering its transmission phase, this function should be called to update the two values at the end of the transmission phase of each window.

- **Video layer update** - Since the bandwidth of a wireless network could be highly varying, StreamServ should make sure that the workahead time saved up is above a minimum required amount at all time, if possible, for emergency cases. When that minimum amount of workahead time is in place, StreamServ is allowed to transmit additional enhancement layers for the next adaptation windows to increase the video quality. However, there must be a function that controls the amount of extra layers StreamServ is allowed to transmit for the next windows. Even if the network bandwidth is predicted to increase, the enhancement layers should be added in a slow manner, as a sudden major change in quality is not desirable. Another reason is that the bandwidth is not reliable, even if it seems to be stable at the moment. If StreamServ starts transmitting too much, it might affect the overall performance if connectivity suddenly is lost. An idea could be to increase the number of allowed enhancement layers by a fixed small value for each consecutive window, as long as the bandwidth is predicted to be usable. The function should also be able to decrease the number of enhancement layers in a similar slow manner, if the bandwidth prediction indicates bad network connectivity. The function sets a global variable with a value that equals the number of allowed enhancement layers, which can be used by the rest of the StreamServ program. This value is set on the basis of the predicted bandwidth information. This global variable is called *layer* and is further described in section 4.1 of chapter 4. The bandwidth information should be compared to some adjustable threshold variables, whose purpose it is to decide the degree of how good or bad the predicted bandwidth is. As discussed in section 3.2.3, precaution is better than taking risks. That means, a small degree of bad connectivity prediction should be taken more seriously than a large degree of good connectivity prediction. The 'video layer update' function should operate with this consideration in mind.

- **Workahead update** - A function to update the workahead time is necessary. This function is called each time StreamServ finishes transmitting an adaptation window before the window's originally estimated deadline, which happens when StreamServ decides to drop some or all of the enhancement layers of the window. This workahead time is then used to adjust the the next window's transmission start deadline.

- **Transmission crisis check** - It should be clear by now that each adaptation window has a pre-determined transmission time. This transmission time should not be exceeded, so each window has a deadline that makes sure the transmission stops by then. This ensures that StreamServ is able to start transmitting the next window. However, if workahead time has been put aside, then a window is allowed to use more time than its determined transmission time, if StreamServ can afford it. In a network in which the connectivity is unstable, there's a probability that StreamServ will not manage to transmit the entire contents of an adaptation window by the time the deadline of the window is reached. There is need for a function that is regularly called to determine if the current adaptation window in transmission has used all of its time yet. In that case, the function must return some kind of crisis message to inform about this.

- **The adaptation function** - This is the main function that takes care of the quality adaptation, according to how the condition of the wireless network is. This function is responsible for a task mentioned earlier, which is to check that a minimum amount of workahead time is constantly maintained, if possible, for emergency cases. This amount can be adjusted by implementing a threshold variable that is configurable. If the configured amount is not reached, then the function must initiate StreamServ to only transmit the minimum required SDUs of the coming adaptation windows, which corresponds to the base layers, so that the emergency amount of workahead time is collected as fast as possible. When the emergency workahead is reached, the function allows more layers of the next adaptation windows to be transmitted. The amount of enhancement layers allowed is found in the global variable set by the 'video layer update' function. For each SDU of a window that is transmitted, the 'transmission crisis' function is called to check if the current window has used all of its transmission time yet. If that's the case, then the adaptation function must estimate if StreamServ can afford to continue transmitting more contents of the window, based on the workahead time saved up. Again this can depend on an adjustable threshold variable, which is used to compare the workahead time with. This function ought to be called by *ss_sdu_next*, which corresponds to step 6 in phase IV of the streaming scenario described in the first part of the analyze. That is, the adaptation algorithm should make a consideration of what to do next, each time an SDU of a window is transmitted.

- **The workahead transmission function** – The purpose of this function is to make sure that the next adaptation window (if it is ready to be transmitted) is able to start its transmission phase immediately, if the adaptation function decides to stop

transmitting the current adaptation window earlier than its originally scheduled transmission end deadline. Recall from section 3.1 that the preparation and mapping of the next adaptation window (phase III) and the transmission of the current window (phase IV) are done concurrently. If the adaptation function decides to end the transmission phase for the current window earlier than its deadline and phase III for the next adaptation window has already finished at this point, then the next window will not begin its transmission phase until the scheduled time is reached. This scheduled time corresponds to the original transmission end deadline of the current window. Thus, there is need for a function that ensures that the next adaptation window is allowed to enter the transmission phase as soon as possible, if the current window's transmission is ended by the adaptation function before its original transmission end deadline.

- **The initiation function** – The purpose of this function is to initiate and regularly update a couple of significant variables. One important update is the information about the predicted bandwidth. This is done by calling the 'bandwidth prediction' function. Another piece of information that needs updating is the number of allowed enhancement layers for the coming window. This is done by calling the 'video layer update' function, which updates the global variable mentioned, based on the predicted bandwidth information. The variable is then used to regulate the number of allowed enhancement layers for the window to be transmitted. This function should be called by *ss_child_send_win_start*, which corresponds to step 4 in phase IV of the streaming scenario described in part one of the analyze. That is, the function is called each time a new adaptation window is about to start transmission.

Most of the functions described above make use of threshold variables, whose purpose is to help exploring and finding the balance point for the trade-off between video quality and workahead. These variables are further explained in chapter 4, and the tests in chapter 5 show that different values of certain threshold variables might lead to different outcomes.

Chapter 4: The Implementation

This chapter shows how the improvement code is implemented. The functions, discussed in section 3.2.4 of chapter 3, are coded using the C programming language, which is also the language that Qstream is written in. Section 4.1 describes the necessary additional variables that are implemented into existing Qstream data structures. Section 4.2 provides an overview of the threshold variables mentioned in section 3.2.4 of chapter 3. The descriptions of the functions in section 4.3 are in pseudo-code, which is based on the source code of the improvement. The purpose is to filter out the unnecessary details and preserve the most essential parts of the functions. The full source code is available in appendix A. Finally, the last section of this chapter provides an illustration of the implemented code, which shows how the implemented functions work together to adapt the quality of the streaming video with regard to the network condition.

4.1 Additions to the Data Structures

This section provides a description of additional fields that need to be implemented into the data structures of Qstream, which is necessary for the improvement code. The data structures that need to be updated are the ones that correspond to the ServPpsSession and the ServPpsWindow objects. These are shown in figure 21 and figure 22.

```
ServPpsSession {
   Time              global_tvnow              //Marks the current time
   Integer           layer                     //Indicate nr of enh. layers allowed to send
   Integer           layer_increase_allowed    //Indicate when layer increase  is allowed
   Integer           bytes_pr_sec_interval_x   //Indicate bytes pr sec for interval just finished
   Integer           bytes_pr_sec_interval_y   //Indicate bytes pr sec for last interval
   Time              time1                     //Mark start of time interval
   Time              time2                     //Mark end of time interval
   Time              timediff                  //Mark difference between time2 and time1
   Unsigned Integer  difftime                  //For converting timediff
   Float             time_used                 //For converting difftime to time in second in
                                                 decimal format
   size_t            bytecount                 //Stores number of bytes written to TCP
  GTimeVal           vid_ahead                 //Marks how far ahead the playback is
   Integer           stablecount               //Marks consecutive stable conditions
   Integer           b_k                       //Marks the current filtered bandwidth
   Integer           b_k1                      //Marks the previous filtered bandwidth
   Integer           tau                       //Time constant used by low-pass filter

 }
```

Figure 21: Updated PPS Session Object for StreamServ

Figure 21 shows the necessary variables that each PPS session has to maintain in addition. The variable *global_tvnow* is used to capture the current time whenever it's needed.

The fields *layer* and *layer_update_allowed* take care of the amount of enhancement layers that StreamServ is allowed to transmit for the coming adaptation windows. The variable *layer_increase_allowed* has the task of enabling and disabling the possibility of increasing the *layer* variable. Section 4.3.8 shows that this is necessary.

The fields *bytes_pr_sec_interval_x* and *bytes_pr_sec_interval_y* contain the number of bytes written to TCP per second for the last two time intervals. These two variables are also known as bandwidth samples x and y from section 3.2.4 of chapter 3, and they are used by the bandwidth prediction function to compute the prediction value.

The next fields are time variables used to compute the time intervals. The result, which is a decimal number representing the interval in seconds, is stored in the float variable *time_used*.

The field *bytecount* is used to store the number of bytes that is written to TCP. The usage of this field is shown in section 4.3.2. The *vid_ahead* field is used to store the amount of time that the playback is ahead of the transmission. The usefulness of this field is elaborated in section 4.3.4. The field *stablecount* is used to store the number of consecutive stable predicted network conditions. The details of this field is elaborated in section 4.3.3. The fields *b_k* and *b_k1* are used to store the filtered measurements of bandwidth, which are computed by using the low-pass filter as described in section 3.2.4 of chapter 3. The last field *tau* is the τ variable that is also used in the low-pass filter. As discussed, this variable has a value of 5.

```
ServPpsWindow {
    Time        win_start_xmit      //Marks the time when the window start xmit
    Integer     percentage_kind     //Indicate what the kind of percentage
    Integer     percentage          //Indicate the percentage of gain or loss,
                                      the bandwidht prediction has returned
}
```

Figure 22: Updated Adaptation Window Object for StreamServ

Figure 22 shows the additional fields that each ServPpsWindow object must maintain.
The field *win_start_xmit* is used to capture the time when the window starts its transmission phase. This is important for later use, when a computation of how long the window has been transmitting is needed.

The fields *percentage_kind* and *percentage* are updated by the bandwidth prediction function once for each new adaptation window. As the name indicates, the *percentage* variable stores the percentage of gain or loss, which is a result of the computation of the two variables *bytes_pr_sec _interval_x* and *bytes_pr_sec_interval_y* described above.
The *percentage_kind* field is simply there to tell if the percentage value indicates a gain or a loss. A value of 1 means gain, while a value of 0 means loss.

4.2 Threshold Variables

Section 3.2.4 of chapter 3 introduced the notion of threshold variables. Some of these might need to be adjusted in order to find a balance point for the trade-off between video quality and workahead. This section provides a more detailed description of these threshold variables.

- **low_workahead_thresh** – This threshold variable indicates the minimum amount of workahead time that StreamServ should maintain at all time, if possible, throughout a streaming session. If the workahead time is below this threshold, then StreamServ must transmit as little as possible of the coming adaptation windows in order to quickly build up the workahead until it reaches above the threshold. The fastest way to do this is by transmitting only the base layers of the windows. There are two situations in which the workahead time is below this threshold. The first involves the start-up of a streaming session, which means that StreamServ only transmits the base layers of the first adaptation windows until the workahead time passes the threshold. The second situation occurs during a streaming session. If the connectivity is bad over a longer period of time due to unstable network condition, the collected workahead time is consumed rapidly by StreamServ. When the connectivity is back, the workahead time might have fallen below the threshold. This triggers StreamServ to start transmitting only the base layers again until the necessary amount of workahead is collected.

- **ignore_bw_workahead_thresh** – As the name indicates, this threshold variable decides when StreamServ can ignore the bandwidth prediction. The reason for implementing this threshold variable, is because it's not always a benefit to decrease the video quality immediately when predicting a bad network condition. If the amount of workahead time saved up is small, then reducing the quality (by lowering the number of allowed enhancement layers to transmit) is a right thing to do. However, if there is a lot of workahead time to spare, then decreasing video quality might be an act of waste due to bad utilization of the workahead time. The idea is that if the collected workahead time surpasses this threshold, then StreamServ can ignore the bandwidth prediction and proceed to increase the number of enhancement layers for each consecutive adaptation window until the maximum number of layers is reached. In such a case, a longer period of bad network connectivity leads to a rapid reduction of the workahead time. If the workahead time falls below the threshold, then StreamServ must take the bandwidth prediction into consideration again and control the amount of enhancement layers to be transmitted. As this threshold variable can cause greedy consumption of the workahead time, it should initially be set to a high value.

- **crisis_workahead_thresh** – Recall that each adaptation window has a limited transmission time. If the network is under a bad condition, then StreamServ might not be able to finish transmitting the contents of an adaptation window by the time the window's transmission time has passed. However, with some workahead time

present, StreamServ can still afford transmitting the rest of the contents of the window. It follows that StreamServ must somehow know the maximum amount of workahead time that it can use for this purpose. It's not desirable to use all the workahead time on the late adaptation windows. That's why there is a need for a threshold variable that controls the amount. If the workahead time falls below this threshold, then StreamServ has to stop transmitting the contents of the current late adaptation window.

Whenever the bandwidth prediction gives a percentage of loss, this percentage value is compared to a couple of threshold variables to determine the condition of the network. There are three kinds of conditions that have to be taken into consideration. The first indicates a network condition that is 'stable'. This means that the number of bytes written to TCP per second for two consecutive time intervals are about the same, but with the latest interval having a slight smaller value than the prior interval. The second condition indicates a network condition that is also kind of 'stable', but bad from an overall viewpoint. In other words, the number of bytes written to TCP per second for two consecutive time intervals differs more, with the latest interval having an even smaller value than the prior interval. In this case, the decreasing of quality should be limited to a reduction of only a certain number of enhancement layers. The third is the 'bad' condition, in which the value of the latest interval is significantly smaller than the value of the prior interval. This condition is treated more seriously in the sense that the number of allowed enhancement layers is dropped by a few for each consecutive window until it reaches 0, if necessary. This could happen if the percentage of loss corresponds to this condition for several consecutive windows. To determine which amount of value difference between the intervals that corresponds to the different conditions, there is need for a couple of threshold variables. These are called *overall_stable_percentage_thresh* and *bad_percentage_thresh*.

- **overall_stable_percentage_thresh** – A percentage of loss below this threshold corresponds to a 'stable' condition, while a percentage above this threshold corresponds to either a 'stable, but overall bad' or 'bad' condition.

- **bad_percentage_thresh** – A percentage of loss below this threshold indicates a network condition that is either 'stable' or 'stable, but overall bad'. On the other hand, a percentage of loss above this threshold indicates a real bad network condition. This threshold variable is implemented to ensure that a slight bad prediction does not cause unnecessary and exaggerated decreasing of video quality. However, this variable should initially be set to a low value, as 'precaution is better than taking risks' when streaming over a network with unpredictable bandwidth.

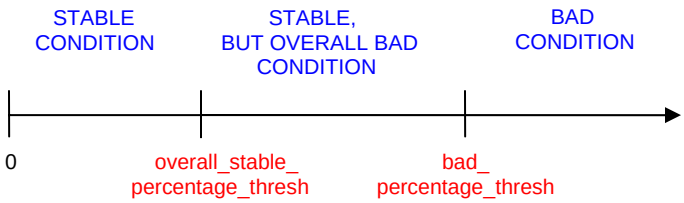


Figure 23: Bad Condition Threshold

- **bad_layer_thresh** – Since a drastic change in video quality is not desirable, only a few layers should be dropped each time a prediction of bad connectivity occurs. This threshold variable regulates the number of layers to be reduced for the case when the percentage of loss corresponds to a 'bad'network condition.

- **bad_stable_layer_thresh** – This threshold variable indicates the maximum allowed enhancement layers for the case when the percentage of loss corresponds to a 'stable, but overall bad' network condition. If the number of enhancement layers lies above this threshold, then it has to be reduced. On the other hand, if the number lies below this threshold, then it's allowed to increase with the threshold as a maximum.

If the network condition is stable, which means that the percentage of loss is below the threshold *overall_stable_percentage_thresh*, then the number of enhancement layers is regulated as if the condition is stable in the case of a percentage of gain. This 'stable' condition is described below.

Whenever the bandwidth prediction gives a percentage of gain, this percentage value is compared to a threshold variable to determine the condition of the network. There are two kinds of conditions that have to be taken into consideration. The first is a 'stable' condition. This means that the number of bytes written to TCP per second for two consecutive time intervals are about the same, but with the latest interval having a slight larger value than the prior interval. The second is a 'better' condition, in which the value difference between the two intervals is even larger. To determine which amount of value difference between the intervals that corresponds to the two different conditions, there is need for a threshold variable. This variable is called *better_percentage_thresh*.

- **better_percentage_thresh** – A percentage of gain below this threshold corresponds to a 'stable' condition, while a percentage above this threshold corresponds to a 'better' condition.

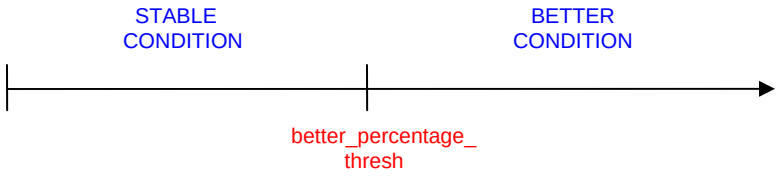


Figure 24: Good Condition Threshold

- **stable_layer_thresh** – This threshold variable is needed to regulate the number of allowed enhancement layers for the case when the percentage of gain corresponds to a 'stable' network condition.

- **stablecount_thresh** – Whenever the condition of the network is predicted to be stable (when either the percentage of loss is below the threshold *overall_ stable_ percentage_thresh* or the percentage of gain is below the threshold *better_ percentage_thresh*), it is counted. If the number of counts reaches this threshold, then StreamServ is triggered to increase the threshold *stable_layer_ thresh* by 1. In other words, StreamServ is allowed to transmit better video quality for the next adaptation window if the number of stable network predictions reaches this threshold. In order for the number of counts to reach this threshold, the condition of the network must be stable and good for a consecutive number of adaptation windows that equals this threshold. However, if the bandwidth starts dropping and leads to a prediction of bad network condition, then the threshold *stable_layer_ thresh* must also be reduced accordingly by 1 for the next adaptation window. The variable that takes care of this counting is described in section 4.3.3.

4.3 The Functions

This section presents the pseudo-code of the functions proposed and discussed in section 3.2.4 of chapter 3.

4.3.1 The Bandwidth Prediction Function

```
BANDWIDTH_PREDICTION(pps, win)
 1  pps.b_k1 = pps.bk
 2  if (pps.time_used > pps.tau / 2)
 3       pps.time_used = pps.tau / 2
 4       pps.bytes_pr_sec_interval_x = 0
 5
 6  pps.b_k = (2*pps.tau – pps.time_used) / (2*pps.tau + pps.time_used) * pps.b_k1 +
 7            pps.time_used * (pps.bytes_pr_sec_interval_x + pps.bytes_pr_sec_interval_y) /
 8            (2*pps.tau + pps.time_used)
 9
10  if (win.number != 0)
11       if (pps.b_k  < pps.b_k1)
12          win.percentage_kind = 0
13          win.percentage = ( (pps.b_k1 – pps.b_k) / b_k1 ) * 100
14       else if (pps.b_k  > pps.b_k1)
15          win.percentage_kind = 1
16          win.percentage = ( (pps.b_k – pps.b_k1) / pps.b_k1 ) * 100
17       else
18          win.percentage_kind = 1
19          win.percentage = 0
```

A bandwidth prediction is made once for each new adaptation window. There are two kinds of outcome that the function can predict. Either it's a prediction of bad connectivity (lines 11-13) or a prediction of good connectivity (lines 14-16). In both cases the function updates the two variables *percentage_kind* and *percentage* for the window. The variable *percentage_kind* indicates what kind of percentage is being dealed with and can have either the value 0 or 1. The *percentage* variable contains the percentage of either gain (good connectivity prediction, indicated by *percentage_kind* = 1) or loss (bad connectivity prediction, indicated by *percentage_kind* = 0), which is computed by using the variables *b_k* and *b_k1*. Recall from section 3.2.4 of chapter 3 that these variables are filtered bandwidth measurements derived from a low-pass filter. Lines 1-8 show how the low-pass filter updates the variables *b_k* and *b_k1*. Notice that Δk from section 3.2.4 is represented here by the variable *time_used* (lines 2, 3, 6, 7, 8). Lines 17-19 correspond to a condition in which the two variables *b_k* and *b_k1* have an equal value. In that case, the network condition should be considered to be stable and good.

4.3.2 The Written Bytes Update Function

```
UPDATE_WRITTEN_BYTES(pps)
1   get_current_time(pps.time2)
2   gint bytes_written = pps.bytecount
3   pps.bytecount = 0
4
5   pps.bytes_pr_sec_interval_y = pps.bytes_pr_sec_interval_x
6
7   pps.timediff = pps.time2 – pps.time1
8   pps.difftime = (pps.timediff.tv_sec * 1000000) + pps.timediff.tv_usec;
9   pps.time_used = (float) pps.difftime / 1000000
10
11  pps.bytes_pr_sec_interval_x = bytes_written / pps.time.used
```

This function is called at the end of each window's transmission phase. Its purpose is to update the two variables *bytes_pr_sec_interval_x* and *bytes_pr_sec_interval_y* for the bandwidth prediction function to use. Line 2 gets the value of the *bytecount* variable, which corresponds to the total number of bytes written to TCP since the last time the function was called. Line 3 resets the *bytecount* variable to 0 as preparation for the next interval. Lines 7-9 computes the time used to write this amount of bytes. The time is in seconds and is represented with a decimal value for accuracy purpose. Line 11 computes the number of bytes written per second and stores the result in the variable *bytes_pr_sec_interval_x*. The previous value of *bytes_pr_sec_interval_x* is stored in *bytes_pr_sec_interval_y*, which is done in line 5.

4.3.3 The Layer Update Function

```
UPDATE_LAYER(pps, win, nr)
1   if (nr = 0)
2       if (win.percentage_kind = 0 & win.percentage ≥ bad_percentage_thresh)
3             stable_layer_thresh - 1
4             if (pps.workahead_limit.tv_sec < ignore_bw_workahead_thresh)
5                 if  (pps.layer > bad_layer_thresh)  pps.layer – 2
6                 else pps.layer – 1
7           else
8                 if  (pps.layer < 15) pps.layer + 1
9
10      else if  (win.percentage_kind = 0 & win.percentage < bad_percentage_thresh
11              & win.percentage > overall_stable_percentage_thresh)
12          stable_layer_thresh - 1
13          if (pps.workahead_limit.tv_sec < ignore_bw_workahead_thresh)
14              if  (pps.layer < stable_bad_layer_thresh)  pps.layer + 1
15              else pps.layer – 1
```

```
16          else
17              if (pps.layer < 15) pps.layer + 1
18
19      else if (win.percentage_kind = 0 & win.percentage <= overall_stable_percentage_thresh)
20          pps.stablecount + 1
21          if (pps.workahead_limit.tv_sec < ignore_bw_workahead_thresh)
22              if (pps.stablecount < stablecount_thresh)
23                  if (pps.layer < stable_layer_thresh) pps.layer + 1
24                  else pps.layer – 1
25              else
26                  if (stable_layer_thresh < 15)
27                      stable_layer_thresh + 1
28                      pps.stablecount = 0
29          else
30              if (pps.layer < 15) pps.layer + 1
31
32      else if (win.percentage_kind = 1 & win.percentage < better_percentage_thresh)
33          pps.stablecount + 1
34          if (pps.workahead_limit.tv_sec < ignore_bw_workahead_thresh)
35              if (pps.stablecount < stablecount_thresh)
36                  if (pps.layer < stable_layer_thresh) pps.layer + 1
37                  else pps.layer – 1
38              else
39                  if (stable_layer_thresh < 15)
40                      stable_layer_thresh + 1
41                      pps.stablecount = 0
42          else
43              if (pps.layer < 15) pps.layer + 1
44
45      else if (win.percentage_kind = 1 & win.percentage ≥ better_percentage_thresh)
46          pps.stablecount + 1
47          if (pps.layer < 15) pps.layer + 1
48
49  else if (nr = 30)
50      if (pps.layer ≥ 3) pps.layer / 2
51      else pps.layer = 0
52
53  else if pps.layer = nr
```

This function is responsible for updating the amount of enhancement layers allowed to be transmitted for the coming adaptation windows. It operates according to the values of the variables *percentage_kind* and *percentage*, which are updated by the 'bandwidth prediction' function.

There are three main conditions that can be triggered by the variable *nr*, which is passed as an argument from the caller. These conditions are seen in lines 1, 49 and 53.

Line 1 is the first main condition. Lines 2-47 are sub-conditions to the one of line 1. If the bandwidth prediction gives a percentage of loss, then the update of the number of allowed enhancement layers are done by either lines 2-8, 10-17 or 19-30. Lines 2-8 are for the case when the percentage value is larger or equal to the threshold *bad_percentage_thresh*. The *layer* variable is then reduced by either 1 or 2 according to the threshold *bad_layer_thresh*. Lines 10-17 are for the case when the percentage value is smaller than the threshold *bad_percentage_thresh*. The *layer* variable is then regulated by the threshold *stable_bad_layer_thresh*. Lines 19-30 are for the case when the percentage value is smaller than the threshold *overall_stable_percentage_thresh*. In this case, the network bandwidth is assumed to be stable. The *layer* variable is regulated by the threshold *stable_layer_thresh*. In a similar manner, lines 32-47 handle the update of the allowed enhancement layers for the case when the prediction gives a percentage of gain. It's worth noticing that when the percentage of gain indicates a 'stable' network condition (line 32), the update of the *layer* variable (lines 33-43) is similar to the one described above, when the 'stable' condition is derived from a percentage of loss (lines 20-30).

Line 49 corresponds to the second main condition that is triggered whenever the workahead time is below the threshold *low_workahead_thresh*. This is caused by either a long-lasting network connectivity loss that leads to a consumption of the entire (or most of it) collected workahead time, or the start-up of a streaming session in which the workahead time is still being built up towards *low_workahead_thresh*. The number of allowed enhancement layers (*layer* variable) is halved (line 50) if the current number is larger or equal to 3. Since there has been a connectivity loss, it's better for StreamServ to be cautious with the transmission of the next window in the sense that less layers should be transmitted. Otherwise, the *layer* variable is set to 0 (line 51). Recall that StreamServ only transmits the base layer of the first adaptation windows until the workahead time passes the threshold *low_workahead _thresh*. That means the *layer* variable remains at the value of 0 until the workahead surpasses the threshold.

Line 53 takes care of the third main condition, which is triggered whenever the threshold *crisis_workahead_thresh* stops StreamServ from using any more of the workahead time to transmit the contents of a late adaptation window. The number of enhancement layers that StreamServ manages to transmit for the window so far, is passed as an argument to this function which is indicated by the variable *nr*. Line 53 sets the *layer* variable equal to *nr*. That way StreamServ is allowed to transmit up to at least *nr* layers for the next adaptation window. This is to ensure that a drastic change in quality does not occur in the transition from the late window to the next window in the transmission queue.

It should also be noted that the bandwidth prediction is considered as long as the workahead time is smaller than the threshold *ignore_bw_workahead_thresh* (lines 4, 13 21, 34), as discussed in section 4.2.

In addition, the *stablecount* variable is increased by 1 each time the predicted network condition is good (lines 20, 33, 46). That is, the value of the variable is increased whenever the condition is either 'stable' or 'better'. If the value of this variable reaches

the threshold *stablecount_thresh* (lines 22 and 35) and the network condition is still stable, then the threshold *stable_layer_thresh* for the coming window is allowed to increase by 1. The *stablecount* variable is then reset back to 0. However, if the bandwidth eventually starts dropping and leads to a bad connectivity prediction, then the threshold *stable_ layer_ thresh* is reduced by 1 (lines 3 and 12). It might be clear by now that the purpose of the *stablecount* variable is to ensure that better video quality is eventually achieved if the network condition is continuously stable. This way the amount of allowed enhancement layers is not completely controlled by a fixed value of the threshold *stable_layer_thresh*, if the condition of the network remains stable.

4.3.4 The Workahead Update Function

UPDATE_WORKAHEAD(pps, win)

1. pps.vid_ahead = win.xmit_deadline – pps.global_tvnow

This is a simple function that updates how much time the playback currently is ahead of the transmission, by computing how much earlier the current adaptation window finishes its transmission phase than it is supposed to. Thus, this function is called whenever StreamServ stops transmitting an adaptation window earlier than its scheduled transmission end deadline. If necessary, this amount of time is used to configure the transmission start deadline of the the next adaptation window, as section 4.3.7 shows.

4.3.5 The Transmission Crisis Check Function

CHECK_TRANSMISSION_CRISIS(pps, win)

```
1  if ( (pps.global_tvnow – win.win_xmit_start) > (win.xmit_end – win.xmit_start) )
2     return 0
3  else if ( (pps.global_tvnow – win.win_xmit_start) < (win.xmit_end – win.xmit_start) )
4     return 1
```

This function has the task of checking if the current adaptation window in transmission has used all of its transmission time yet. The time that the window has used up to this moment, is computed by subtracting the time when the window started transmitting (win_xmit_start) from the current time. This is compared to the window's limited transmission time, which is computing by subtracting the window's transmission start time (xmit_start) from the window's transmission end time (xmit_end). If the window has used all its transmission time, then the condition of line 1 is triggered, and the value 0 is returned. Otherwise, the transmission is still within schedule. The condition of line 3 is triggered, which returns the value 1.

4.3.6 The Adaptation Function

```
ADAPTATION(pps, win, sdu)
 1  if (pps.workahead_limit < low_workahead_thresh)
 2      if (sdu.priority < (15 – pps.layer) )
 3          UPDATE_WORKAHEAD(pps, win)
 4          SS_WIN_DONE(pps, win)
 5  else
 6      no_crisis = CHECK_TRANSMISSION_CRISIS (pps, win)
 7
 8      if (no_crisis = 1)
 9          if (sdu.priority < (15 – pps.layer) )
10          UPDATE_WORKAHEAD(pps, win)
11          SS_WIN_DONE(pps, win)
12      else
13          UPDATE_WORKAHEAD(pps, win)
14          if (pps.workahead_limit > crisis_workahead_thresh)
15              if (sdu.priority < (15 – pps.layer) )
16                  UPDATE_WORKAHEAD(pps, win)
17                  SS_WIN_DONE(pps, win)
18          else
19              if (sdu.priority < 15)
20                  UPDATE_LAYER (pps, win, 15 – sdu.priority)
21                  UPDATE_WORKAHEAD(pps, win)
22                  SS_WIN_DONE(pps, win)
```

This function is responsible for the quality adaptation. For each SDU that is transmitted, it is called to determine if the number of layers transmitted has surpassed the amount of allowed layers, which is regulated by the *layer* variable (lines 2, 9, 15). If that's the case, then the rest of the contents of the adaptation window are dropped. However, if the number of layers transmitted has not reached the allowed threshold yet, then the function must estimate if StreamServ can afford sending the next SDU of this window, based on the condition of the network.

Since Qstream operates with 16 layers in total, each adaptation window contains SDUs with priorities in a range from 15 to 0. The SDUs with priority 15 correspond to the base layer, while those with priorities 14 to 0 correspond to the enhancement layers. This explains the way lines 2, 9, 15 and 19 are coded. In line 2 for instance, the *layer* variable is set to 0 (reason explained in section 4.3.4). The if-condition is supposed to check if StreamServ has finished transmitting the base layer of the current adaptation window. This is done by checking if the priority of the transmitted SDU is smaller than 15-0 = 15. As mentioned, SDUs with priorities less than 15 belong to enhancement layers.

Lines 1-4 make sure that StreamServ only transmits the base layer of the adaptation window, as long as the workahead time is below the threshold *low_workahead_thresh*.

Otherwise, lines 5-22 are in charge of adapting the quality. Line 6 calls the 'transmission crisis check' function to determine if the current adaptation window has used all of its transmission time yet. If the returned value is 1, then the condition of line 8 is triggered. StreamServ is then allowed to transmit the next SDU, which belongs to an enhancement layer, provided that its priority is not exceeding the threshold set by the *layer* variable. When all the allowed SDUs are transmitted, line 10 updates the workahead time, and line 11 calls ss_win_done to end the transmission phase of the window.

In the case of a transmission time crisis, the returned value from the 'transmission crisis check' function is 0, which triggers the condition of line 12. Line 13 updates the workahead time. If the workahead time is larger than the threshold *crisis_ workahead_ thresh*, then StreamServ can afford transmitting the next SDU of the current late adaptation window (lines 14-17). On the other hand, if the workahead time is below *crisis_workahead_thresh*, then there is no time left for StreamServ to transmit the contents of the late window, provided that at least the base layer SDUs are done (line 19). Line 20 calls the 'layer update' function and passes the number of layers transmitted so far for this late window as an argument (reason explained in section 4.3.4). Lines 21 and 22 update the workahead time and call ss_win_done to end the transmission phase of the late window.

4.3.7 The Workahead Transmission Function

```
WORKAHEAD_START_NEXT_WIN(pps)
1.  ServWindow win = g_queue_peek_head(pps->mapped_wins)
2.
3.  if (win == NULL)
4.      pps.workahead_limit = pps.vid_ahead
5.  else
6.      g_aio_cancel_timeout(win->start_timeout)
7.      ss_win_xmit_start(&tvnow, win)
```

This function is called each time the adaptation function ends the transmission phase of an adaptation window earlier than its scheduled transmission end deadline. Its purpose is to ensure that the next adaptation window is allowed to start the transmission phase immediately, if the window is finished prepared and ready to be transmitted.

Line 1 attempts to retrieve the next adaptation window that is finished prepared and mapped (phase III, section 3.1.5 of chapter 3) from the *mapped_wins* queue. If there is no window that is finished yet, then the condition of lines 3-4 is triggered. This condition makes sure that when the next adaptation window is ready to be transmitted (when it finishes phase III), it will know that it can start the transmission phase earlier than planned. The procedure of how to compute an earlier start time of the transmission phase is described in step 5 of phase III in section 3.1.5 of chapter 3. For this procedure to work, the variable *workahead_limit* of the ServPpsSession object needs to be updated with the time of the *vid_ahead* variable (line 4).

However, if there is an adaptation window in the *mapped_wins* queue, which means that the next window has finished its preparation and mapping phase, then the condition of lines 5-7 is triggered. In this case, the preparation of the next adaptation window finished before the adaptation function decided to end the transmission of the current window. This means that the next adaptation window was scheduled with a transmission start deadline that corresponded to the originally scheduled transmission end deadline of the current window. Thus, there is need for an update that makes it possible for the next adaptation window to start its transmission phase right away. Line 6 cancels the original transmission start deadline of the next window. Line 7 calls the function *ss_win_xmit _start* to start the transmission phase for this adaptation window.

4.3.8 The Initiation Function

```
INITIATION(pps, win)
 1  if (win.number = 0)
 2       pps.workahead_limit = 0
 3       pps.layer_increase_allowed = 0
 4       pps.bytes_pr_sec_interval_x = 0
 5       pps.bytes_pr_sec_interval_y = 0
 6       win.percentage_kind = 0
 7       win.percentage = 0
 8       pps.layer = 0
 9
10  BANDWIDTH_PREDICTION(pps, win)
11
12  if (pps.workahead_limit ≥ low_workahead_thresh)
13       pps.layer_increase_allowed = 1
14  else if (pps.workahead_limit < low_workahead_thresh & win.number ≠ 0)
15       pps.layer_increase_allowed = 0
16       UPDATE_LAYER(pps, win, 30)
17
18  if (win.number ≠ 0 & pps.layer_increase_allowed = 1)
19      UPDATE_LAYER(pps, win, 0)
20
21  get_current_time(win.win_xmit_start)
22  pps.time1 = win.win_xmit_start
```

This function is called each time a new adaptation window is entering its transmission phase. The purpose of this function is to initiate and update the necessary variables for later use. Lines 2-8 initiate certain variables of the first adaptation window. Lines 3 and 9 set the variables *layer_increase_allowed* and *layer* to 0. This indicates that StreamServ is only allowed to send the base layer for the first adaptation window. Line 10 calls the 'bandwidth prediction' function, which makes a prediction with an outcome that is relevant for all adaptation window except the first one. The assumption is that the network bandwidth should be good enough for StreamServ to transmit the base layer of the first window.

The variable *layer_increase_allowed* is set to 1 when the workahead time has reached the threshold *low_workahead_thresh* (lines 12-13), which means that StreamServ is allowed to transmit more than the base layer for the current new adaptation window. This triggers the condition of line 18, in which the 'layer update' function is called to update the number of allowed enhancement layers (*layer* variable) for this window.

Line 21 simply marks the current time as the transmission start time of the window and stores it in the variable *win_xmit_start*, which is a helper variable for the 'transmission crisis check' function. Line 22 stores this time value in another variable called *time1*, which is relevant for the 'written bytes update' function.

Also worth noticing is the condition of line 14, which is triggered whenever the workahead time falls below the threshold *low_workahead_thresh*. This is usually caused by a long period of network connectivity loss. Line 15 sets the *layer_increase_allowed* variable back to 0 as to indicate that increasing the *layer* variable is not allowed right now, due to the low workhead time and possible network failure. Line 16 calls the 'layer update' function with the number 30 passed as an argument (reason explained in section 4.3.4).

4.4 An Illustration of the Improvement Code

This section provides an illustration of how the functions described in section 4.3 co-operate to form the quality-adaptive algorithm needed to handle a streaming session over a wireless network with varying bandwidth. Figure 25 shows phase IV of the streaming scenario, which is described in section 3.1.6 of chapter 3. The two parts marked by 1 and 2 in the circles indicate the insertion points of the improvement code. Sections 4.4.1 and 4.4.2 present the codes of these two parts.

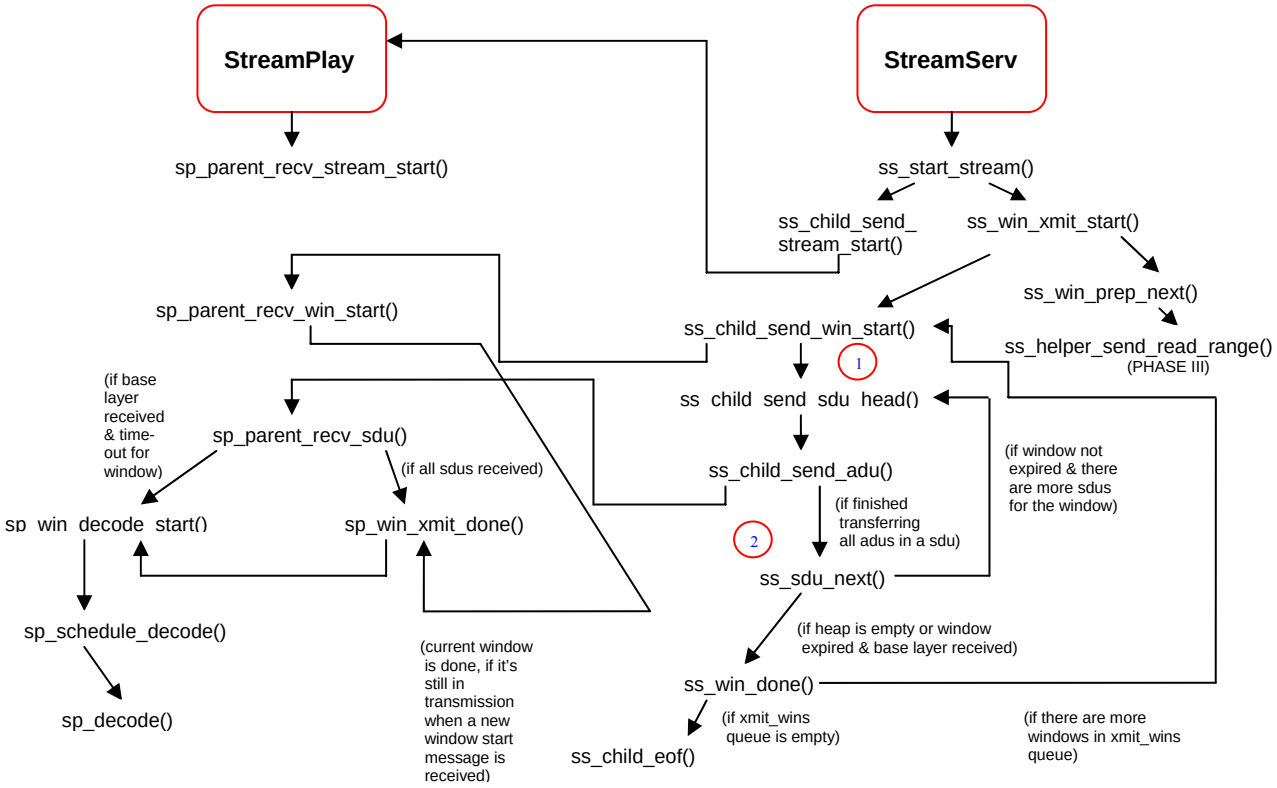


Figure 25: Phase IV of the Streaming Scenario with Marked Areas that Indicate the Insertion Points of the Improvement Code

4.4.1 Part I – Initiation

This part of the improvement code takes care of the initiation and updating of a couple of important variables. It is depicted in figure 26.

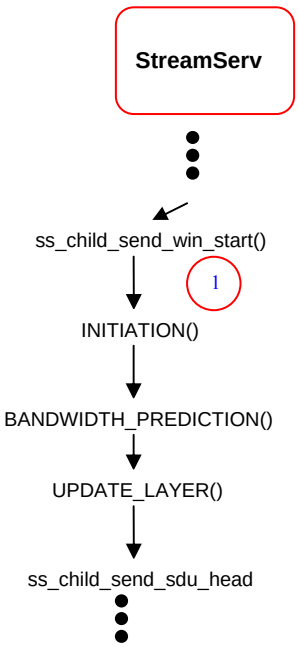


Figure 26: Implemented Code I: Initiation

1) In phase IV of the streaming scenario described in section 3.1.6 of chapter 3, the function *ss_child_send_win_start* is called to start the transmission of a new adaptation window. This function will call the implemented *INITIATION* function (4.3.8).

2) The *INITIATION* function initiates all the necessary variables. Among them are two essential ones that correspond to the predicted bandwidth information (*percentage_kind*, *percentage*) and the number of allowed enhancement layers (*layer*). These variables are updated by calling the implemented *BANDWIDTH_ PREDICTION* and *UPDATE_LAYER* functions.

4.4.2 Part II – Quality Adaptation

This part of the improvement code contains the improved quality-adaptation algorithm. According to how the network condition is, which is estimated with the help of the bandwidth prediction function, this algorithm determines the most efficient way to stream a multimedia file (SPEG file) over the wireless network, with regard to video quality and workahead/buffering.

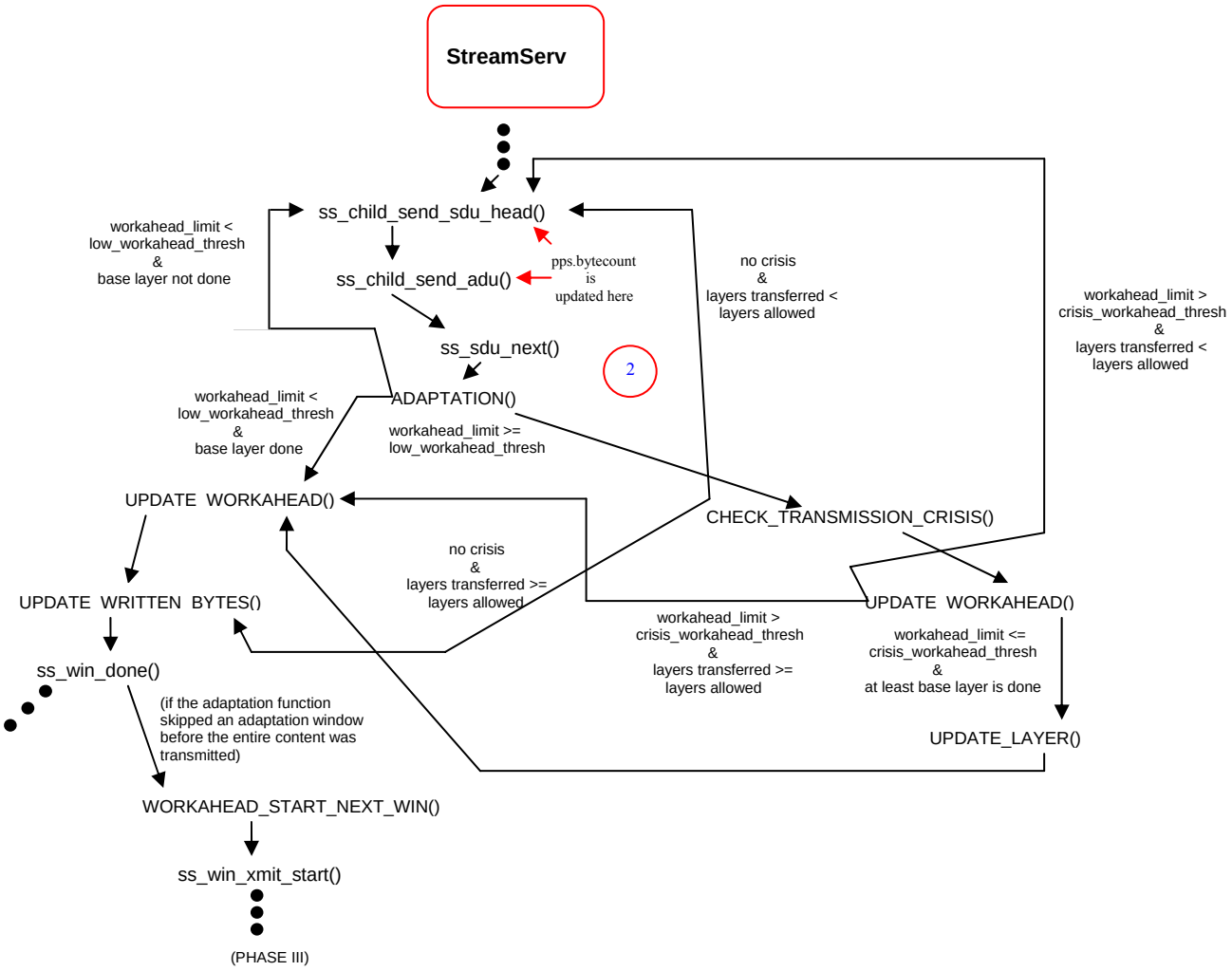


Figure 27: Implemented Code II: Quality-Adaptation Algorithm

1) The function *ss_sdu_next* is called each time an SDU is finished transmitting. The purpose is to check if there are still any SDUs left in the adaptation window currently in transmission phase. This is where the implemented *ADAPTATION* function comes into the picture. If the heap of SDUs is not empty, then StreamServ considers if it can afford transmitting another SDU of the adaptation

window, based on the predicted network condition. Therefore, it is reasonable that *ss_sdu_next* calls the adaptation function.

2) For the bandwidth prediction to work, the variable *bytecount* of the ServPps-Session object needs to be updated whenever a number of bytes is successfully written to TCP. This update is done in the functions *ss_child_send _sdu_head* and *ss_child_send_adu*, which was explained at the end of section 3.1.6 of chapter 3.

3) The procedure of the ADAPTATION function is provided in section 4.3.6 and will not be repeated. This is depicted in figure 27.

Chapter 5: Testing the Improvement Code

This chapter introduces a number of test cases which are designed to investigate how efficient the improvement code is. Section 5.1 provides an overview of the bandwidth scenarios that are used in the test cases. These scenarios are emulations of various conditions that could occur in a wireless network. Section 5.2 describes two test cases that are based on the original implementation of Qstream. The purpose is to verify that this original implementation is not quite suitable for streaming over a wireless network with intensely varying bandwidth (section 3.2.1 of chapter 3). Section 5.3 provides an insight into the improved outcomes when using the quality-adaptive algorithm of the improvement code. Section 5.4 introduces an objective quality metric that is required to make an objective assessment of the improvement code. Section 5.5 reviews a number of test cases that are performed on the improvement code. In order to make an objective assessment of the improvement code, the outcome of these test cases are evaluated by using the objective quality metric. Finally, section 5.6 provides an evaluation of the overall performance of improvement code.

These simplified names for the threshold variables (section 4.2 of chapter 4) are introduced for the purpose of easy reference in the further discussions of the test cases:

- low_workahead_thresh – **l_wt**
- ignore_bw_workahead_thresh – **ig_wt**
- crisis_workahead_thresh – **c_wt**
- overall_stable_percentage_thresh – **os_pt**
- bad_percentage_thresh – **b_pt**
- better_percentage_thresh – **be_pt**
- bad_layer_thresh – **ba_lt**
- bad_stable_layer_thresh – **bs_lt**
- stable_layer_thresh – **s_lt**
- stablecount_thresh **- st**

The initial values of the variables are as follow:

- **l_wt** = 5
- **ig_wt** = 40
- **c_wt** = 10
- **os_pt** = 2
- **b_pt** = 10
- **b_pt** = 40
- **ba_lt** = 8
- **bs_lt** = 5
- **s_lt** = 5
- **st** = 5
-

For all the subsequent tests that make use of the threshold variables, the initial values of these variables are used, unless stated otherwise.

5.1 Bandwidth Scenarios

As discussed in section 2.2.2 of chapter 2, a simulated wireless network is required for this thesis. With the help of the Token Bucket Algorithm (TBF), the varying network bandwidth can be emulated by adjusting the token rate at various times during a streaming session. Simple scripts are written to control the rate changes, and a sample of such a script can be seen in appendix B. The scripts give the following bandwidth scenarios that are used to test the quality-adaptation algorithm of the implemented code.

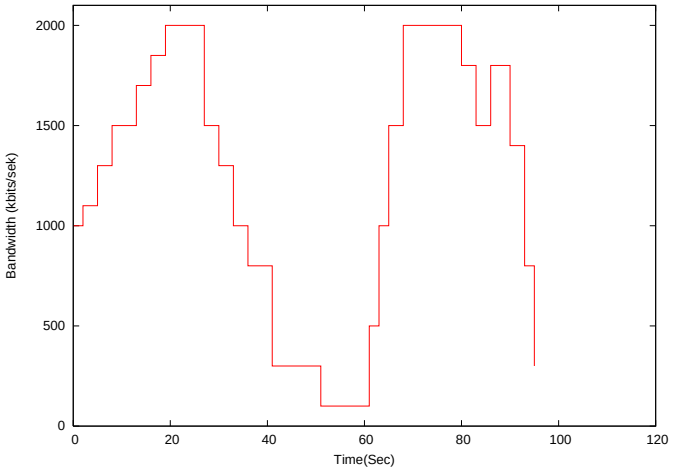


Figure 28: Bandwidth Scenario 1

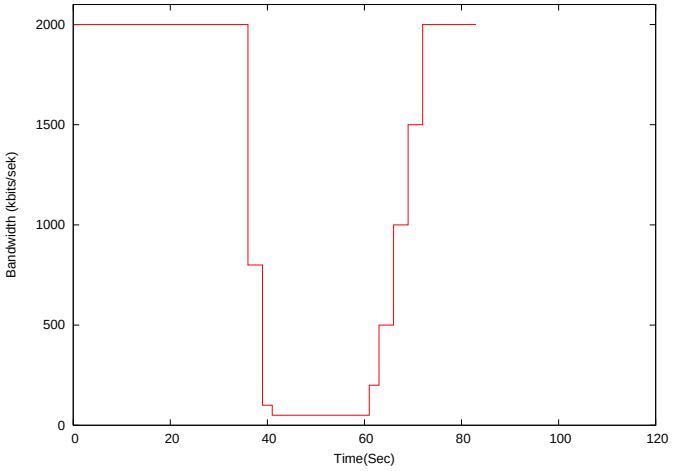


Figure 29: Bandwidth Scenario 2

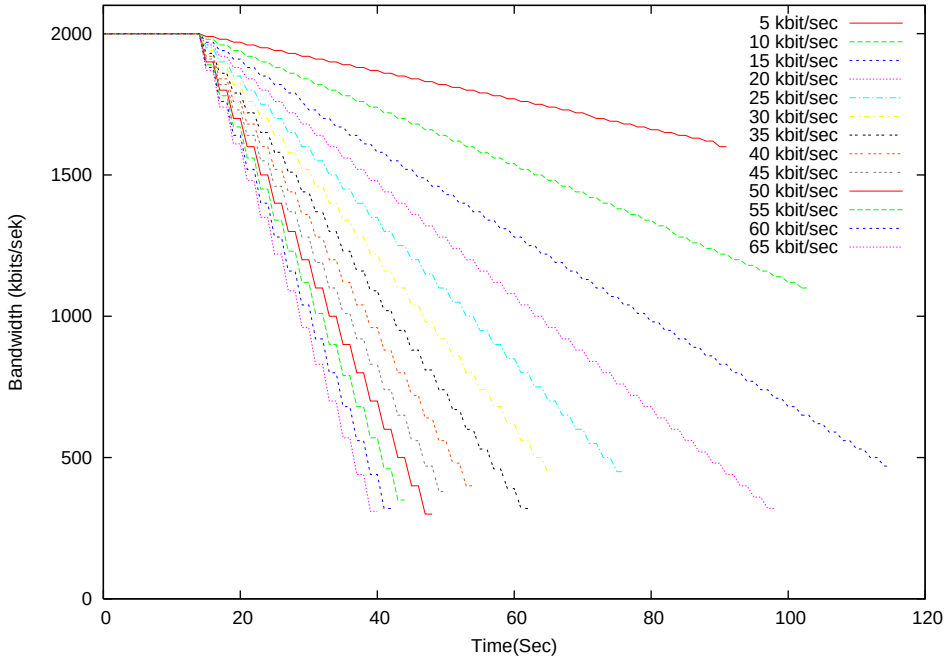


Figure 30: Bandwidth Scenario 3

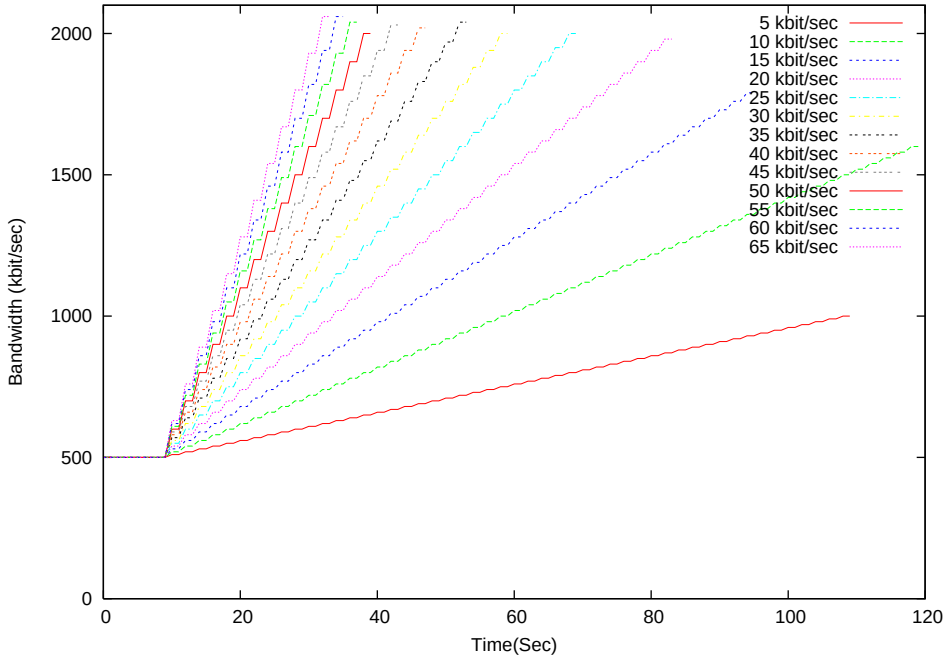


Figure 31: Bandwidth Scenario 4

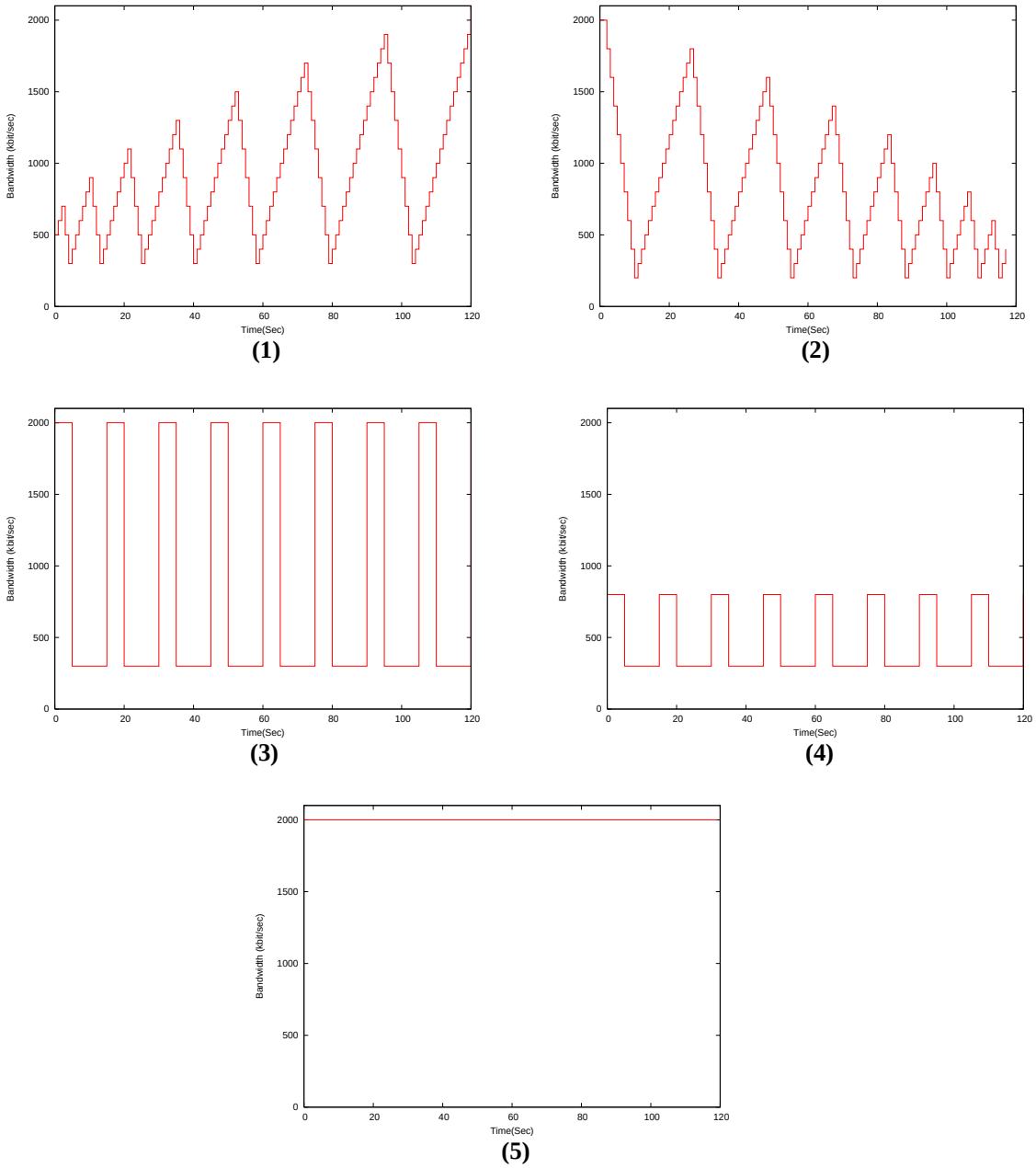


Figure 32 – Bandwidth Scenario 5

5.2 Using the Original Qstream Code

As discussed in section 3.2.1 of chapter 3, the original implementation of Qstream can't handle a streaming session over a network with varying bandwidth efficiently enough. Since the original implementation assumes a good and stable network bandwidth all the way along, StreamServ attempts to achieve top quality streaming by transmitting all the video layers for each adaptation window.

The two cases of this test show that when the original streaming system of Qstream is applied over a network with unstable condition, the achieved video quality is quite varying and unsatisfactory.

Items in use for this test:

- **Media Content:** An MPEG-1 file of about 2 minutes running time converted to SPEG
- **Bandwidth Scenario:** 1, 2

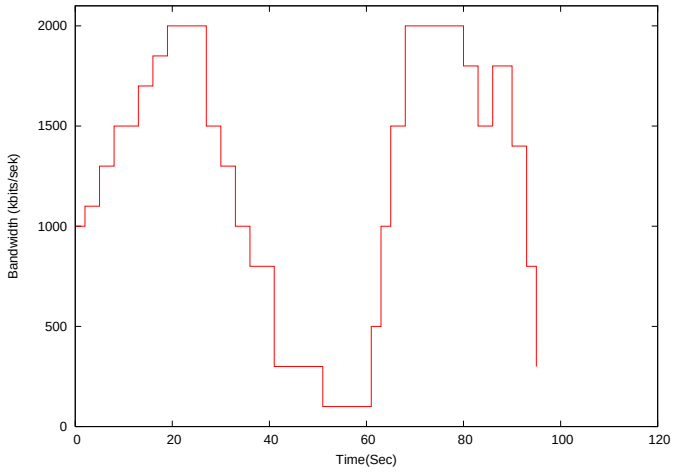


Figure 33: Bandwidth Scenario 1

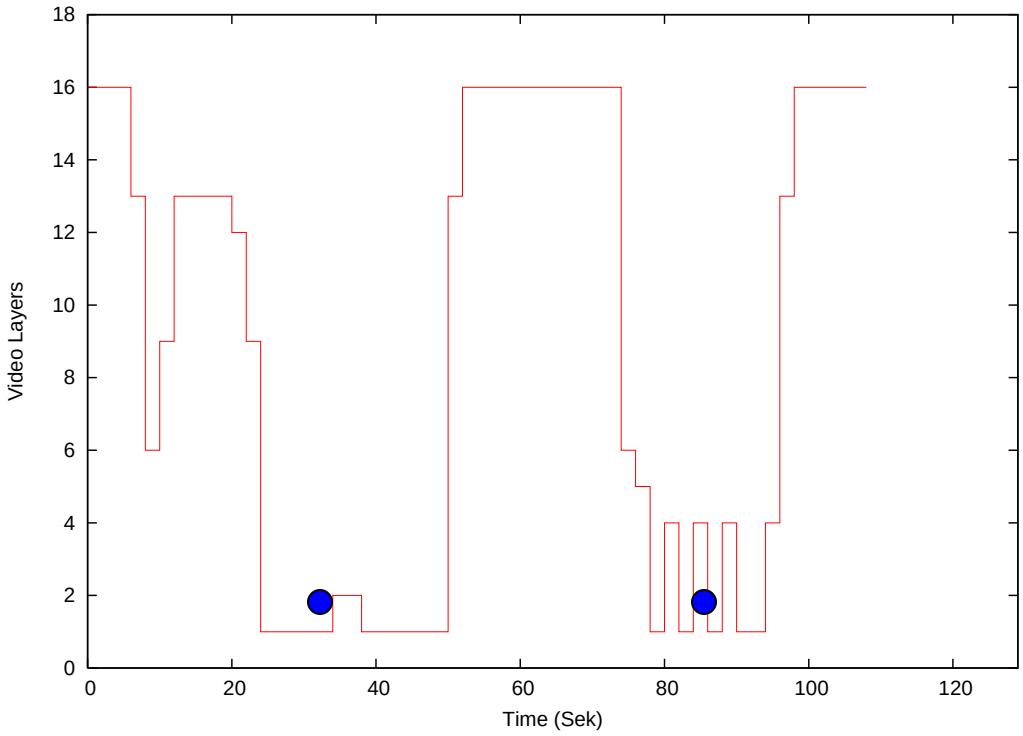


Figure 34: Outcome of Streaming Test, Case I

Figure 33 shows the bandwidth scenario used for the first case. When streaming over a network with this kind of bandwidth variation, the original streaming system of Qstream produces an outcome as seen in figure 34.

In figure 34, the horizontal axis indicates the running time of the streaming video, while the vertical axis corresponds to the quality level (number of video layers received) reached at StreamPlay at different times. As figure 34 shows, the quality change is quite drastic at places (marked with round dots). This happens because StreamServ's attempt to transmit top quality video, at all time, is limited by the unstable network condition.

When the bandwidth drops to an unusable level, it affects the transmission of the window at that moment. This leads to late SDUs of the window being dropped, and as a consequence the video content of this particular window has relatively bad quality. However, if the bandwidth suddenly increases when the next window starts its transmission phase, then StreamServ immediately continues to aim at top quality transmission. This is the reason why the graph of figure 34 is quite jumpy at the two marked areas. From a general viewpoint, this is unacceptable because the quality changes of the streaming video are too abrupt.

An extremely bad network connectivity also heightens the risk of getting timeouts for windows, as the next case shows.

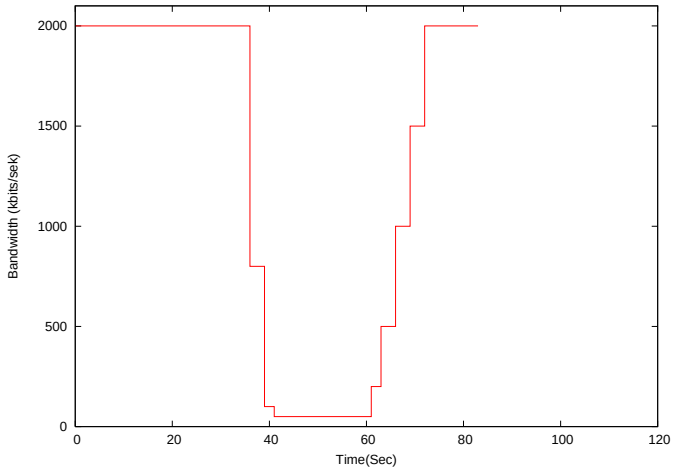


Figure 35: Bandwidth Scenario 2

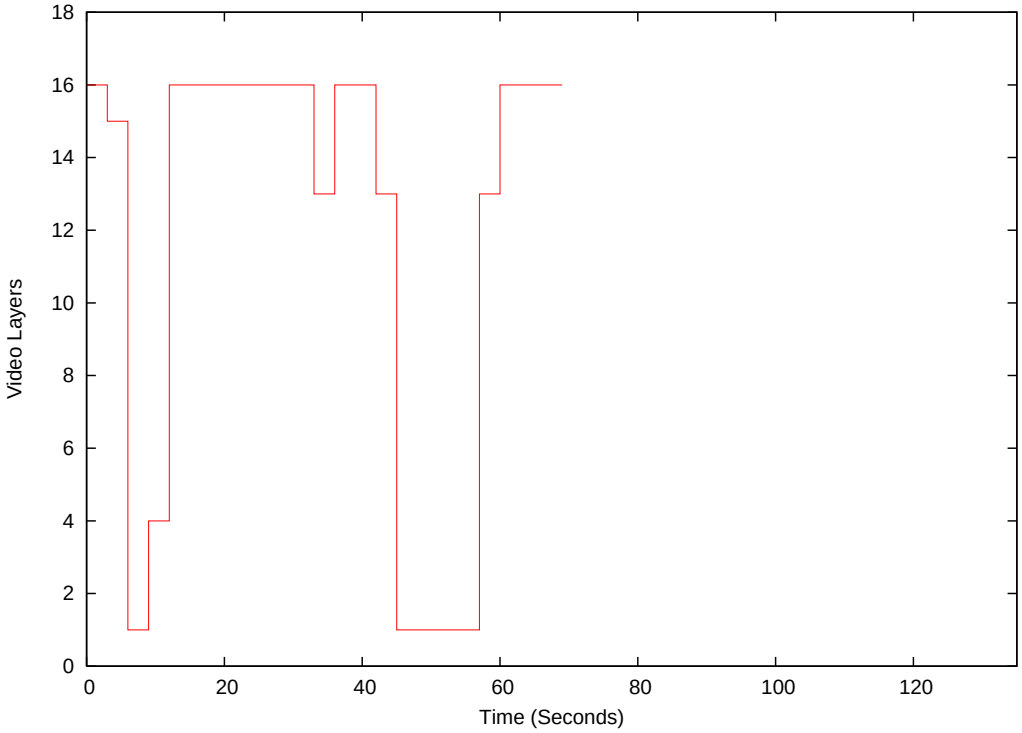


Figure 36: Outcome of Streaming Test, Case II

Figure 35 shows the bandwidth scenario used for the second case. When streaming over a network with this kind of bandwidth variation, the original implementation of Qstream produces an outcome as seen in the graph of figure 36.

As the graph shows, the running time of the streaming video is only about 60 seconds. Since the SPEG file that is used for this test is about 2 minutes of running time, it means that about 60 seconds of the video is missing. This is caused by timeout of several adaptation windows due to the extreme loss of connectivity seen in the graph of figure

35. In addition to this, the video layers received for the streaming video are quite uneven and result in major quality changes.

The conclusion to be drawn from these two test cases is that the original implementation of Qstream is not quite suitable for streaming over a network with highly varying bandwidth. Even though the system is equipped with the work conserving strategy which allows StreamServ to start transmitting the next adaptation windows a fixed time earlier, this still doesn't solve all the issues. It can prevent timeout of future adaptation windows from happening by transmitting faster than playback speed whenever it's possible. However, if the network bandwidth is highly varying and the duration of bad connectivity is long, then the quality changes will still follow an edgy pattern if StreamServ can't keep up with its transmission schedule.

5.3 The Addition of the Improvement Code

In the two following test cases, the streaming system with the additional improvement code is used. Bandwidth scenario 1 from the previous test is re-used for the streaming session, as the purpose is to show that the improvement code contributes in giving a better outcome compared to the one seen in figure 34.

Items in use for this test:

- **Media Content:** An MPEG-1 file of about 2 minutes running time converted to SPEG
- **Bandwidth Scenario:** 1
- **Threshold Variable Values:** The initial values of the variables are used in the first case. The second case of the test makes use of the following different values of *ig_wt*: 15, 30 and 40.

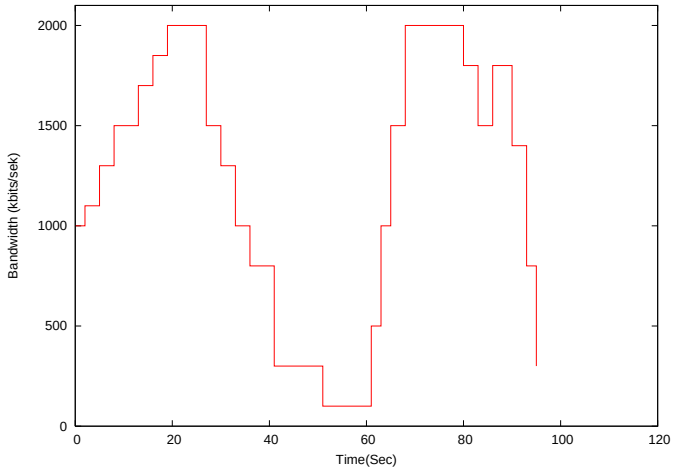


Figure 37: Bandwidth Scenario 1

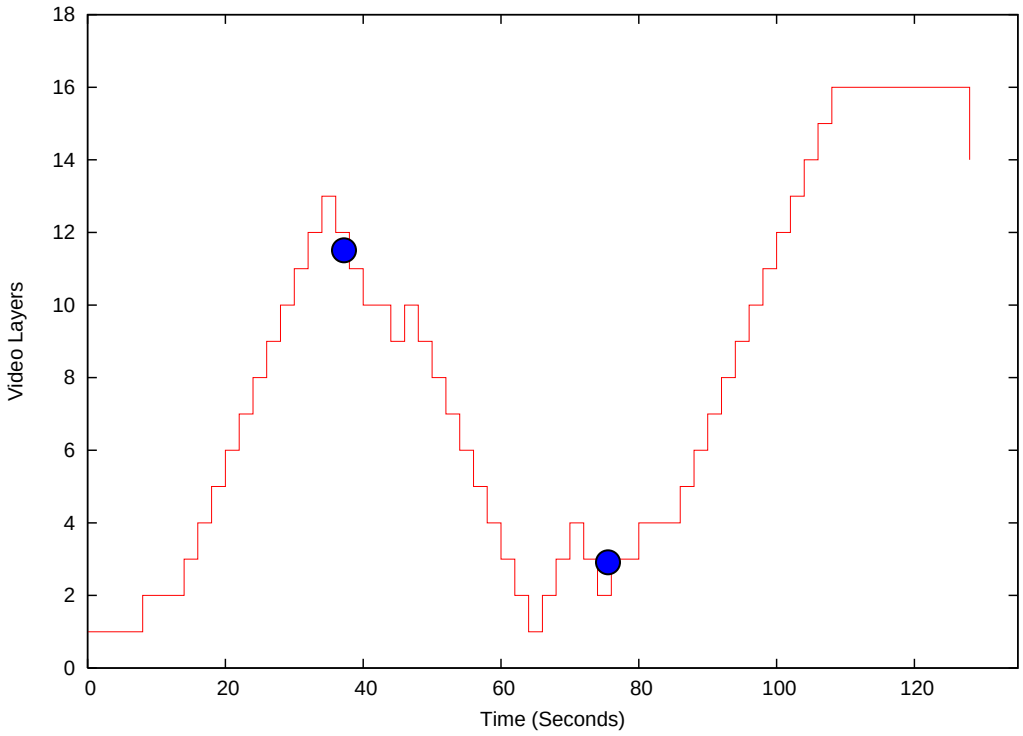


Figure 38: Outcome of Streaming Test, Case I: Streaming with Initial Values of the Threshold Variables

The graph of figure 38 shows an outcome that is quite improved compared to the one achieved from the first case of the test in section 5.2.

For the first few seconds, the graph shows that the streaming video has the lowest possible quality. This is because the *l_wt* variable is set to 5 seconds, which means that only the base video layers are transmitted for the first couples of adaptation windows. When the collected workahead amount reaches this threshold, the number of allowed enhancement video layers starts increasing in a slow manner for each consecutive adaptation window, which means that the quality of the streaming video is getting better.

The first dot in the graph indicates the corresponding moment of bandwidth scenario 1 in which the bandwidth starts dropping heavily (at about 8 seconds in figure 37). Since StreamServ has put some workahead time aside, by transmitting small amounts of video layers of the earlier windows, the streaming session is not facing an immediate critical situation at this moment even though the network connectivity is getting bad. By utilizing the workahead time, StreamServ can afford to decrease the number of allowed enhancement video layers for each consecutive window in a slow manner, as long as the predicted bandwidth is bad and there is enough workahead time left. The slow increase/decrease of allowed enhancement layers is desirable, because it prevents the video quality from changing in an abrupt way, as seen in the two test cases of section 5.2. In other words, the workahead time is aiding by giving the implemented quality-adaptive algorithm some time to smooth out heavy rate changes of the video, which are caused by partially or completely lost network connectivity.

The second dot in the graph indicates the corresponding moment of bandwidth scenario 1 in which the bandwidth starts increasing again (at about 55 seconds in figure 37). Since the bandwidth is increasing, the bandwidth prediction should indicate good network connectivity for the next windows. This triggers StreamServ to transmit more enhancement layers for the consecutive adaptation windows. Although the predictions indicate good connectivity, the implemented quality-adaptive algorithm only allows an increase of a few extra enhancement layers for each window, which means that the video quality of the streaming video is getting slowly better. As mentioned, the purpose is to prevent an instantaneous, high change in the video quality. Another reason for the slow increase is to maintain the precaution against a possible future loss of network connectivity.

With the implemented quality-adaptive algorithm present, clearly the abrupt changes of the video quality are minimized when streaming over a wireless network with highly varying bandwidth. The quality changes are instead evenly distributed over the streaming session, providing a less noticeable quality adaptation on the video.

Recall that the threshold variables were implemented with the intention of finding a balance point for the trade-off between video quality and workahead. If some of these variables are adjusted, then the quality of video is also likely to change in a different way as the next test case will show.

In the second case of the test, it will be shown that by adjusting the *ig_wt* variable, the achieved video quality is significantly changed when streaming over a network with the varying bandwidth of scenario 1. Notice that in the first case of the test, the *ig_wt* variable is set to the initial value 40. This means that StreamServ is allowed to ignore a bad connectivity prediction and increase the layers until top video quality is achieved, only when the collected workahead time is longer than 40 seconds. As mentioned, this is a precaution against sudden network connectivity loss whose duration might be long.

The larger the value of *ig_wt* is, the more it ensures that StreamServ has enough workahead time to handle a connectivity loss well, in the sense that the video quality is gradually reduced rather an extreme rapid change towards bad quality. Unfortunately, if the duration of bad connectivity is long, then this is done at the expense of the overall achieved quality of the streaming video. However, if a risk is taken by lowering the value of *ig_wt*, then parts of the streaming video might reach the highest quality level. The lower the *ig_wt* value is, the more greedy is the consumption of the workahead time to provide higher video quality during a period of bad connectivity. Thus, it heightens the risk of getting abrupt video quality degradation when the connectivity is bad over a longer period, because not enough workahead is available to maintain a slow decrease of quality.

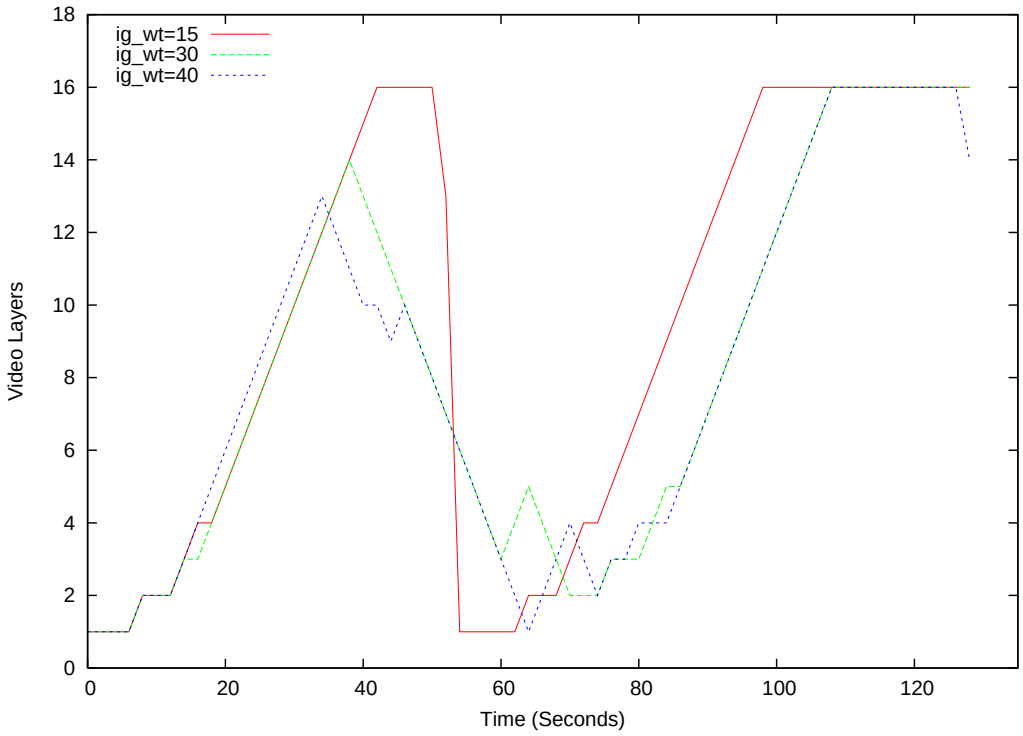


Figure 39: Outcome of Streaming Test, Case II: Streaming with Different Values of the Threshold Variable ig_wt

The graphs of figure 39 represent the outcome of three streaming sessions based on three different values of the *ig_wt* variable. With a value of 15 for the *ig_wt* variable, the quality of the streaming video reaches the top at about 40 seconds into the video and lasts for about 10 seconds. Then a drastic loss in quality occurs, and within a couple of seconds the video quality reaches the lowest quality level. The larger the value of *ig_wt* gets, the less are the occurences of such drastic quality changes, as seen in the streaming sessions where the value of *ig_wt* is 30 or 40. This is good, as less noticeable quality adaptation is preferable. But clearly a disadvantage with larger *ig_wt* values is that top video quality is seldom reached, unless the network connectivity is continuously good over a longer period.

The above test case is mainly to show that by adjusting a threshold variable, the outcome of the streaming might be different. However, to verify how efficient the implemented algorithm is, an objective assessment is required. There is need for an objective metric that can be used to present the objective quality of the streaming video. This is covered in the next section.

5.4 An Objective Metric to Represent the Perceived Quality

To make an assessment of the performance of the implemented quality-adaptive algorithm when streaming over a unreliable network, there is need for an objective metric that can be used to represent the perceived quality of layer-encoded video.

The PSNR is a popular metric to present the objective quality of video data. It is described by the following mathematical expression:

$$\text{PSNR} = 10 \log_{10} \frac{255^2}{MSE}$$ where MSE is the mean square error. [15] (b)

However, this metric does not represent the perceived quality of layer-encoded video well enough [4], so further details about it will not be provided in this thesis.

The lack of a metric to represent the perceived quality of layer-encoded video led to a new metric that was developed by Michael Zink (*currently a postdoctoral fellow in the Computer Science Department at the University of Massachusetts in Amherst*) for his dissertation about scalable Internet Video-on-Demand systems [4]. This metric is called the **spectrum**.

The objective quality metric spectrum is described by the following mathematical expression:

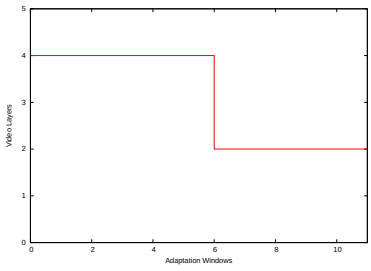
$$s(v) = \sum_{t=1}^{T} z_t \left(h_t - \frac{1}{\sum_{i=1}^{T} z_i} \left(\sum_{j=1}^{T} z_j h_j \right) \right)^2 \qquad \text{(c)}$$

The variables h_t and z_t are defined as:

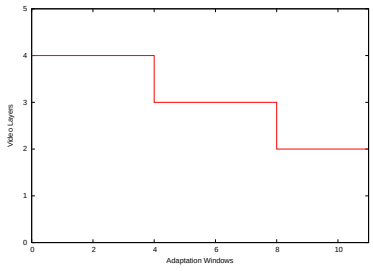
- h_t – number of layers in time slot t, where t = 1, …, T.
- z_t – indication of a step in time slot t, where $z_t \in \{0,1\}$ and t = 1, …, T.

The spectrum captures the frequency (the number of layer variations) and the amplitude (amount of layers decreased/increased in each layer variation) of quality variations. The frequency of variations is represented by z_t. Thus, a step in a time slot corresponds to an increase or decrease of video layers between two consecutive adaptation windows. A value of 0 for the spectrum represents the best possible quality, while the spectrum increases with a decreasing quality.

After further investigation, it turns out that the spectrum doesn't always capture well the fact that gradually reduced video quality (slow decrease of video layers) is generally better than rapid quality drops.



s(v) = 2

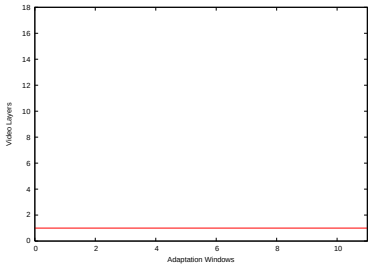


s(v) = 2

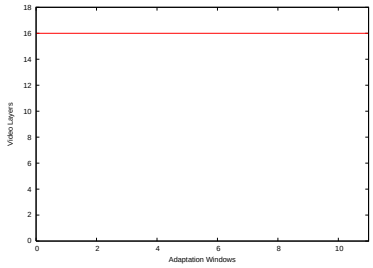
Figure 40: Rapid and Gradual Drops

In the two figures above, the spectrum calculation gives a result of 2 for both cases. In a crisis situation, the implemented algorithm of this thesis tries to achieve good video quality adaptation by gradually adjusting the amount of video layers to transmit. Thus, the algorithm is based on the assumption that gradual quality change is an essential strategy that leads to better perceived quality than a rapid change, which unfortunately is not well represented by the spectrum.

Another drawback with the spectrum metric is that the quality levels are not well represented either. A constant reception of only base layers (lowest quality level) for a number of adaptation windows is indicated by the spectrum to have equally good quality as a constant reception of all 16 layers (highest quality level) for the adaptation windows.



s(v) = 0



s(v) = 0

Figure 41: Lowest and Highest Quality Reception

The two figures above show that the spectrum calculation does not take into account the fact that the quality level (level of video layers) has an influence on the perceived quality. As long as there are no layer changes during the streaming session, the result calculated from the spectrum corresponds to perfect quality. In regard to the improvement code of this thesis, this is not an appropriate way to interpret the perceived video quality.

In order to achieve a reasonable assessment of the improvement code, it is necessary to develop a new simple metric for objective quality assessment with regard to the following two issues, as discussed above:

1) The metric must capture the fact that gradual quality changes are better than rapid quality changes.
2) The quality levels have an influence on the perceived video quality and must be taken into consideration by the metric.

Inspired by the spectrum, the following new objective quality metric is developed for this thesis, based on the two issues above. For easy reference, this new metric will be called spectrum2:

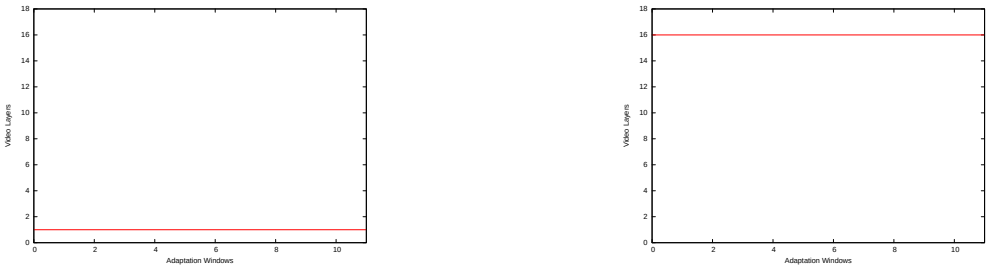
$$s_2(v) = \frac{1}{\frac{\sum_{t=1}^{T} h_t}{T}} + \left(\frac{\sum_{j=1}^{T} z_j d_j}{\sum_{k=1}^{T} z_k} \right) \qquad \text{(d)}$$

The variables h_t, z_t and d_t are defined as:

- h_t - number of layers in time slot t, where t = 1, …, T.
- z_t - indication of a step in time slot t, where $z_t \in \{0,1\}$ and t = 1, …, T.
- d_t – number of layer difference between h_{t-1} and h_t, when $z_t = 1$.

Notice that the definition of h_t and z_t are the same as for the spectrum. In this thesis, the time slot mentioned above corresponds to an adaptation window of fixed duration. That is, there are T adaptation windows in total for the streaming video. Similar to the spectrum metric, the lower the value, the better is the perceived quality.

The first part of the equation (d) corresponds to $\frac{1}{average}$, where *average* indicates the average number of total received layer amounts for all the adaptation windows. This means that the higher the quality level is, the lower is the value of the 'layer average' part. The following example describes this situation.



(1) **(2)**

Figure 42: Lowest and Highest Quality Level

Value of 'layer average' part for (1) of figure 42:

$$\frac{1}{\frac{\sum_{0}^{11} 1}{12}} = 1$$

Value of 'layer average' part for (2) of figure 42:

$$\frac{1}{\frac{\sum_{0}^{11} 16}{12}} = 0.0625$$

As the calculations above show, if the entire streaming video is represented by the lowest quality level, the 'layer average part' of the equation (d) gives a value of 1. On the other hand, if the highest quality level is achieved all the way, then the value is 0.0625. In other words, this metric takes the quality level into consideration when representing the perceived quality, unlike the spectrum where the result of both cases above are 0. It should be noted that if the perceived quality is perfect, then spetrum2 gives a value of 0.0625, which in the case of spectrum is 0.

The second part of the equation (d) corresponds to $\frac{sum\_layer\_diff}{tot\_num\_layer\_change}$, where *sum_layer_diff* indicates the sum of all layer difference amounts between the adaptation windows, and *tot_num_layer_change* indicates the total number of layer changes for the entire streaming session. In other words, this is a measure of the average layer difference between all the adaptation windows. That is, the higher the quality change is, the higher is the value of the average layer difference. The following example describes this.

(1) **(2)**

Figure 43: Higher and Lower Quality Changes

Value of 'average layer difference' part for (1) of figure 43:

$$\frac{(4-2)}{1} = 2$$

Value of 'average layer difference' part for (2) of figure 43:

$$\frac{(4-3)+(3-2)}{2} = 1$$

The calculations above show that if the quality drops gradually, then the 'average layer difference' part of the equation (2) gives a smaller value compared to when the quality changes faster. This is good in the sense that it matches the way good perceived quality is defined in this thesis, which is the fact that gradual quality changes are preferable when streaming over a network with highly varying bandwidth.

Since quality changes play a significant role in the assessment of the video quality, the weight is put on the 'average layer difference' part of the equation (2). That is, if the quality level is high for the streaming video and there are high quality changes during the playout, then the perceived quality of this video is considered to be less good in comparing to a video in which both the quality level and the quality changes are low. The following example explains this more clearly.

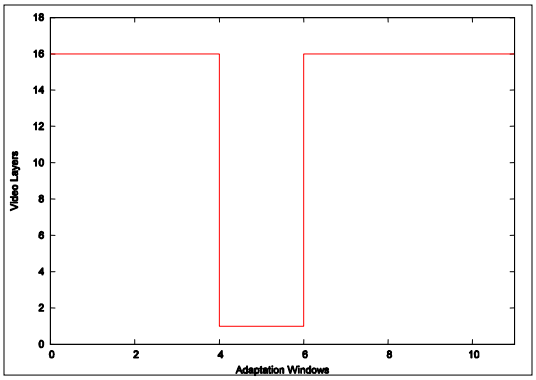


Figure 44: Higher Quality Level, High Quality Change

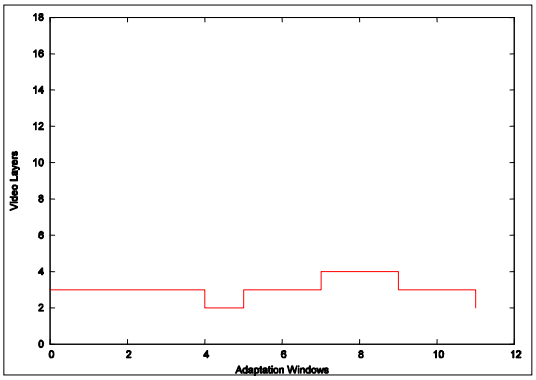


Figure 45: Lower Quality Level, Low Quality Change

$s_2(v)$ for the streaming session of figure 44:

$$s_2(v) = \frac{1}{\frac{\sum_{0}^{3} 16 + \sum_{4}^{5} 1 + \sum_{6}^{11} 16}{12}} + \frac{(16-1)+(16-1)}{2} \approx 15.074$$

$s_2(v)$ for the streaming session of figure 45:

$$s_2(v) = \frac{1}{\frac{\sum_{0}^{3} 3 + \sum_{4}^{4} 2 + \sum_{5}^{6} 3 + \sum_{7}^{8} 4 + \sum_{9}^{10} 3 + \sum_{11}^{11} 2}{12}} + \frac{(3-2)+(3-2)+(4-3)+(4-3)+(3-2)}{5} \approx 1.333$$

The example above shows that even though the highest quality level (16 video layers) for the video is reached most of the time during the streaming session of figure 44, the perceived quality of this video is still considered by spectrum2 to be worse than the one seen in figure 45, because of the high quality drop. As mentioned earlier, one of the purposes of the implemented quality-adaptive algorithm is to prevent high quality changes in the video from occuring, such as the one depicted in figure 43. Thus, it is

approriate that spectrum2 indicates the quality changes of figure 43 to be less good than those of figure 44.

Based on the examples given, this new objective metric is verified to a degree of being suitable for representing the perceived quality of layer-encoded video. The fact that it can distinguish between gradual and rapid quality changes, and that the objective quality is based on the quality levels of the video whenever there are no quality changes during a streaming session, makes it appropriate enough for the objective assessment of the further test cases in this thesis.

Although this metric works for the objective assessments in this thesis, it is not guaranteed to work in other circumstances. Since the development of an objective metric is not part of the goals of this thesis, further investigations are probably required in order for this metric to be working on a general basis.

5.5 An Objective Assessment of the Improvement Code

The objective assessment of the implemented quality-adaptive algorithm is based on a number of test cases. Since the bandwidth of an unreliable, wireless network can vary in numerous ways, it is hard to pick a specific scenario that can be used to verify how efficient the implemented algorithm is. Thus, it is better to show how the algorithm reacts upon different rates of bandwidth development. In a wireless network, the available bandwidth for a mobile receiving device can increase or decrease at different rates according to the condition of the network and the signal strength between the sender and the receiver. Thus, it is interesting to show how the implemented algorithm handles the different bandwidth rates. This section concludes with an objective quality comparison between the original Qstream code and the improvement code. The comparison is based on bandwidth scenario 5, which consists of 5 sub-scenarios that capture some interesting network conditions.

5.5.1 Objective Quality when Connectivity is Getting Bad

Items in use for this test:

- **Media Content:** An MPEG-1 file of about 2 minutes running time converted to SPEG
- **Bandwidth Scenario:** 3
- **Threshold Variable Values:** Different values of *ig_wt*: 10, 15, 20, 25, 30, 35, 40, 45, 50, 55 and 60.

The following test cases are based on bandwidth scenario 3 of section 5.1.

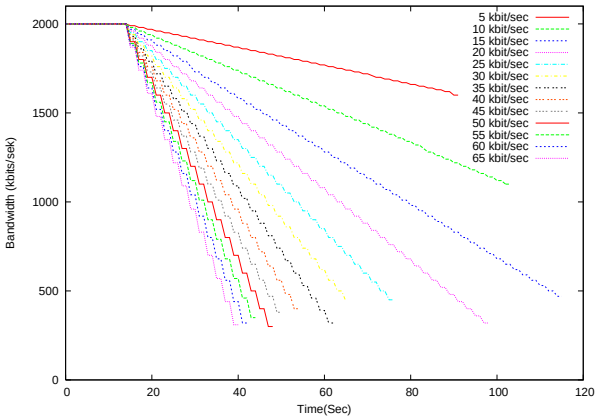


Figure 46: Bandwidth Scenario 3

The different graphs of bandwidth scenario 3 represent 13 sub-scenarios in which the bandwidth drop rate starts out at 5 kbit/sec and grows by 5 kbit/sec for each sub-scenario.

Recall that the threshold variable *ig_wt* decides when StreamServ can ignore the bandwidth prediction. Thus, when this threshold is reached, StreamServ is able to increase the video layers to transmit for each consecutive adaptation window until the highest quality level is reached. When the bandwidth drops at different rates, it is interesting to see how various values of this threshold variable affect the objective quality of the streaming video.

Notice that in each of the 13 sub-scenarios, the bandwidth starts out at 2 Mbit/sec and remains at this value for 15 seconds before it starts dropping. The following objective assessments that make use of these sub-scenarios are based on the period after these 15 seconds, since the interesting part of the assessments lies in the moments in which the bandwidth is dropping.

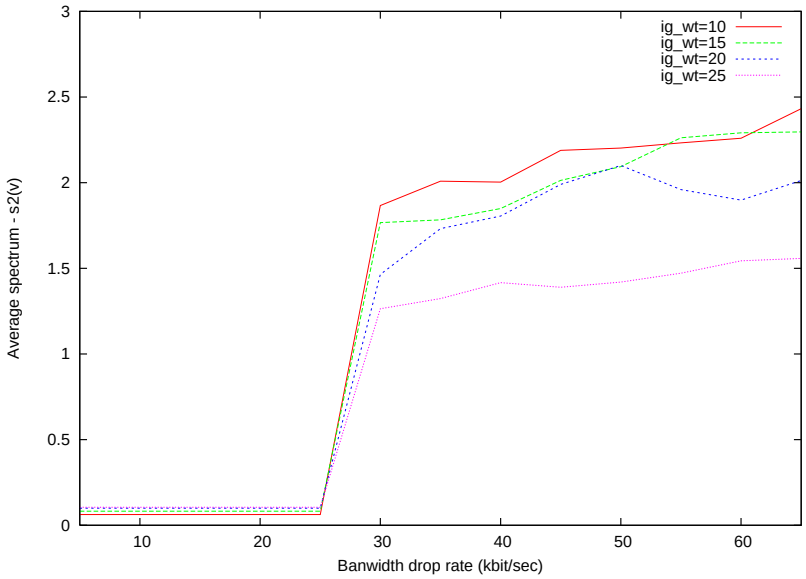


Figure 47: Objective Quality when Bandwidth Decreases, ig_wt = 10 to 25

Figure 47 and figure 48 show the objective quality for a number of test cases (streaming sessions) with different combinations of the sub-scenarios (different bandwidth drop rates) and *ig_wt* values. The horizontal axis indicates the different bandwidth drop rates, while the vertical axis indicates the average spectrum2 value, which corresponds to the objective quality achieved from the different streaming sessions. According to the definition of spectrum2, a value of 0.0625 represents the best possible quality, while the spectrum2 increases with a decreasing quality. That is, the objective quality decreases with each increasing y value.

An interesting detail to notice from figure 47 is that for the *ig_wt* values of 10 to 25, the perceived quality of the streaming sessions are nearly perfect for drop rates up to 25 kbit/sec. As mentioned, the bandwidth starts out at 2 Mbit/sec and remains at this value for 15 seconds before it starts dropping. During these 15 seconds of good network condition, the implemented quality-adaptive algorithm put aside an amount of workahead for a time of crisis. In other words, the bandwidth drop rates up to 25 kbit/sec did not cause any problems for the streaming sessions, since the collected workahead amounts were sufficient to cover the period in which the bandwidth was dropping.

Since the quality achieved is nearly perfect even when the drop rate reaches 25 kbit/sec, this means there has been a greedy consumption of workahead time to provide better video quality during the period when the bandwidth was dropping, due to the low values of *ig_wt*. The risk in such behaviour is that high quality changes can occur, if the condition of the network stays bad for a longer period or the bandwidth drop rate gets higher. When the drop rate surpasses 25 kbit/sec in figure 47, there is a noticeable high jump in the objective quality of the streaming sessions with *ig_wt* values of 10 to 25. The interesting detail to notice here, is that the larger *ig_wt* is, the lower the quality jump is.

The reason high quality jumps occur at the lower *ig_wt* values, is because of the threshold variable *c_wt*. This variable is also known as the crisis workahead threshold variable. As described in section 4.2 of chapter 4, it decides how much of the workahead time that can be used for transmitting late adaptation windows. The initial value of this variable is 10 seconds. The lower the *ig_wt* value is, the more workahead time is spent to transmit better quality during a period of bad connectivity. Thus, the possibility of the workahead time reaching the threshold *c_wt* is also larger. If it happens that the workahead time falls below *c_wt* during the transmission of an adaptation window X, then the rest of the contents of window X are dropped. However, if the amount of data transmitted for window X-1 was much more than for window X, due to greedy consumption of workahead, then there will be a high quality jump in the transition from window X-1 to window X. This explains why low *ig_wt* values might lead to high quality changes, as figure 47 shows.

Figure 48 shows the objective quality of the streaming sessions with *ig_wt* values 30 to 60. Notice that the spectrum2 values achieved from the sessions of drop rates up to 25 kbit/sec are higher than the ones seen in figure 47. Since the *ig_wt* values are higher, it means that the consumption of workahead time is less greedy. StreamServ spends more time limiting the amount of data to transmit according to the threshold variables *b_pt*, *ba_lt* and *bs_lt* (section 4.2 of chapter 4). In other words, StreamServ is more cautious when transmitting the contents of the adaptation windows, and as a result certain enhancement video layers for a number of adaptation windows are dropped in favour of more workahead time. The consequence is reduced video quality, which is reflected by the higher spectrum2 values. But although the perceived quality is less good for drop rates up to 25 kbit/sec, the payoff is seen in the streaming sessions with higher drop rates. The quality jump at drop rate 30 kbit/sec and the spectrum2 values of the sessions above this drop rate are lower than those seen in figure 47.

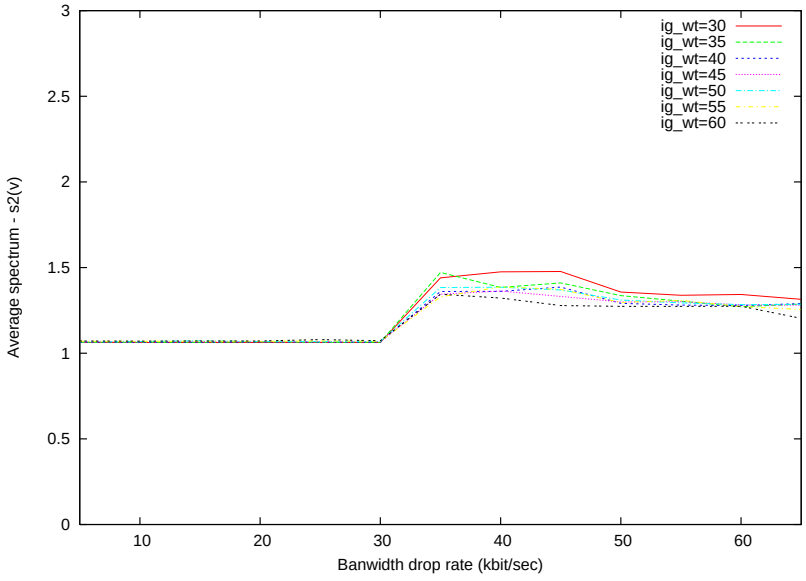


Figure 48: Objective Quality when Bandwidth Decreases, ig_wt = 30 to 60

These test cases clearly show that the threshold variable *ig_wt* has an effect on the achieved objective quality. However, the question that ought be asked, is whether a high value of *ig_wt* always gives a good result.

When looking at the streaming sessions of these test cases from another perspective, it turns out that increasing *ig_wt* is not always a good idea, if the decreasing bandwidth is not yet critical for the performance of the streaming process.

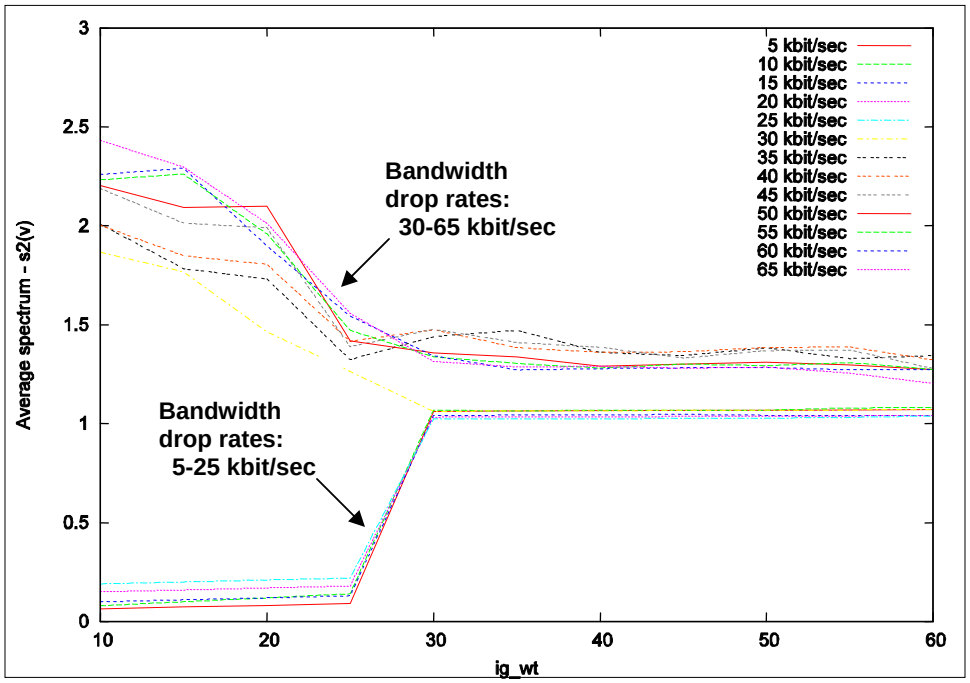


Figure 49: Objective Quality for ig_wt = 10 to 60 from a Different Perspective

Figure 49 represents the same streaming sessions from figure 47 and figure 48, but viewed from a different perspective. In this figure, the horizontal axis indicates the different values of *ig_wt*, while the vertical axis indicates the average spectrum2 value as before.

An important thing to notice here is the development of the objective quality when using the sub-scenarios that correspond to drop rates up to 25 kbit/sec. At the value of 30 for *ig_wt*, there is a high jump in the objective quality, which indicates that the perceived quality is reduced. Since StreamServ could afford transmitting nearly perfect quality at the lower *ig_wt* values, despite the fact that the bandwidth was dropping, this means it was not the decreasing bandwidth that was causing the reduced quality. The reason this happened, was because the increased *ig_wt* values made the implemented algorithm behave more cautiously than necessary when transmitting the contents of the adaptation windows. In other words, precaution against network failures was preferred most of the time rather than taking risks to transmit more contents, even though the slowly decreasing bandwidth was still sufficient for more contents to be transmitted. This means the available bandwidth was not utilized well enough, because the *ig_wt* was set too high.

If the bandwidth decreases at a higher rate than 25 kbit/sec, the spectrum2 values of the sessions with lower *ig_wt* values are also higher. This can be seen in the graphs of Figure 49 49 that represent the sessions with drop rates above 25 kbit/sec. Even though the quality achieved with low *ig_wt* values are bad compared to the sessions in which the drop rate is lower, this is made up for as *ig_wt* increases. The implemented algorithm is less greedy as *ig_wt* grows, and thus, the quality changes occur in a slower manner.

The point to grasp here, is that high *ig_wt* values are not always efficient for the streaming process, if the bandwidth initially is good but starts dropping at a low rate. On the other hand, if the bandwidth drop rate is high and leads to bad connectivity over a longer period, then it's the low *ig_wt* values that might impede the efficiency of the streaming. The low *ig_wt* values make StreamServ over-estimate the capacity of the available bandwidth. Thus, more workahead time is consumed, and high quality changes might occur if the workahead time falls below the threshold *c_wt*.

The conclusion to be drawn from the test cases of this section, is that high values of *ig_wt* are preferable if the bandwidth is decreasing. Although high values of *ig_wt* can lead to under-estimation of the capacity of the available bandwidth, it is probably better to take precaution after all, rather than to risk getting high quality changes or playout interruptions. Another way to avoid high quality jumps when using low *ig_wt* values during periods of bad connectivity, is to lower the threshold c_wt. This doesn't actually solve the problem, but at least it provides StreamServ with some more time to transmit the late adaptation windows. However, it also heightens the risk of getting playout interruptions sooner, if the bad connectivity lasts over a longer period.

Since the actual future condition of a wireless network is unknown, it is impossible to predict exactly how the bandwidth will behave whenever it seems to decrease/increase. Thus, it is a matter of choice whether to transmit fewer data in favour of workahead time, or to transmit more data at the risk of getting higher quality changes.

5.5.2 Objective Quality when Connectivity is Getting Good

Items in use for this test:

- **Media Content:** An MPEG-1 file of about 2 minutes running time converted to SPEG
- **Bandwidth Scenario:** 4
- **Threshold Variable Values:** Different values of *ig_wt*: 10, 15, 20, 25, 30, 35, 40, 45, 50, 55 and 60.

The following test cases are based on bandwidth scenario 4 of section 5.1.

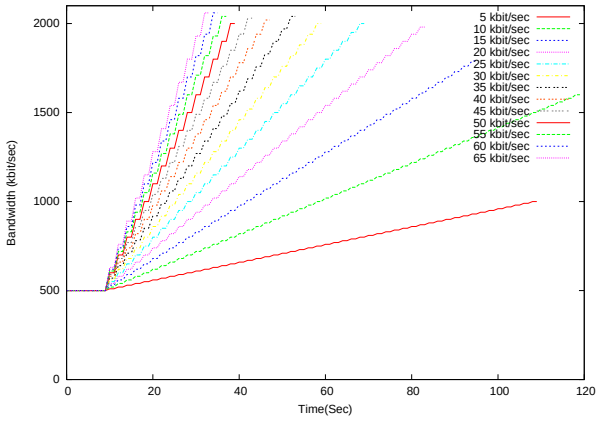


Figure 50: Bandwidth Scenario 4

The different graphs of bandwidth scenario 4 represent 13 sub-scenarios, in which the bandwidth growth rate starts out at 5 kbit/sec and increases by 5 kbit/sec for each sub-scenario.

Similar to the test cases of the previous section, the following ones are also making use of different values of the threshold variable *ig_wt*. However, in these test cases the purpose is to show how the implemented algorithm reacts upon different bandwidth growth rates.

In each of the sub-scenarios, the bandwidth starts out at 500 kbit/sec and remains at this value for 10 seconds before it starts increasing. The intention is to investigate how the implemented algorithm handles the gradual transition from bad to good connectivity. The following objective assessments that make use of these sub-scenarios are based on the period after the 10 seconds. Similar to the test cases of bandwidth drops, the interesting part of the assessments lies in the moments in which the bandwidth is increasing.

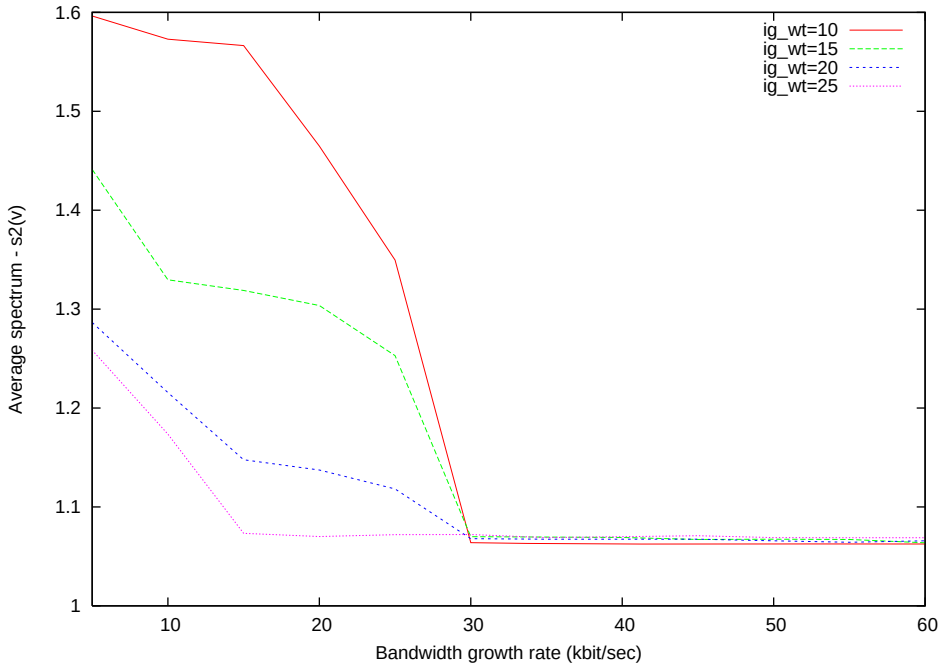


Figure 51: Objective Quality when Bandwidth Increases, ig_wt = 10 to 25

Figure 51 and figure 53 provide an illustration of the objective quality achieved from the test cases (streaming sessions). These are based on combinations of different bandwidth growth rates and *ig_wt* values. The horizontal axis indicates the different bandwidth growth rates, while the vertical axis indicates the objective quality of the streaming sessions.

An interesting graph of figure 51 is perhaps the one that corresponds to the *ig_wt* value of 10. This graph shows that the objective quality is quite bad for the streaming sessions that are based on low bandwidth growth rates. This happens because of the low *ig_wt* value. When the workahead time surpasses the *ig_wt* threshold, StreamServ aims at top quality transmission. Thus, the number of enhancement video layers is increased by 1 for each consecutive window with the intention of reaching the highest quality level. However, if the bandwidth growth rate is low, the bandwidth will probably be overloaded at some point during the transmission. In other words, the slowly increasing bandwidth can not keep up with the continuous increasing amount of data to transmit. Thus, the time interval (section 3.2.2 of chapter 3) used to transmit an amount of data will eventually be longer. This results in a percentage of loss being returned from the bandwidth prediction function. However, if the *ig_wt* value is very low, then the possibility of StreamServ ignoring the prediction is also larger. If this is the case, then the workahead time will still be consumed in a greedy way. Thus, the workahead time might fall below the threshold *c_wt*. The consequence are high quality jumps, which are reflected by the high spectrum2 values. Figure 52 provides an illustration of a streaming session in which the *ig_wt* value and the bandwidth growth rate are low. Notice all the quality jumps, which are caused by the issues mentioned above.

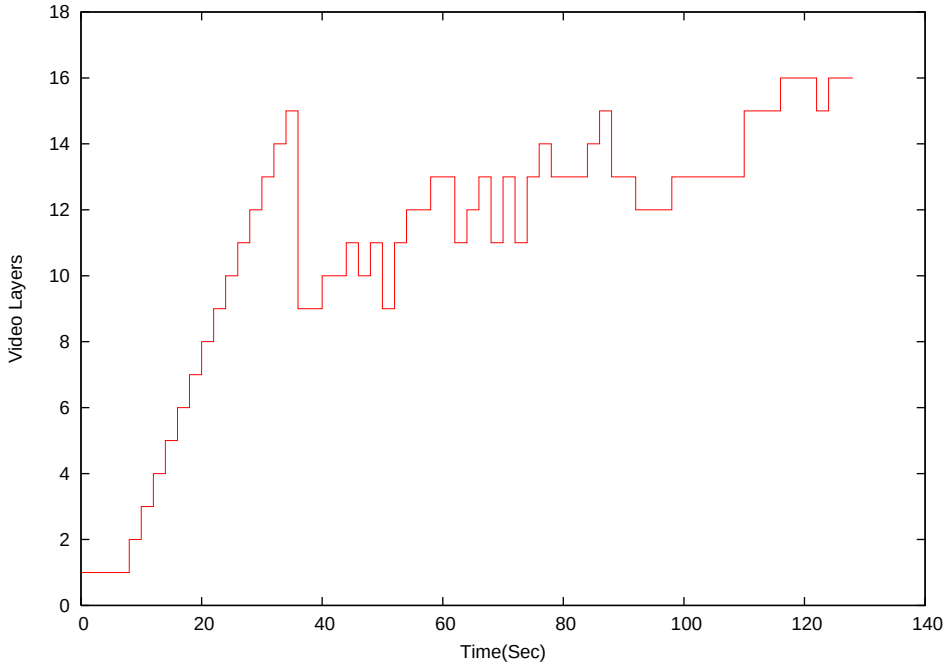


Figure 52: Streaming Session with Low ig_wt and Low Bandwith Growth Rate

As the growth rate increases in figure 51 and figure 53, the objective quality of the sessions get better, which is indicated by the lower spectrum2 values. This is not surprising, as higher bandwidth makes it possible for more data to be transmitted without complications. Thus, higher video quality is achieved.

Figure 51 also shows that when *ig_wt* increases, the objective quality is better for the streaming sessions that are based on low bandwidth growth rates. Larger *ig_wt* values mean that StreamServ spends more time limiting the amount of data to transmit according to the threshold variables *os_pt, be_pt*, *s_lt* and *st* (section 4.2 of chapter 4). Thus, certain enhancement layers are dropped if the bandwidth is limited, and the probability of overloading the available bandwidth is smaller. On the contrary, the lower *ig_wt* is, the sooner StreamServ ignores the four threshold variables and further bandwidth predictions with the intention of increasing the quality. If the bandwidth initially is low and the growth rate is also low, then such behaviour heightens the risk of getting high quality jumps, as described above.

In figure 53, the graphs correspond to the streaming sessions with *ig_wt* values 30 to 60. Notice that when the bandwidth growth rate is low, the objective quality achieved from these sessions are better than the ones of figure 51. StreamServ is more cautious with the amount of data to transmit for each increasing *ig_wt* value. Thus, even though the growth rate is low, there is a higher possibility that the bandwidth will not be overloaded. The result is better objective quality.

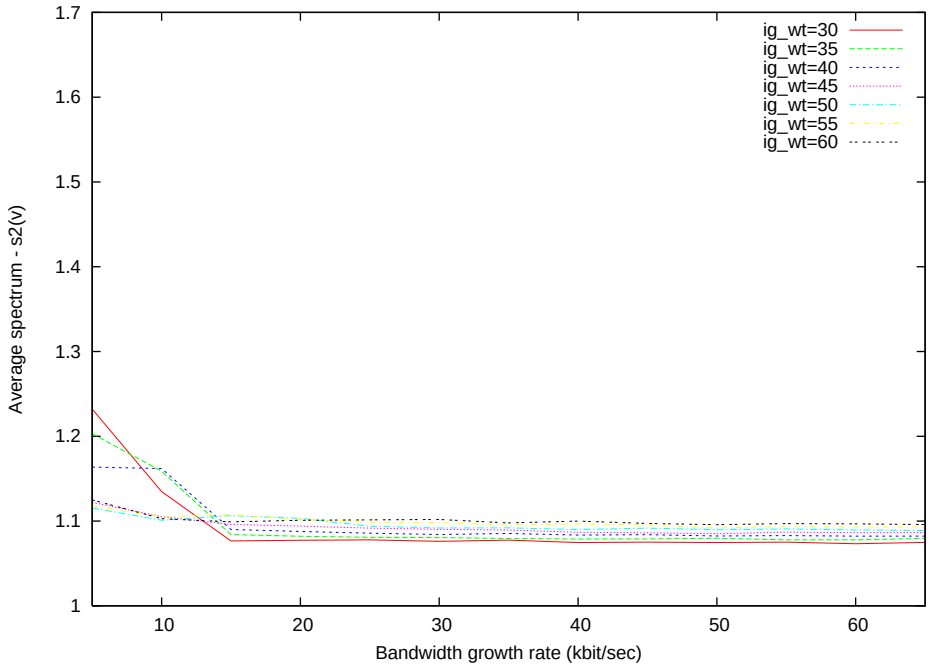


Figure 53: Objective Quality when Bandwidth Increases, ig_wt = 30 to 60

There is another interesting detail to notice from figure 51 and figure 53. The higher *ig_wt* is, the less good is the objective quality of the sessions with high growth rates. Figure 54 provides a better and clearer illustration of this.

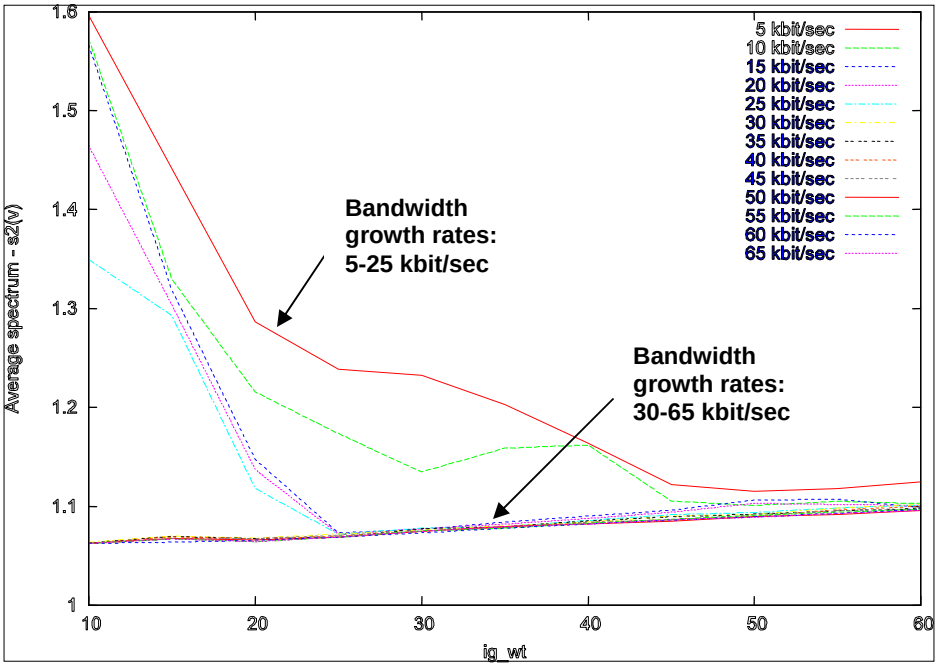


Figure 54: Objective Quality when Bandwidth Increases from a Different Perspective

Figure 54 shows that when the bandwidth growth rate is high, lower *ig_wt* values give the best objective quality. On the other hand, if the bandwidth growth rate is low, the objective quality is better with higher *ig_wt* values. However, if both the growth rate and the *ig_wt* value are high, then the capacity of the available bandwidth might be under-estimated by StreamServ. Recall from the test cases of the previous section, that when the bandwidth drop rate is low, high *ig_wt* values might lead to under-estimation of the available bandwidth. Thus, the bandwidth is not utilized well enough for the streaming. A similar problem occurs when the bandwidth increases. In this case, as mentioned above, it is the high growth rate that might lead to inefficient streaming if the *ig_wt* value is set too high. The logical explanation to this might be clear by now. If the bandwidth is good and keeps increasing at a high rate, then there is no point in transmitting data in a cautious way by dropping enhancement video layers. By doing so, the perceived quality of the video is unnecessarily bad, because the available bandwidth is not utilized well enough.

From the result of the test cases in this section, there is an indication that high *ig_wt* values might result in better objective quality in general. Even though a high *ig_wt* value can lead to bad utilization of the available bandwidth, it is probably better this way. If the bandwidth is under-estimated (due to high *ig_wt* value), the probability of getting fewer high quality jumps is larger, but at the expense of achieving lower quality levels. On the other hand, if the bandwidth is over-estimated (due to low *ig_wt* value), the probability of getting more high quality jumps is larger. But the advantage is that higher quality levels are more likely to be achieved, if the bandwidth is sufficient. According to the goal of this thesis, fewer high quality jumps are preferable. Thus, a higher *ig_wt* value is probably more preferable.

5.5.3 An Objective Quality Comparison Between The Original Qstream Code and The Improvement Code

Items in use for this test:

- **Media Content:** An MPEG-1 file of about 2 minutes running time converted to SPEG
- **Bandwidth Scenario:** 4
- **Threshold Variable Values:** Different values of *ig_wt*: 10, 60

This final test is designed with the intention of comparing the performances of the improvement code and the original Qstream code. There are many different ways in which the bandwidth can vary. Rather than investigating all sorts of variations, this test is based on bandwidth scenario 5 of section 5.1. This scenario consists of 5 sub-scenarios. Sub-scenarios 1-4 are meant to represent the kinds of intense bandwidth variations that will lead to high quality jumps in the streaming video, if the original Qstream implementation is used. The purpose is to verify that the improvement code handles these variations in a better way. The last sub-scenario shows the bandwidth being constantly good. It should be noted that the bandwidth in this sub-scenario is sufficient for an entire SPEG file (converted from MPEG-1) to be transmitted at full quality. The purpose is to

show whether there is any difference in the perceived quality when using either the original Qstream code or the improvement code for streaming over a network, in which the bandwidth is good with no variations.

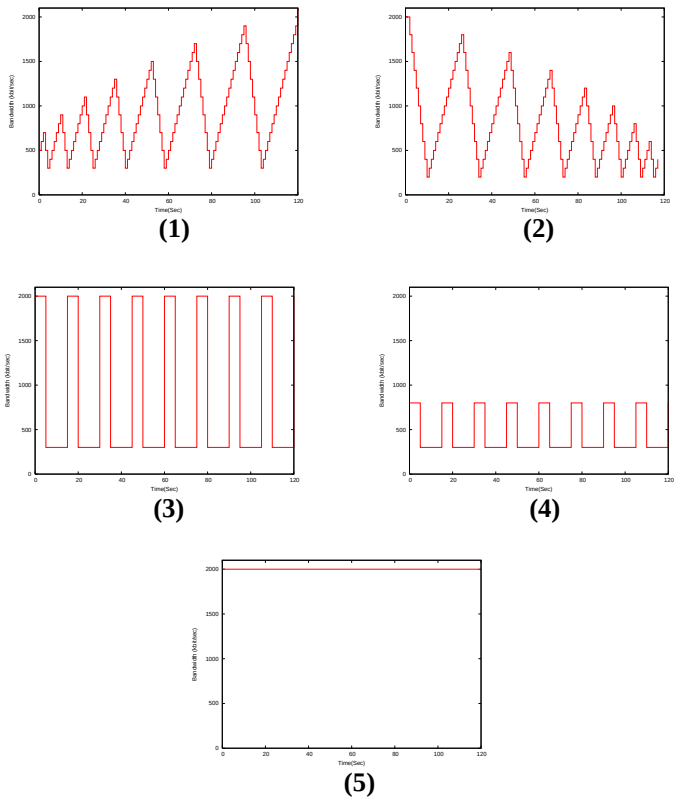


Figure 55: Bandwidth Scenario 5

The first sub-scenario shows that the bandwidth starts out low, but gradually increases through a period of intense variations. The second sub-scenario shows the opposite. The third sub-scenario simulates a situation in which the connectivity either is very good or very bad. An important detail to notice here is that the durations of bad connectivity are longer than those of good connectivity. The fourth sub-scenario shows a similar bandwidth variation pattern as the third. However, the difference is that the connectivity is never really good. It is either bad or worse. As described, the fifth sub-scenario represents a bandwidth that is constantly good with no variations. It should be good enough for an SPEG file (derived from an MPEG-1 file) to be transmitted at top quality without any complications.

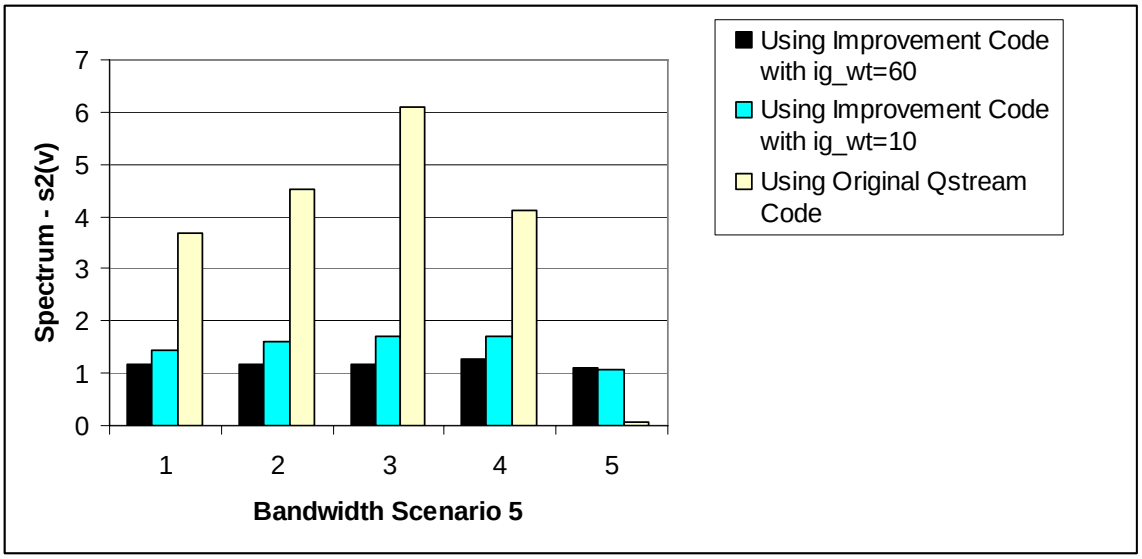


Figure 56: **Comparing the Improvement Code with the Original Qstream Code**

Figure 56 provides an illustration of the outcome of the test cases (streaming sessions). The horizontal axis indicates the 5 sub-scenarios of bandwidth scenario 5. The vertical axis indicates the spectrum2 value achieved from the streaming sessions.

This figure shows that when the bandwidth varies intensely over a longer period, the original Qstream code is not able to handle the quality-adaptation too well in comparing to the improvement code. This is reflected by the highest spectrum2 values in the figure, which are achieved from using the original Qstream code. Worth noticing is the streaming session in which the original Qstream code is used with sub-scenario 3. This one has the highest spectrum2 value of all the sessions depicted in the figure, which means that the objective quality of this session is the least good of them all. Since the bandwidth in sub-scenario 3 is very good in certain periods, StreamServ's attempts to transmit top quality during these periods are successful. However, when the bandwidth suddenly drops to an unusable level, the streaming process is affected instantaneously, which results in a high quality jump. This session is depicted in figure 57.

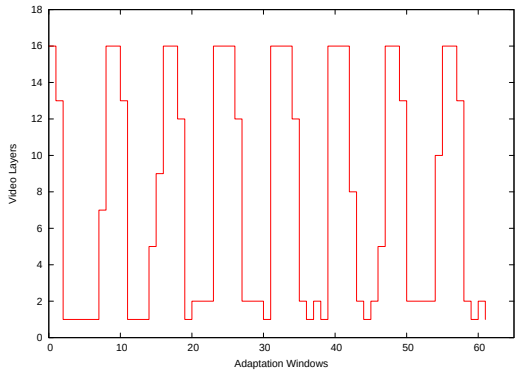


Figure 57: Streaming Session Using Original Qstream over Sub-Scenario 3

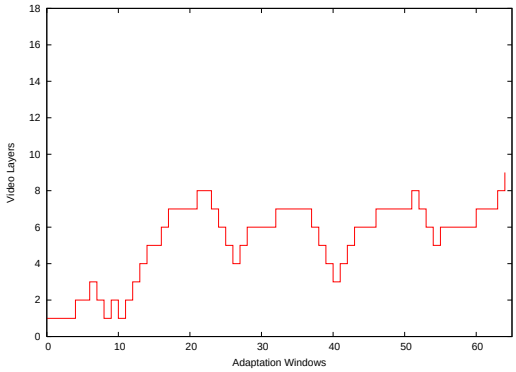


Figure 58: Streaming Session Using Improvement Code over Sub-Scenario 3, with ig_wt = 60

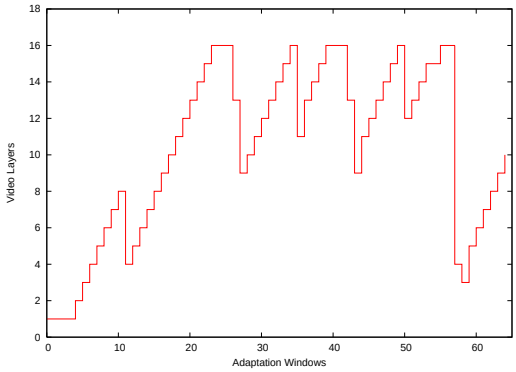
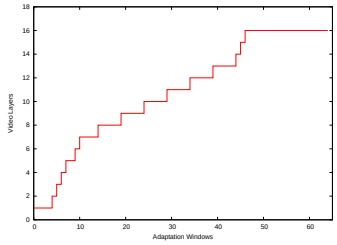


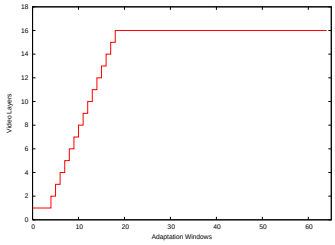
Figure 59: Streaming Session Using Improvement Code over Sub-Scenario 3, with ig_wt = 10

When the improvement code is applied to the original Qstream code, a much better objective quality is achieved for the streaming sessions over the 5 sub-scenarios, as seen in figure 56. Figure 58 and figure 59 show the outcomes of two streaming sessions using the improvement code. Sub-scenario 3 is used for these two sessions, and the outcomes are quite improved compared to the one depicted in figure 57. Even when the *ig_wt* value is set to 10, the outcome is still better than the one achieved with the original Qstream code. Because of the greedy consumption of workahead time, there are still a couple of high quality jumps, as seen in figure 59. However, with a *ig_wt* value of 60, all the high jumps are successfully avoided. As explained earlier, StreamServ is more cautious with the transmission, and more workahead time is put aside for emergency cases. Thus, whenever the bandwidth drops, StreamServ can afford applying a gradual decrease of quality to the streaming video as desired. The disadvantage is that the overall quality levels achieved are rather low. As long as high quality jumps are avoided, the objective quality of the sessions with higher *ig_wt* values are still better, even though the quality levels might be low. This is clearly shown in all the test cases in figure 56 that are using sub-scenarios 1-4.

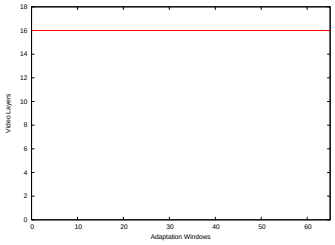
On the other hand, if the bandwidth is good with no variations, then the original Qstream code is the clear winner when it comes to the objective quality. This can be seen in figure 56 when using sub-scenario 5.



(1)
Improvement Code,
ig_wt=60, $s_2(v)$ = 1.092199



(2)
Improvement Code,
ig_wt=10, $s_2(v)$ = 1.074286



(3)
Original Qstream Code
$s_2(v)$ = 0.0625

Figure 60: Streaming Sessions over Sub-Scenario 5

The reason is illustrated in figure 60. This figure shows the outcome of streaming sessions achieved from using the improvement code and the original Qstream code over sub-scenario 5. When using the original code, top quality is achieved for the entire streaming video. This is not a surprise, as the original StreamServ aims at top quality transmission no matter how the condition of the network is. However, when using the improvement code, the threshold variables make StreamServ act more cautious during the transmission. Based on the fact that the future condition of a wireless network is unknown, the improved StreamServ puts aside a certain amount of workahead time before it starts top quality transmission. Thus, the quality of the first adaptation windows are low. An additional enhancement layer is added for each consecutive window, as long as the bandwidth is predicted to be good. So it takes a while before the highest quality level is reached. This is the reason why the objective quality achieved from the improvement code is lower than when using the original code.

The figure above also shows that the top quality level is reached sooner if the *ig_wt* value is low. In other words, the objective quality is better if a low *ig_wt* value is used instead, which is also shown in figure 56. Unfortunately this doesn't come out very clear in figure 56, since the difference is just 0.017913. But notice that the value of the 'average layer difference' part is the same for both (1) and (2) of figure 60. So although the difference value is small, it is still a significant value in regard to the 'layer average' part of the metric.

To summarize, the test cases of this section show that when the bandwidth is highly varying, the improvement code leads to a much better outcome compared to the original code. The perceived quality of the video is more balanced throughout the streaming session, which is expressed by both the objective quality metric and the supporting figures. However, if the bandwidth is constantly good and sufficient for top quality transmission, then the original code leads to better performance.

5.6 Evaluation of the Improvement Code

Based on the test results of the previous sections, it is now appropriate to claim that the improvement code handles streaming sessions over a network with intensely varying bandwidth in a more efficient way than the original code does.

The threshold variables (section 4.2 of chapter 4) are one part of the improvement code that determine the level of efficiency. Originally all the threshold variables were intended to be adjusted, in order to investigate the trade-off between video quality and workahead. However, further research showed that most weight is put on the *ig_wt* variable. By adjusting this variable, it led to the most visible and essential results of the test cases. Another threshold variable closely related to the *ig_wt* variable, is the *c_wt* variable. Recall that it is the *c_wt* variable that actually causes the high quality jumps. The lower the *ig_wt* value is, the sooner the workahead time reaches the *c_wt* threshold, if the connectivity is bad. Thus, the possibility of getting high quality jumps is larger. By lowering the value of this variable, StreamServ is able to get some extended time to transmit late adaptation windows. If the connectivity is bad over a longer period, then a low value of *c_wt* might lead to playback interruption sooner. Although in general, if the duration of bad connectivity is very long, then the playback interruption will occur after a while anyway.

The 'bandwidth prediction' function is another part of the improvement code that has a significant influence on the level of efficiency achieved. This function is based on a comparison of two values, which correspond to the number of bytes written to TCP per second for the two latest fixed time interval. The comparison makes it possible to predict whether the bandwidth is increasing or decreasing, by computing a percentage of gain or loss based on the two values. However, the outcome of the function does not reflect the actual capacity of the available bandwidth. Thus, whenever the predicted percentage is either a gain or a loss, StreamServ only knows that the bandwidth is either increasing or decreasing, respectively. It does not actually know the amount of data that will lead to fully utilization of the current available bandwidth. Therefore, the amount of data to be transmitted is determined by comparing the predicted percentage with the threshold variables *os_pt*, *b_pt* and *be_pt*. The idea behind these threshold variables is to help avoiding over-estimation of the available bandwidth. This can be achieved by setting the *b_pt* threshold to a low value (initial value 10) and the *be_pt* to a high value (initial value 40). In other words, precaution is preferred rather than taking risks.

However, during the research it turned out that the predicted percentages of gain rarely reached the initial value of *be_pt*. Unless the bandwidth was changing rapidly from low to high, the percentages of gain normally were below *be_pt*, which corresponded to a 'stable' network condition. When lowering the value of *be_pt*, it led to over-estimation of the bandwidth, which resulted in more frequent high quality jumps whenever the bandwidth was not too high. Thus, it was better off leaving the *be_pt* variable at the initial value.

Since the predicted percentages of gain mostly corresponded to a 'stable' network condition, it was necessary to introduce the threshold variable *st*, in order to increase the efficiency of the streaming process. Recall that this threshold variable decides when a 'stable' network condition is good enough for increasing the number of enhancement layers to transmit. Thus, in the case of a 'stable' network condition, the number of allowed enhancement layers is not always restricted by a fixed value of the *s_lt* threshold variable.

In the test cases of section 5.5.3, it was shown that when the bandwidth was constantly good, the improvement code led to a poorer outcome compared to the original code. Since the future condition of an unstable network is unknown, the improvement code must make sure that StreamServ acts more cautiously. This is a necessary precaution that should be considered, even if the bandwidth is high at the start-up of a streaming session. However, if the viewer accepts the risks of getting playback interruptions or frequent quality jumps, then it is possible to achieve better quality by lowering the values of the threshold variables *ig_wt*, *l_wt* and *c_wt*, provided that the bandwidth is good enough.

Apart from the issues mentioned above, generally the improvement code delivers the desired outcomes. However, it should be mentioned that certain factors might have affected the outcome of the test cases. This is the reason why it was necessary to compute the average spectrum2 values for most of the test cases. The average value was based on several outcomes of the same test case.

The following factors might have affected the outcomes:

- **Computer performance** - The processor and memory chips of the sender and receiver machines are important hardwares which play a significant role in the efficiency of the performance.

- **Accuracy of bytes written to TCP** – Investigations reveal that the number of bytes written to TCP for the different time intervals is a little varying, even though the numbers are based on a fixed time interval, and the bandwidth is equally good throughout the streaming session. However, when the bandwidth is highly varying, these minor variations are not significant at all. Since the time intervals differ greatly whenever the bandwidth varies intensely, they compensate for the minor variations of the written bytes.

- **Human delays** - A test case is run by manually starting the streaming process and a bandwidth scenario script simultaneously.

Chapter 6: Conclusion

This chapter provides a description of the achievement of this thesis and some ideas about future work.

6.1 The Achievement of this Thesis

The primary goal of this thesis was to implement an optimal solution for efficient utilization of the varying bandwidth of a wireless network when streaming a video.

An improvement code was developed and added to the existing Qstream software. This software supported quality-adaptive streaming by dynamically adjusting the data rate of the streaming video, according to the condition of the network. The video was rate-adjustable, because it was in a layer-encoded format. This made it possible to apply a priority data dropping strategy, which basically meant that the less important video layers (aka. enhancement layers) were dropped if the connectivity of the network was getting bad during the streaming session. Thus, the data rate of the video got lower. In other words, the streaming video was dynamically adapted to the condition of the network.

An important fact that the improvement code was based on, was the independency between the transmission speed and the playback speed of the video. Buffering was employed at the client side, and thus, whenever the connectivity was good, data could be transmitted faster than playback speed and got temporarily stored in the buffer for later use.

The fact that the video data could be transmitted faster than playback speed, was a fundamental assumption for developing the improvement code. Whenever an amount of data was transmitted faster than the playback speed, the streaming server would be lying ahead of time.. This amount of time was known as the workahead time. Basically, the idea behind the improvement code was to collect workahead time during a period of good network connectivity, and then make use of that amount of time in a most efficient way during the period in which the connectivity was bad. If there was no workahead time available during a period of bad connectivity, then either the streaming process would stop immediately, or there would be high quality jumps occuring in the streaming video. This was not considered to be a desirable outcome. Thus, the workahead time would assist in smoothing out the rate changes, providing a gradual change of the video quality. The workahead time would also help lowering the risk of getting playback interruptions during a period of bad connectivity.

Since the bandwidth of a wireless network could be intensely varying, a challenge would be to decide wisely the amount of workahead time to acquire during good connectivity, and the amount to spend during bad connectivity.

In chapter 3 and 4, the improvement code was proposed and implemented, respectively. The improvement code consisted of a number of functions that would co-operate with each other and the existing software to achieve the goal. The functions solved the following tasks:

- Perform predictions of the future condition of the network on a regular basis, based on past conditions of the network.
- Determine the amount of data that can be transmitted, based on the outcome of the prediction of the network condition.
- Update and keep an overview of the amount of workahead time available for the streaming server to use.
- Determine the amount of workahead time to acquire based on the outcome of the prediction of the network condition. This is done by dropping enhancement video layers.
- Determine the amount of workahead time that the streaming server can make use of during a period of bad connectivity, in order to achieve the most efficient way of transmitting data.

Since there was a trade-off between the quality of the video and the workahead time, it was necessary to implement a set of threshold variables. These variables would help finding a balance point for the trade-off.

In chapter 5, the improvement code was tested with a number of streaming sessions (test cases). In order to investigate how efficient the improvement code was, the streaming sessions were based on bandwidth scenario scripts which emulated different bandwidth variations of a wireless network. In order to make an objective assessment of the performance of the improvement code, an objective quality metric was needed. Originally the intention was to use a pre-developed objective metric called spectrum to make the objective assessment. However, after further investigations, it turned out that this metric did not work as it was supposed to. Thus, this led to the development of a new objective metric spectrum2.

By making use of the new objective metric, it was possible to evaluate the outcomes achieved from the test cases. By adjusting a certain threshold variable, the outcomes were significantly changed. According to the condition of the network, some outcomes were better than other. Although certain threshold variables didn't lead to any visible improvements when adjusted, it didn't change the fact that they should be adjustable variables. It would be wrong to claim that one specific value of the threshold variables would lead to the best outcome in all situations, because in reality, the bandwidth of an unreliable, wireless network could behave in unexpected ways.

6.2 Future Work

Given the time constraints that were provided for this thesis, there are still some issues that would have been interesting to elaborate.

- In several of the test cases, there was still an amount of workahead time left when StreamServ had finished transmitting the entire video stream. Since the actual future condition of the network is unknown, StreamServ does not really know the most efficient amount of workahead time to put aside. Thus, it happens that StreamServ puts aside more workahead time than needed. This results in a video stream with poorer quality than it should be. The workahead time left could have been used to increase the quality of some of the already transmitted adaptation windows, but unfortunately this is not possible with the current implementation. An idea could be to implement a solution that would utilize this remainding amount of workahead time. Since the transmission is ahead of the playback, a possibility could be to find a way to increase the quality of some of the already transmitted adaptation windows that have not reached the display phase yet.

- The improvement code of this thesis is based on a linear video stream. Consider a stream that is not linear, which means that the viewer will choose to pause, fast forward, rewind, etc. It would be interesting to implement a solution based on the current improvement code that considers such a case.

Appendix A

This appendix presents the complete C-coded version of the implemented code. This is the actual code of the one described in pseudo-code in section 4.3 of chapter 4. As chapter 4 already provides descriptions of the different functions, the sections of this appendix will not provide any further comments on the code.

A.1 The Bandwidth Prediction Function

```
void
bandwidth_prediction(ServSession *pps, ServWindow *win){

  float t1,t2, t3;
  float percent;

  pps->b_k1 = pps->b_k;

  if(pps->delta_k > (tau / 2)){
      pps->delta_k = tau / 2;
      pps->cur_win_bytes = 0;
  }

  pps->b_k = (2 * tau - pps->delta_k) / (2 * tau + pps->delta_k) *
              pps->b_k1 + pps->delta_k * (pps->cur_win_bytes +
              pps->last_win_bytes) / (2 * tau + pps->delta_k);


 if(win->number != 0 && pps->b_k != 0 && pps->b_k1 != 0){

      if(pps->b_k < pps->b_k1){

            t1 = pps->b_k1 - pps->b_k;


            percent = (t1 / pps->b_k1) * 100;

            #if PERCENT
            printf("\n Loss percent: %f % \n", percent);
            #endif

            #if PRINTLOG_TO_FILE
            fprintf(pps->logfile, "\n Loss percent: %f \n", percent);
            #endif

            win->percentage_kind = 0;
            win->percentage = percent;


      }
      else if(pps->b_k > pps->b_k1){
```

```
            t1 = pps->b_k - pps->b_k1;

            percent = (t1 / pps->b_k1) * 100;

            #if PERCENT
            printf("\n Gain percent: %f % \n", percent);
            #endif

            #if PRINTLOG_TO_FILE
            fprintf(pps->logfile, "\n Gain percent: %f \n", percent);
            #endif

            pps->goodcount++;

            win->percentage_kind = 1;
            win->percentage = percent;

      }
      else if(pps->b_k == pps->b_k1){

            win->percentage_kind = 1;
            win->percentage = 0;
      }



 }
 else{
      win->percentage_kind = 0;
      win->percentage = 0;
 }

}
```

A.2 The Written Bytes Update Function

```
void
update_written_bytes(ServSession *pps, ServWindow *win){

  size_t bytes_written;

  bytes_written = pps->bytecount;
  pps->bytecount = 0;


  #if BYTESWRITTEN
  printf("\n Amount bytes written for time interval: %d n",
           bytes_written);
  #endif

  #if PRINTLOG_TO_FILE
  fprintf(pps->logfile, "\n Amount bytes written for time interval: %d
          \n", bytes_written);
  #endif
```

```
  g_get_current_time(&pps->time2);

  timersub(&pps->time2, pps->time1, &pps->timediff);
  pps->difftime = (pps->timediff.tv_sec * 1000000) +
                  pps->timediff.tv_usec;

  pps->time_used = (float) pps->difftime / 1000000;

  pps->last_win_bytes = pps->cur_win_bytes;
  pps->cur_win_bytes = roundf(bytes_written / pps->time_used);

  pps->delta_k = pps->time_used;

}
```

A.3 The Layer Update Function

```
void
update_layer(ServSession *pps, ServWindow *win, gint nr){

  if(nr == 0){

      if(win->number != 0){

       if(win->percentage_kind == & win->percentage >=
          bad_percent_threshold){   //BAD BW

           pps->stablecount = 0;
           pps->goodcount = 0;
           if(pps->stablelayer_incr > 0) pps->stablelayer_incr--;

           if(pps->vid_ahead.tv_sec < ahead_bw_threshold){
             if(pps->layer > bad_layer_threshold) pps->layer =
                                                  pps->layer - 2;
             else if(pps->layer <= bad_layer_threshold && pps->layer
                     != 0) pps->layer = pps->layer - 1;
           }
           else{
             if(pps->layer < 15) pps->layer = pps->layer + 1;
           }
       }    //STABLE BW, BUT OVERALL BAD
       else if(win->percentage_kind == 0 & win->percentage <
               bad_percent_threshold && win->percentage >
               overall_stable_percent_threshold){

            pps->stablecount = 0;
            pps->goodcount = 0;
            if(pps->stablelayer_incr > 0) pps->stablelayer_incr--;

            if(pps->vid_ahead.tv_sec < ahead_bw_threshold){

              if(pps->layer < bad_stable_layer_threshold) pps->layer =
                                                  pps->layer + 1;
```

```
        else pps->layer = pps->layer - 1;
      }
      else{
        if(pps->layer < 15) pps->layer = pps->layer + 1;
      }
}       //STABLE BW
else if(win->percentage_kind == 0 & win->percentage <=
       overall_stable_percent_threshold){

     pps->stablecount++;

     if(pps->vid_ahead.tv_sec < ahead_bw_threshold){
       if(pps->stablecount < stablecount_threshold){
          if(pps->layer < (stable_layer_threshold +
             pps->stablelayer_incr)){ pps->layer =
                                        pps->layer + 1;
          }
          else if(pps->layer > (stable_layer_threshold +
                  pps->stablelayer_incr)){ pps->layer =
                                             pps->layer - 1;
          }
       }
       else{
          if((stable_layer_threshold + pps->stablelayer_incr)
              < 15) pps->stablelayer_incr++;

          pps->stablecount = 0;
       }
     }
     else{
       if(pps->layer < 15) pps->layer = pps->layer + 1;
     }
}      //STABLE BW
else if(win->percentage_kind == 1 & win->percentage <
       better_percent_threshold){

     pps->stablecount++;

     if(pps->vid_ahead.tv_sec < ahead_bw_threshold){
       if(pps->stablecount < stablecount_threshold){
         if(pps->layer < (stable_layer_threshold +
            pps->stablelayer_incr)){ pps->layer =
                                       pps->layer + 1;
         }
         else if(pps->layer > (stable_layer_threshold +
                 pps->stablelayer_incr)){ pps->layer =
                                            pps->layer - 1;
         }
       }
       else{
         if((stable_layer_threshold + pps->stablelayer_incr) <
             15) pps->stablelayer_incr++;

         pps->stablecount = 0;

       }
     }
```

```
            else{
              if(pps->layer < 15) pps->layer = pps->layer + 1;
            }
       }
       else if(win->percentage_kind == 1 && win->percentage>=
              better_percent_threshold && pps->goodcount > 2){

            pps->stablecount++;
            pps->goodcount = 0;

            if(pps->layer < 15) pps->layer = pps->layer + 1;
       }
   }
 }
 else if(nr == 30) pps->layer = 0;
 else if(nr == 40){

   if(pps->layer >= 3) pps->layer = pps->layer / 2;
   else pps->layer = 0;

 }
 else       pps->layer = nr;

}
```

A.4 The Workahead Update Function

```
void
update_video_ahead(ServSession *pps, ServWindow *win, gint mode){

      GTimeVal temp;

      if(mode == 0){

         timersub(&win->xmit_deadline, &pps->global_tvnow, &temp);
         pps->vid_ahead = temp;


         if(temp.tv_sec < 0 || temp.tv_usec < 0){
            pps->vid_ahead.tv_sec =_u48 ?;
            pps->vid_ahead.tv_usec = 0;

            pps->workahead_limit.tv_sec = 0;
            pps->workahead_limit.tv_usec = 0;
         }

         #if WORKAHEAD
         printf("\n In update_video_ahead, vid_ahead: %d \n",
                pps->vid_ahead.tv_sec);
         #endif

        #if PRINTLOG_TO_FILE
        fprintf(pps->logfile, "\n In update_video_ahead, vid_ahead: %d
                \n", pps->vid_ahead.tv_sec);
```

```
        #endif
      }
      else if(mode == 1){ //ONLY FOR WINDOW-TIMEOUT SITUATION
        timersub(&win->xmit_deadline, &pps->global_tvnow, &temp);
        pps->vid_ahead = temp;
      }
}
```

A.5 The Transmission Crisis Check Function

```
gint
check_video_ahead_crisis(ServSession *pps, ServWindow *win){

      GTimeVal t1;
      GTimeVal t2;
      GTimeVal t3;

      g_get_current_time(&win->win_tvnow);

      /* win_start_xmit oppdateres i ss_child_send_win_start */
      timersub(&win->win_tvnow, &win->win_start_xmit, &win->t2);

      timersub(&win->xmit_end, &win->xmit_start, &win->t3);

      if(timercmp(&win->t2, &win->t3, >)){
        return 0;
      }
      else if(timercmp(&win->t2, &win->t3, <)){
        return 1;
      }

}
```

A.6 The Adaptation Function

```
void
adaptation(ServSession *pps, ServWindow *pps_win){

  if(pps_win->xmit_timeout){

   if(pps->vid_ahead.tv_sec < lower_workahead_threshold){

      if((QSF_MAX_PRIORITY - pps->layer) > sdu->priority){
         update_video_ahead(pps, pps_win, 0);
         goto win_done;
      }

   }
   else{

     no_crisis = check_video_ahead_crisis(pps, pps_win);
```

```
      if(no_crisis == 1){

         if((QSF_MAX_PRIORITY - pps->layer) > sdu->priority){
             update_video_ahead(pps, pps_win, 0);
             goto win_done;
         }
      }
      else if(no_crisis == 0){

         #if PRINTLOG_TO_FILE
         fprintf(pps->logfile, "\n CRISIS!! \n");
         #endif

         update_video_ahead(pps, pps_win, 0);

         if(pps->vid_ahead.tv_sec > crisis_workahead_threshold){
             if((QSF_MAX_PRIORITY - pps->layer) > sdu->priority){
                 update_video_ahead(pps, pps_win, 0);
                 goto win_done;
             }
         }
         else{
             printf("CRISIS: NO TIME LEFT FOR THIS WINDOW! \n");

             #if PRINTLOG_TO_FILE
             printf(pps->logfile, "\n CRISIS: NO TIME LEFT FOR
                    THIS WINDOW! \n");
             #endif

             if(QSF_MAX_PRIORITY > sdu->priority){
               update_layer(pps, pps_win, (15 - sdu->priority));
               printf("\n crisis updated layer is: %d \n", pps->layer);

               #if PRINTLOG_TO_FILE
               fprintf(pps->logfile, "\n crisis updated layer is: %d
                       \n", pps->layer);
               #endif

               update_video_ahead(pps, pps_win, 0);
               goto win_done;
             }
           }
         }

        }//slutt else
    } /*slutt my if(pps_win->xmit_timeout)*/
}
```

A.7 The Workahead Transmission Function

```
void
workahead_start_next_win(ServSession *pps){
```

```
        ServWindow *win;
        GTimeVal tvnow;

        win = g_queue_peek_head(pps->mapped_wins);

        if(win == NULL){
                pps->workahead_limit = pps->vid_ahead;
        }
        else {
                g_aio_cancel_timeout(win->start_timeout);
                g_get_current_time(&tvnow);
                ss_win_xmit_start(&tvnow, win);
        }
}
```

A.8 The Initiation Function

```
void
initiation(ServSession *pps, ServWindow *pps_win){

   char wo[30];
   GTimeVal test;

   #if PRINTLOG_TO_FILE
   if(pps_win->number == 0) pps->logfile = fopen("servlog-auto", "a_);
   #endif

   #if WINDOW
   printf("\n");
   printf("\n WINDOW: %d \n", pps_win->number);
   #endif

   #if PRINTLOG_TO_FILE
   fprintf(pps->logfile, "\n");
   fprintf(pps->logfile, "\n WINDOW: %d \n", pps_win->number);
   #endif

   pps->myfile = fopen("text", "r");

   fscanf(pps->myfile, "%s \n", wo);
   #if BW
   printf("\n Current bandwidth from script: %s \n", wo);
   #endif

   #if PRINTLOG_TO_FILE
   fprintf(pps->logfile, "\n Current bandwidth from script: %s \n",
           wo);
   #endif

   pps_win->marked_baselayer_bytes = 0;

   if(pps_win->number == 0) pps->teller = 0;
   pps->teller = pps->teller + 1;
```

```
   if(pps_win->number == 0){
      pps->vid_ahead.tv_sec = 0;
      pps->vid_ahead.tv_usec = 0;
      pps->start_up = 0;
      pps->cur_win_bytes = 0;
      pps->last_win_bytes = 0;
      pps->b_k = 0;
      pps->b_k1 = 0;
      pps->goodcount = 0;
      pps->stablecount = 0;
      pps->stablelayer_incr = 0;

      update_layer(pps, pps_win, 30);
    }

    bandwidth_prediction(pps, pps_win);

    if(pps->vid_ahead.tv_sec >= lower_workahead_threshold){
      pps->start_up = 1;
    }
    else if(pps->vid_ahead.tv_sec < lower_workahead_threshold &&
            pps_win->number != 0){
      pps->start_up = 0;
      update_layer(pps, pps_win, 40);
    }


    if(pps_win->number != 0 && pps->start_up == 1){
        update_layer(pps, pps_win, 0);
    }

    g_get_current_time(&pps_win->win_start_xmit);
    pps->time1 = pps_win->win_start_xmit;

}
```

A.9 Registering the Number of Bytes Written to TCP

```
void
ss_child_send_adu(ServSession *pps)
{
        ServWindow       *pps_win;
        QsfMsg           *msg;
        PpsSdu           *sdu;
        PpsAdu           *adu;
        off_t             offset;
        size_t            count;
        gint              flush;
        gint              may_drop;
        gboolean          is_audio_adu;

              GTimeVal tid1, tid2, tid3;

        g_assert(pps);
```

```
        /*MY CODE*/
        pps->bytecount = pps->bytecount + pps->temp_count_store;

        pps_win = g_queue_peek_head(pps->xmit_wins);
        msg     = heap_min_data(pps_win->sdus);
        QSF_SET_AFTER(msg, sdu);

        if(pps_win->adu_count == sdu->num_adus) {
                    /* We've finished the entire SDU */
                goto sdu_done;
        } else if(!pps_win->frag_num_adus) {

                     /* We finished the current frag. */
                if(ss_arg_drop_mid_sdu &&
                   (!pps_win->xmit_timeout) &&
                   (sdu->priority != QSF_MAX_PRIORITY)) {

                       /* Drop the remaining frags */
                        qsf_log_session(pps->log_session,
                              "win %d: "
                              "drop adus after sending %d of %d",
                              pps_win->number,
                              pps_win->adu_count,
                              sdu->num_adus);
                        goto sdu_done;
                } /* if */
                ss_child_send_frag_start(pps, 0);
        } /* else */

        QSF_SET_AFTER(sdu, adu);

        offset       = adu[pps_win->adu_count].offset;
        count        = adu[pps_win->adu_count].length;
        is_audio_adu = adu[pps_win->adu_count].is_audio;


        /*MY CODE*/
        pps->temp_count_store = count;

        ...
        ...
        ...
}
/* ss_child_send_adu */


void
ss_child_send_sdu_head(ServSession *pps)
{
        QsfMsg     *msg;
        PpsSdu     *sdu;
        ServWindow *pps_win;
        PpsAdu     *adu;
        size_t      total = 0;
```

```
        size_t      limit;
        size_t      count;
        gint        i;

        #if FULL_DATA_DEBUG
        MD5_CTX     md5_ctx;
        #endif /* FULL_DATA_DEBUG */

        g_assert(pps);
        ...
        ...
        ...

        count = sizeof(QsfMsg) + sizeof(PpsSdu) +
                sdu->num_adus * sizeof(PpsAdu);

        /*MY CODE*/
        pps->temp_count_store = count;
        ...
        ...
        ...
}
/* ss_child_send_sdu_head */
```

Appendix B

This appendix provides a sample of the bandwidth scenario scripts that were used in this thesis. Also the code for the spectrum2 filter is provided.

B.1 Sample of a Bandwidth Scenario Script

```
#!/bin/sh

tc qdisc add dev eth0 root handle 1:0 netem delay 100ms

tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 2mbit buffer 10000
limit 10000
      echo "2mbit"
      echo "2mbit" > text
sleep 8

tc qdisc change dev eth0 parent 1:1 handle 10: tbf rate 500kbit buffer
10000 limit 10000
      echo "500kbit"
      echo "500kbit" > text
      sleep 10

for data in 590 680 770 860 950 1040 1130 1220 1310 1400 1490 1580 1670
1760 1850 1940 2030; do
      tc qdisc change dev eth0 parent 1:1 handle 10: tbf rate
      ${data}kbit buffer 10000 limit 10000
            echo "${data}kbit"
            echo "${data}kbit" > text
            sleep 2
done
```

B.2 The Code for Computing the Spectrum2 Value

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

int main()
{
      char wo[10];
      double getdata[65];

      double x;
      int teller = 0;
      int counter = 0;
```

```
  int arraycount = 1;
  double result;
  double newcount = 0;

 FILE *writefile;
 writefile = fopen("specvalue", "a");

 while(scanf("%s \n", wo) != EOF){
      newcount++;

      getdata[counter] = atof(wo);
      counter++;
      arraycount++;
 }


arraycount--;

double h[arraycount];
double z[arraycount];
double average;
double layer_change_avg;

x = 0;

while(teller < arraycount){
    x = x + getdata[teller];
      teller++;
}

average = x / newcount;

teller = 0;
x = 0;
counter = 0;
newcount = 0;

while(teller < arraycount){

    if(teller != 0){
      if(getdata[teller-1] != getdata[teller]){
        if(getdata[teller-1] < getdata[teller]){
             x = x + (getdata[teller] - getdata[teller-1]);
        }
        else x = x + (getdata[teller-1] - getdata[teller]);

        newcount++;
      }
    }

    teller++;

}/*slutt while*/

if(newcount != 0) layer_change_avg = x / newcount;
else layer_change_avg = 0;
```

```
  result = (1 / average) + layer_change_avg;

  printf("Specvalue: %f \n", result);
  fprintf(writefile, "%f \n", result);


}/*slutt main*/
```

Bibliography

[1] Eyal Menin, "Streaming Media Handbook", Upper Saddle River, NJ : Prentice Hall, c2003, ISBN: 0-13-035813-4 (h.)

[2] Steve Mack, "Streaming Media Bible", New York : Hungry Minds, c2002, ISBN: 0-7645-3650-8 (h.)

[3] Peter Symes, "Digital Video Compression", New York : McGraw-Hill, c2004, ISBN 0-07-142487-3 (bok+CD-ROM), 0-07-142494-6 (bok) (h.), 0-07-142495-4 (CD-ROM)

[4] Michael Zink, "Scalable Internet Video-on-Demand Systems", Darmstadt University of Technology, September 2003, pp. 47- 66

[5] P.N Tudor, N.D Wells, "Digital Video Compression: Standardisation of Scalable Coding Schemes", Tadworth: Research and Development Department, Engineering Division, British Broadcasting Corporation, 1994

[6] John Watkinson, ``The MPEG Handbook : MPEG-1, MPEG-2, MPEG-4", Amsterdam : Elsevier, c2004, ISBN: 0-240-80578-x (ib.)

[7] Bob O'Hara, Al Petrick, "The IEEE 802.11 Handbook - A Designer's Companion" , New York : IEEE, 1999, ISBN: 0-7381-1855-9 (h.)

[8] Andrew S. Tanenbaum, "Computer Networks", Prentice Hall International Edition 2003; fourth edition, pp. 292-302, 402-405, 547-550. ISBN 0-13-038488-7

[9] Richard W. Stevens, Gary R. Wright, "TCP/IP Illustrated Volume 1 & 2", Reading, Mass. : Addison-Wesley, c1994-1996

[10] Charles Krasic, "A Framework for Quality-Adaptive Media Streaming: Encode Once – Stream Anywhere", PhD thesis, Oregon Graduate Instituate, chapter 1-5, Portland, OR, USA, February 2004

[11] Charles Krasic, Kang Li, Jonathan Walpole, "The Case for Streaming Multimedia with TCP + Extended Version", Oregon Graduate Institute, Beaverton OR 97206, USA, September 2001

[12] Charles Krasic, Jonathan Walpole, "Priority-Progress Streaming for Quality-Adaptive Multimedia", ACM Multimedia Doctoral Symposium, Ottawa, Canada, October 2001

[13] Charles Krasic, Jonathan Walpole, Wu-chi Feng, "Quality-Adaptive

Media Streaming by Priority Drop", 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2003), June 2003

[14] Luigi Alfredo Grieco, Saverio Mascolo, "End-to-End Bandwidth Estimation for Congestion Control in Packet Networks", Second International Workshop, QoS-IP 2003, section 4, 4.3, Milano, Italy, February 2003.

[15] Belgacem Bouallegue, Ridha Djernal, Hattab Guesmi, Rached Tourki, "A Flow Control Approach and Interleaving Method for Real-Time Application in High-speed Network, section 4.2, Dedicated Systems Magazine, France, 2004