Michael Kropfberger, Dipl.-Ing.

MULTIMEDIA STREAMING OVER BEST EFFORT NETWORKS USING MULTI-LEVEL ADAPTATION AND BUFFER SMOOTHING ALGORITHMS

DISSERTATION

zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften

Universität Klagenfurt
Fakultät für Wirtschaftswissenschaften und Informatik

1. Begutachter:	Univ.-Prof. Dr. Hermann Hellwagner
Institut:	Institut für Informationstechnologie, Universität Klagenfurt
2. Begutachter:	o.Univ.-Prof. Dr. László Böszörményi
Institut:	Institut für Informationstechnologie, Universität Klagenfurt

October 2004

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Schrift verfasst und die mit ihr unmittelbar verbundenen Arbeiten selbst durchgeführt habe. Die in der Schrift verwendete Literatur sowie das Ausmaß der mir im gesamten Arbeitsvorgang gewährten Unterstützung sind ausnahmslos angegeben. Die Schrift ist noch keiner anderen Prüfungsbehörde vorgelegt worden.

Word of Honour

I honestly declare that the thesis at hand and all its directly accompanying work have been done by myself. Permission has been obtained for the use of any copyrighted material appearing in this thesis and all such use is clearly acknowledged. The thesis has not been presented to any other examination board.

Unterschrift: _____

Klagenfurt, 24. Oktober 2004

Contents

List of Tables

List of Figures

Danksagung

Mein Dank gebührt all meinen unschuldigen “Diskussionspartnern”, die sich, trotz denkbar ungeeigneter fachlicher Voraussetzungen, über die letzten Jahre regelmäßig mit diversen Problematiken der Multimediaübertragung im Internet auseinander setzen mussten. Ich hoffe aber, dass ich all meine Freunde, meine Familie und den Rest der Welt von der Sinnhaftigkeit meiner (und vieler anderer Forscher) Arbeit überzeugen konnte und bedanke mich für ihr geduldiges Lauschen und Kopfnicken.

Insbesondere danken möchte ich meiner geliebten Frau Inga, die mir in so manch schwerer Stunde wieder Kraft gab, diese Arbeit weiter zu führen und letztendlich zu beenden. Ich wünsche Ihr einen ebenso erfolgreichen Abschluss der ihrigen und hoffe, ihr auf gleiche Weise beistehen zu können.

Weiters danke ich meinem Betreuer Prof. Hellwagner und meinem Institutsvorstand Prof. Böszörményi, welche mich beide sicher auf dem Weg der wahren Forschung durchs “Welt-Rettungs-Dickicht” führten. Dies als Basis und mit deren Unterstützung startet nun ein hoffentlich ebenso erfolgreiches Projekt, welches sich mit einer Produktüberführung der im Folgenden beschriebenen Dissertationsthemen befasst.

Weitere treue Wegbegleiter waren alle Kollegen am Institut für Informationstechnologie, insbesondere mein ViTooKi-Kollege Peter Schojer, sowie mein äußerst interessierter Bürokollege Mario Taschwer, der ungewollt in einige hitzige Diskussionen verwickelt wurde, aber immer hilfreich zur Seite stand.

Großer Dank und ein edler Schwenk muss natürlich der geheimen Organisation der `Hekkas` zukommen, welche mich durch unzählige kaffeegetränkte Unterausschüsse zu informatisch hochwertigen Themen immer wieder an das Gute im Computer glauben ließ ...

Kurzfassung

Neben den bereits bekannten Netzwerkanwendungen wie das Versenden von E-Mails und das Surfen im Internet, ist eine neue Technologie im Vormarsch: *das Strömen von Multimediadaten.* Doch bevor dieser neue Anwendungsfall sich wirklich im heutigen Internet durchsetzen kann, müssen noch einige Probleme beseitigt werden. Die folgenden Maßnahmen sind absolut notwendig für eine funktionierende Streaming-Umgebung:

- Das gleichmäßige Aussenden von Strömen mit variablen Bitraten, welches unnötige Spitzenbelastungen verhindert.
- Kontrolle des Clientseitigen Pufferfüllstands, um ein ruckelfreies Abspielen der Daten zu gewährleisten.
- Das Erkennen von Paketverlusten und deren Fehlerbehebung bzw. die Veranlassung einer erneuten Übertragung.
- Schnelle und exakte Messung der real vorhandenen Bandbreite.
- Auf kurzfristige und kleinbereichige Bandbreitenvariationen muss schnell und effektiv mit Adaptionsstrategien (wie zB das Verwerfen von Frames) reagiert werden, welche die benötigte Bandbreite verringern, aber trotzdem die visuelle Qualiät nicht übermäßig beeinträchtigen.
- Bei langfristigen und starken Bandbreitenvariationen muss der aktuell aktive Strom durch eine Bitraten-mäßig angepasste Variante ersetzt werden.

Viele Forscher haben die obig genannten Themen bereits erkannt und behandelt. Leider wurden diese immer nur als Einzelprobleme angesehen, und es wurden niemals alle miteinander in einem großen Server-Client Streaming-System eingesetzt und auf deren Zusammenspiel analysiert. Diese Dissertationsarbeit hat sich genau dieses zum Ziel gesetzt und wird die am meisten versprechenden Kombinationen präsentieren und analysieren.

Da die einfache Kommunikation zwischen Multimedia-Servern und Klienten eine der wichtigsten Anforderungen für alle Internetapplikationen darstellt, wurde besonderer Wert auf Standardkonformität gelegt. Als Ergebnis daraus entstand ein frei verfügbares Open-Source-Projekt, welches das Strömen und Adaptieren von Multimediadaten in Internet-ähnlichen Netzwerken unterstützt. Diese einfach zu erweiternde Software nennt sich *ViTooKi – The Video ToolKit* und ist unter `http://vitooki.sourceforge.net` zu finden. Die äusserst flexible Programmierbibliothek repräsentiert ein stabiles Streaming-Testsystem und ermöglicht ein effizientes Strömen von Multimediadaten im Internet. Es vereint alle obig genannten Themen wie gleichmäßiges Aussenden von Daten, sichere Pufferverwaltung, erneutes Senden bei Paketverlusten und schnelle Bandbreitenanpassung durch eine Kombination von fein- und grobgranularen Adaptionsmechanismen. All das immer unter Bedacht der absoluten Standardkonformität, um mit anderen Multimedia-Applikationen auch in Zukunft zusammenarbeiten zu können.

Abstract

In addition to the well-known networking applications like email and browsing the Web, there is a new emerging "killer application": *multimedia streaming*. Before this new use case will be widely available within today's Internet, there are various obstacles which have to be solved first. The measures, which are vital for a well-functioning streaming environment are as follows:

- smoothed streamout of variable bit rate streams, to avoid peaks over the full connection time,
- monitoring the fill state and restrictions of the client buffer, to guarantee jitter-free playback,
- detecting packet loss and reacting with appropriate error correction or retransmission of the packets,
- fast and accurate measuring of the really available bandwidth,
- ways to overcome short-term and small-ranged bandwidth fluctuations by using fast and effective methods of adaptation like frame dropping, which reduces the needed bandwidth without decreasing the perceived quality too much,
- reacting to long-term massive bandwidth changes by completely switching from the active stream to a version encoded for lower bandwidth and quality.

Many researchers have investigated different parts of the above mentioned measures, offering interesting solutions. Unfortunately, those works were always seen as seperate problems, they were never put together for analyzing their coexistence when combined in a full-fledged server-client streaming environment. This thesis wants to combine all of the above mentioned measures by using and evaluating the most promising and performance-wise feasible solutions.

Since inter-connectivity between various multimedia servers and clients is one of the most important requirements for Internet applications, it was absolutely important to be compliant to available standards and to offer well-known and widely accepted ways of communication.

As a result, a freely available open-source client-server environment for multimedia streaming and adaptation for best effort networks is available for world-wide usage and for further extension. It is called *ViTooKi – The Video ToolKit* and is downloadable on `http://vitooki.sourceforge.net`.

The very flexible library incorporates a well-tested and analyzed streaming testbed which meets all requirements of high quality multimedia streaming with respect to best effort networks, combining the above mentioned topics like smoothed streamout and buffer management, packet retransmission and fast bandwidth adjustment using a combination of fine- and coarse-grained adaptation methods, always keeping in mind standard conformance for coexistence with other multimedia applications.

CHAPTER

1 Introduction

Well-known and widely used network applications like email and web services are more and more accompanied by a newly emerging "killer application": *multimedia streaming*. This multimedia streaming will be available on eg. the home computer, where all family members want to watch DVD-quality videos, or on different types of cell phones or handheld devices, where people want to see movie clips or sports scenes.

The video and audio data will not always be available on a local harddisk, but have to be streamed from a connected server somewhere in the network. To offer a wide range of content from many world-wide providers to a huge audience, this network has to be the well-known Internet, which is IP based and imposes various limitations onto multimedia streaming applications. This best effort network, penetrated by thousands of users with varying load patterns, has no means of providing a fixed quality of service, neither in terms of bandwidth nor of latency [1].

It is a fact that today's Internet is put together by many Internet service providers, using lots of cabling and different hardware equipment. Many companies have already taken high investments in this streaming-wise unsatisfying infrastructure, so it will take a long time until better hardware is available world-wide. The same problem is valid for already emerging wireless networks – so called "hot spots" – which even more suffer from ever-changing usage patterns and other external influences like buildings or by-passing cars [2].

1.1 Multimedia Streaming over TCP/IP

Standard networking applications like email or ftp, which only need reliable non-realtime data transmission, are using TCP on top of lossy IP networks. TCP countervails the problems of the underlying network layers by most notably implementing congestion control and guaranteed retransmission. As an example, for a file transfer it is acceptable, that the amount of time used for this transfer is determined by the available bandwidth. So it will take longer to transfer the same amount of data over a low bandwidth link than over a faster connection.

For multimedia streaming, a stream with a certain bandwidth requirement should ideally never suffer from lacking bandwidth. If it is subjected to a sudden bandwidth reduction and no measures are taken to prevent such a case, the client application stops playing and has to refill the buffer again. TCP, when used for multimedia streaming and subjected to packet loss, sacrifices bandwidth for retransmission, even though some retransmitted packets would be late anyway. Further, TCP congestion control reacts with heavy and discontinuous reduction of the available streamout rate, which also wastes available bandwidth. Multimedia streaming environments cannot compensate those heavy and frequent fluctuations over a longer period of time without running out of buffer sooner or later.

Although attempts were made to use TCP/IP for multimedia streaming [3] [4], the widely accepted approach is to use the real-time transport protocol (RTP [5]) on top of UDP/IP, which circumvents most of TCP's unwanted behavior. Using recently introduced extensions on RTP retransmission of lost packets [6] and more immediate RTP feedback about network behavior [7], an intelligent server-client software like the one presented in this work, will overcome some of the Internet's deficiencies without suffering from TCP's rigid behavior.

Further note that using UDP also offers the possibility of sending data in a multicast fashion, which means that multiple clients are connecting to the same stream, which hereby severely decreases the network load. Still, for this work, we want to focus on personalized *video on demand* streaming, so each client can start and pause its stream at will, which is only functional with the unicast scenario.

1.2 Adapting to Changing Bandwidth

As stated above, applications in best effort networks are always exposed to changing bandwidth. So if bandwidth goes down, it is not possible to transmit enough data, so the client buffer will run out of data sooner or later. Since video and audio playback should never stop because of drained buffers, the stream has to be somehow adjusted to the available bandwidth. This can be done by reducing the quality and hereby the needed bandwidth, as long as the bandwidth is too low for the standard data rate.

To do so, video streams encoded with modern codecs offer various means of adaptation. The most widely available and fastest way in terms of computational requirements is to use temporal adaptation, which means to drop some frames. A frame dropping algorithm leveraging modern codecs has to distinguish between different frame types since dropping can only be performed on specially droppable frames. As a favorable side effect, codecs using droppable frames can produce smaller streams by leveraging motion detection and compensation over still images and additionally offer some amount of extra adaptability.

To offer good visual results at the client, it is advisable to drop frames uniformly distributed over time instead of dropping a number of consecutive frames. If the necessary means are available, even results can be improved further with a priori information about the importance of certain frames. By this, only the visually less important ones are dropped.

If available network bandwidth stays low or decreases even more, fine-grained in-stream adaptation methods like frame dropping will reach their limit and the available client buffer is not able to recover but will dry out inevitably. Only coarse-grained adaptation will help to overcome those severe network problems. This means switching to a lower quality or lower resolution video stream encoded at a lower bitrate.

1.3 Variable Bitrate Streams

Modern video codecs use many different optimizations to code a stream at low bitrates while still maintaining a reasonable quality. This includes the already mentioned motion detection and compensation, but also variable bitrate coding. If eg. a video scene is very complex, more bits are needed to code it, but if there is eg. a very

slow panning scene of a panorama landscape, fewer bits per second are needed to code this information. So it is obvious that the bitrate of the coded video changes over time, which is not reflected by a simple average bitrate. For video streaming, it is not advisable to use the average bitrate as the streamout rate, since this will not compensate all the peaks and will lead to buffer underrun. To overcome this problem, a special smoothing buffer and streaming algorithm will be introduced.

1.4 Outline of the Thesis

This thesis discusses the topics of multimedia streaming using RTP over UDP/IP, packet retransmission, adaptation, stream switching and smoothing buffer algorithms. The term *multimedia* in the following work is used interchangeably for a video or audio stream. Although only video streams are thoroughly analyzed in this thesis, modern audio codecs underly the same rules of network packetization, retransmission, switching and buffering and also offer similar types of adaptation, so video and audio streams are treated the same throughout this work. Still, a detailed analysis of audio in a streaming and adaptation context was out of scope of this work.

In Chapter 2, a new algorithm is proposed which analyzes variable bitrate streams in order to avoid buffer underrun. This is achieved by a smoothing buffer approach to calculate hints and the necessary bandwidth for the streaming server.

Whenever the real bandwidth does not provide the needed streamout bandwidth, additional measures like in-stream adaptation have to be taken. Chapter 3 introduces various adaptation techniques, whereas Chapter 4 focusses on temporal adaptation and the optimal preparation process for such temporally adaptable streams. We further come up with frame prioritization schemes for increasing user satisfaction when performing the necessary adaptation.

Chapter 5 discusses the steps that have to be taken, when the bandwidth changes in a coarse-grained manner. Here stream switching to another pre-encoded version is suggested and analyzed for different sources of control (server driven vs. client driven). Client notification about forthcoming switching is described. Special focus is put not only on varying bandwidth but also on the available buffer fill level. Further, a combination of fine-grained in-stream temporal adaptation and coarse-grained stream

switching is proposed, and the results show increased visual quality and client buffer stability with this novel two-dimensional dynamic adaptation.

Chapter 6 presents the standard conformant protocols which are needed to be interoperable with various already available multimedia streaming tools. Those protocols define how to send consecutive frames over the unreliable network (RTP) and to receive feedback on overall packet loss (RTCP). Moreover, ways to communicate and setup multimedia streams (RTSP) with an included description of these streams (SDP) are explained.

To better support the two-dimensional dynamic adaptation mentioned above, Chapter 7 discusses extensions to these basic standards, whereas the first extension introduces more immediate RTCP feedback mechanisms for faster information on the exact frame loss. In combination to this extension, a second one for RTP packet retransmission is analyzed, since retransmission significantly increases the visual quality of the displayed video. This work evaluates the improvements and hereby shows the importance of these extensions for a viable multimedia streaming environment.

All the proposed ideas and methods were implemented in *ViTooKi – The Video ToolKit*, which is available on `http://vitooki.sourceforge.net`. Chapter 8 describes the internal data structures and basic ideas of the ViTooKi video and audio streaming environment, the RTP networking engine, the buffer smoothing and adaptation engine and the stream switching engine. Finally, Chapter 9 comes up with a conclusion, open questions and the future work which can be based on this thesis.

Appendix A offers some ViTooKi implementation details on the proposed extensions and Appendix B lists some useful and convenient tools to access internal statistics information and shows how to add further frame prioritization algorithms.

CHAPTER

2 Smoothed Video Streamout

2.1 Overview

The following chapter discusses modern codecs, which have different bandwidth needs to code different scenes. Further evaluation is done for the aspects of sending such a multimedia stream over the network. Please note, that throughout this work, we will use the terms *multimedia stream* and (as a subset, so called single media) *video stream* exchangably. Modern audio codecs are subjected to the same coding strategies, and hereby can be treated in the same way when buffering and streaming is needed.

MPEG-4 [8] encoders can be configured to generate streams with a (quasi) fixed average bitrate. Coding at a certain constant bitrate means to force lower quality in highly detailed scenes and to raise quality in not so complex scenes. Since this would result in varying visual quality for the user, most MPEG-4 encoders offer no constant bitrate (CBR) but only a variable bitrate (VBR) to code different scenes with the needed bitrate. This finally can be averaged to a specified bitrate. Because for efficiency and quality reasons they even code consecutive frames in variable sizes, according to their motion and their frame type. MPEG-4 uses three frame types, self-standing I-frames for resynchronization, P-frames which reference previous I- or P-frames and finally the smallest ones, the B-frames, which reference in two directions to other I- or P-frames. Please find a more detailed analysis and description of frame types in the next chapter, now we only want to outline the fact of varying frame sizes, which is shown in Figure 2.1.

Different frame sizes also lead to different-sized bitrates for full seconds with 25

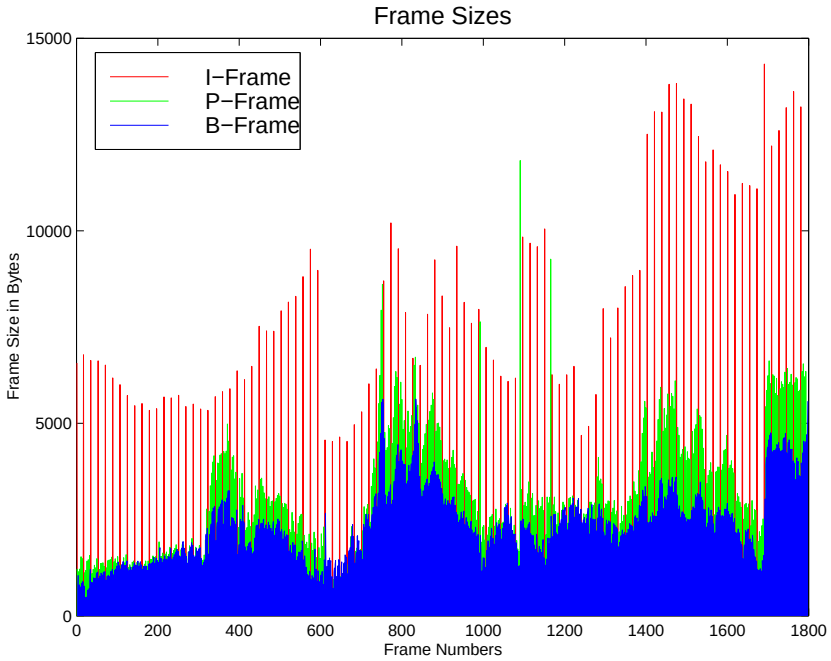


Figure 2.1: Frame size variations

frames per second. So, for example, if a stream has an average bitrate of 512 kbps, every 25 frames of this stream should need about 512 kbps, but this will only work on the overall average.

The first major block of Table 2.1 labelled as *MPEG-4 video* lists a video with four seconds, where each second requires a different bitrate.

If we want to stream out such a video via a network, there are two possibilites. The obvious, but worst, would be to stream out one second of video (so the bitsize $VsecBS_k$ of a given video second $Vsec_k$) in one second of sending time (called one streamout second ($Ssec_k$). This would introduce extreme peaks to the network traffic (see major block two of Table 2.1 labelled *simple streamout*).

To use and smooth the bandwidth better, we try to stream out exactly the average bitsize of all $VsecBS_n$, which leads to a bitrate of 512 kbps. According to the top row shown in Figure 2.2, we can send out all bits of video second one ($VsecBS_1$) in the first stream-out second ($Ssec_1$), since the size of $VsecBS_1$ exactly matches our streamout bandwidth of 512 kbps. Within $Ssec_2$, we could only stream $\frac{2}{3}$ of $VsecBS_2$, where in $Ssec_3$, we could add the remainder of $VsecBS_2$ and the full $VsecBS_3$. In

MPEG-4 video		*simple streamout*		*smoothed streamout*	
second	bitsize [kbits]	bitsize [kbits]	streamed video seconds	bitsize [kbits]	streamed video seconds
1	512	512	$VsecBS_1$	512	$VsecBS_1$
2	768	768	$VsecBS_2$	512	66% of $VsecBS_2$
3	256	256	$VsecBS_3$	512	33% of $VsecBS_2$ +100% of $VsecBS_3$
4	512	512	$VsecBS_4$	512	$VsecBS_4$
$avgBW = 512kbps$					

Table 2.1: Variable bitrate streamed with simple and smoothed approach

$Ssec_4$ we could exactly stream $VsecBS_4$. Find the tabulated overview in the major block three of Table 2.1 labelled *smoothed streamout*.

Figure 2.2 shows the streamout seconds ($Ssec$) on the timeline, and, offset by one second, the playout seconds ($Psec$). We assume that after one second of sending data, the client starts to decode and play out the data which it has received within the previous second.

According to our example, at $Psec_1$ there would be the full block of $VsecBS_1$ available for decoding, but we would already run into decoding problems in playout second two ($Psec_2$), since $Vsec_2$ is not fully available yet in the buffer[1]. At $Psec_3$, we would be able to decode the meanwhile fully arrived data of $VsecBS_2$, but because of already missed presentation timestamps and synchroization issues, we have to totally abandon $Vsec_2$ and directly continue with $Vsec_3$, which is fully available with $VsecBS_3$ kbits in the buffer.

[1]Buffer underrun could be handled at a finer granularity, eg. at the level of frames, but this would introduce extra complexity because of frame dependencies. Still, the same problem of wrong average sending rates would occur. So for simplicity, we assume, that it is necessary for the decoder to receive a full $VsecBS_k$ to successfully decode it. To let this assumption hold, we further assume that every $Vsec$ starts with an I-Frame and is coded as a *closed GOP*, so a group-of-pictures (GOP) is completely independent from other preceding or following $Vsecs$.

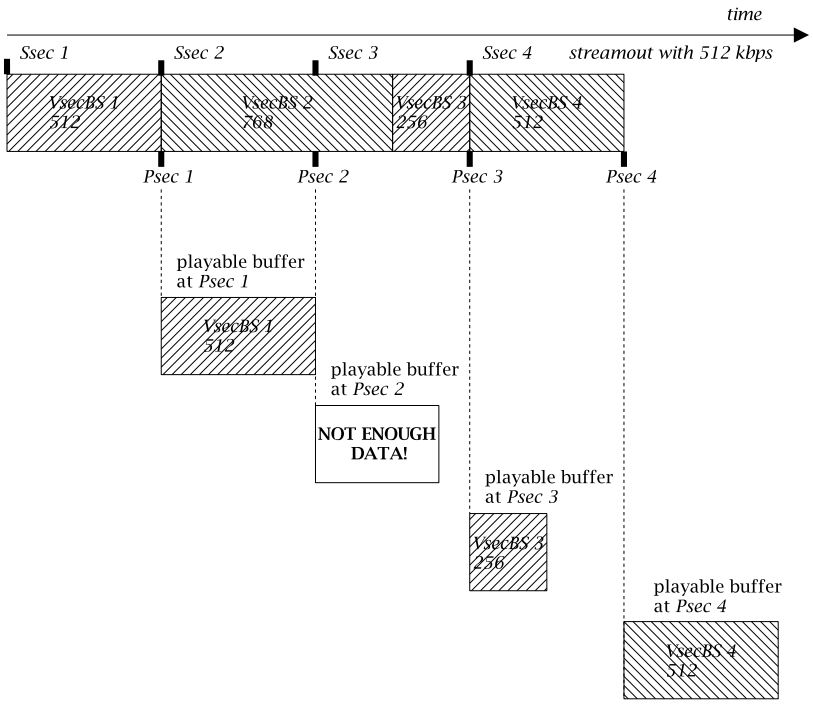


Figure 2.2: Buffer underrun because of low streamout rate (see Table 2.1)

2.2 Techniques for Smoothed Streamout

In the following we will discuss three methods of avoiding buffer underflow. First, this can be achieved by delaying the playback until more data is stored in the buffer, second, the streamout bandwidth can simply be increased, and third, adaptation is used to reduce the stream's overall bandwidth.

2.2.1 Delayed Playback

To make sure that the decoder always has enough data available, we have to introduce some *prefetching time*, which effectively causes a startup latency for the client, before a video is displayed. The receiver side waits until some amount of $Vsecs$ is buffered, before the decoder starts to work on the bits of $Vsec_1$ (see Figure 2.3).

Since a long video might contain large, long and early peaks in the needed bitsize of various $Vsecs$, we might need to prefetch a large number of $Vsecs$ to overcome those problems. This increases the needed client buffer size, which might be problematic

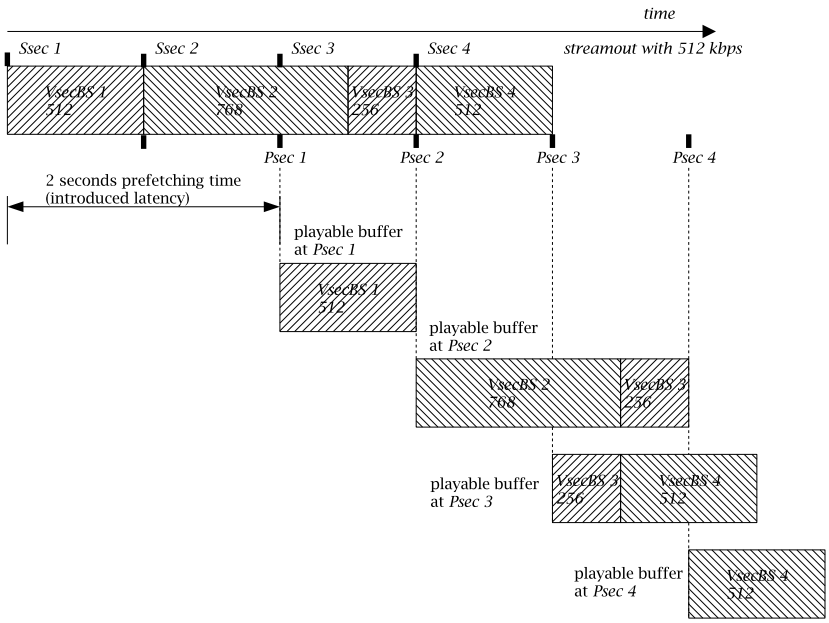


Figure 2.3: Prefetching time avoids buffer underrun

on resource-constrained mobile devices. Furthermore, it introduces a large start-up delay before the first frame of $Vsec_1$ goes into the decoder to be played out. As seen in many publicly available video players like RealPlayer or Microsoft's MediaPlayer, a prefetching delay of about 8 $Ssecs$ is used, assuming that users will not tolerate significantly longer prefetching periods. RealPlayer also uses a higher streamout rate just for this prefetching period, but this results in a more bursty traffic load which could easily lead to congestion. MediaPlayer keeps the streamout rate constant over all time (find a comparison of both players in [9]).

2.2.2 Increased Streamout Bandwidth

Although having this tolerable maximum of 8 streamout secs ($Ssecs$), we still cannot know the needed bitsizes of the first video seconds, so prefetching 8 $Ssecs$ can still result in a very low number of full $Vsecs$, when they have large bandwidth requirements.

To stay within this prefetch boundary and always receive complete $Vsecs$, we have to increase the streamout bandwidth. In Table 2.2 we go up from 512 to 640 kbps.

With this higher streamout bandwidth, no prefetching is needed and there are always enough $Vsecs$ in the client buffer for decoding (see Figure 2.4).

MPEG-4 video		*smoothed streamout*	
second	bitsize [kbits]	bitsize [kbits]	streamed video seconds
1	512	640 (512+256)	100% of $VsecBS_1$ + 33% of $VsecBS_2$
2	768	640 (512+256)	66% of $VsecBS_2$ + 100% of $VsecBS_3$
3	256	640 (512)	100% of $VsecBS_4$
4	512	none	already done...
$avgBW = 640kbps$			

Table 2.2: Smoothed streaming with increased streamout bandwidth

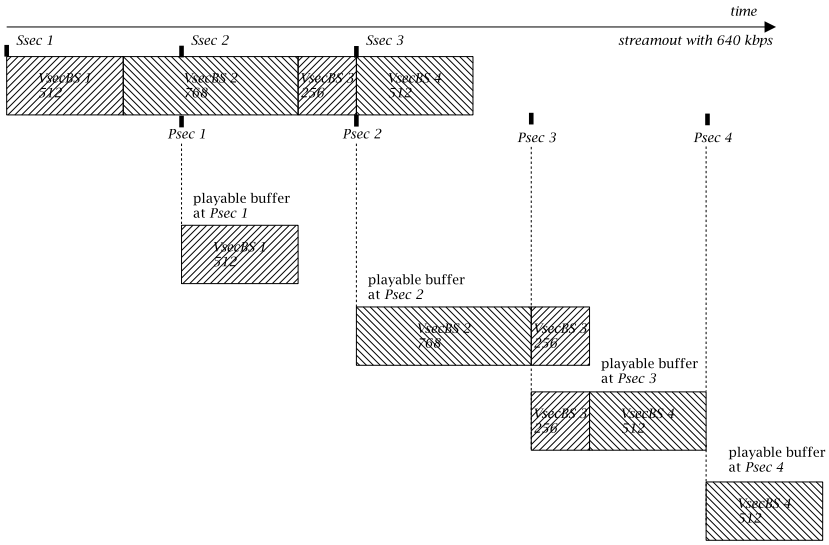


Figure 2.4: Higher streamout bandwidth allows decoding without prefetching

2.2.3 Reduced Bitrate by Adaptation

If we need to keep the 512 kbps streamout bandwidth under all circumstances (eg. the underlying network does not support a higher bandwidth), we have to reduce the overall average bitrate by applying some kind of adaptation to the whole video stream. Hereby we again prevent a client buffer underrun, since all $VsecBSs$ are smaller by a certain percentage.

This is obvious, since if we multiply every $VsecBS_k$ of a video by a certain factor β, also the average is adjusted by the same factor β (eg. $\beta = 0.66$ represents a 33% reduction).

$$avgBW \times \beta = \frac{\sum_{k=1}^{numSec} (VsecBS_k \times \beta)}{numSec}$$

This reduction might happen by re-encoding the stream at a lower quality, or just by dropping some frames or any other adaptation technique available. Table 2.3 shows the smoothed approach on an adapted stream with only 66% of its original size. With this reduction, no prefetching is needed.

MPEG-4 video		*adapted video (66% of original)*	*smoothed streamout*	
second	bitsize [kbits]	bitsize [kbits]	bitsize [kbits]	streamed video seconds
1	512	341	512 (341+171)	100% of $VsecBS_1$ + 33% of $VsecBS_2$
2	768	512	512 (341+171)	66% of $VsecBS_2$ + 100% of $VsecBS_3$
3	256	171	512 (341)	100% of $VsecBS_4$
4	512	341	none	already done...
$avgBW = 512kbps$				

Table 2.3: Smoothed streaming of an adapted stream

2.2.4 The Client Buffer Tunnel

The main goal of smoothed streaming is to avoid draining the buffer, so there is always a $Vsec_n$ at time $Psec_n$ available. Prefetching $Vsecs$ or increasing of $VsecBS$ throughput (either by increasing streamout bandwidth or decreasing the $VsecBS$ by adaptation) means to increase the need of a larger client-side buffer. Especially on memory-constrained mobile devices, there are restrictions on the maximum available client buffer size, so we can only buffer a limited number of $Vsecs$ to overcome longer peaks in the video. For every video, we have to find out a streamout bandwidth, which fits in between the buffer minimum (the completely empty buffer) and the buffer maximum (absolutely full buffer). This safe area is called the *buffer tunnel*.

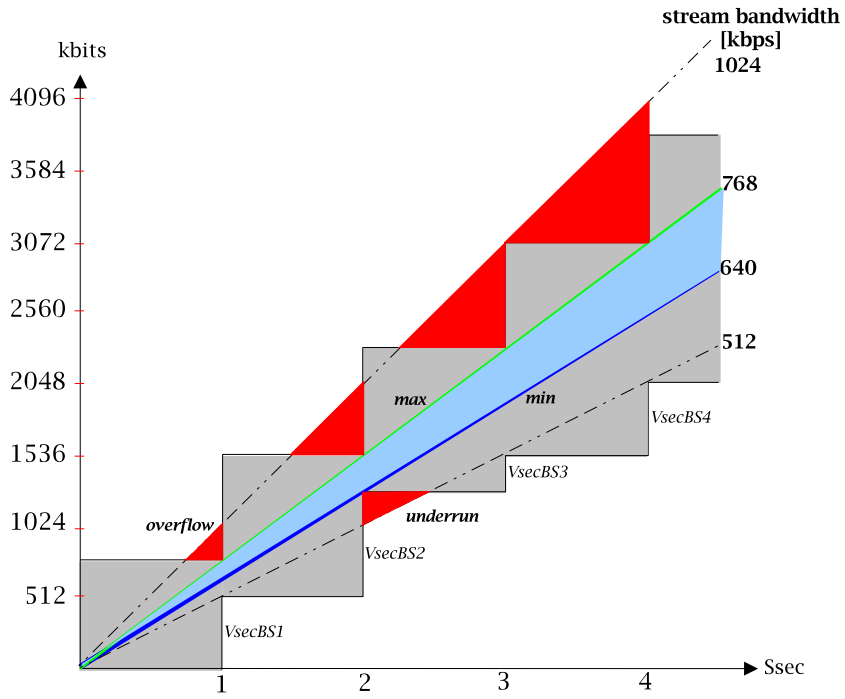


Figure 2.5: Buffer tunnel with different streamout bandwidths

Figure 2.5 illustrates the previous video example with a buffer size of 768 kbits. It shows the minimum stream bandwidth in dark blue (640 kbps), that allows timely decoding of $Vsecs$ and a maximum stream bandwidth in light green (768 kbps), that will never cause overflow to the buffer. So the streamout bandwidth may vary between 640 kbps and 768 kbps (the light blue area) to always stay in the tunnel (the gray area) and hereby always satisfies buffer constraints. If the streamout bandwidth exceeds the maximum stream bandwidth (eg. 1024 kbps) or falls below the minimum (eg. 512 kbps), a buffer overflow or underflow is shown as red triangles.

2.3 Related Work

The area of buffer smoothing algorithms with special respect to variable bitrate of MPEG streams is discussed in many publications eg. [10, 11, 12, 13]. Different algorithms for achieving optimum buffer usage or excessively constant streamout rate are introduced. Also histograms and autocorrelation functions for better smoothing

are derived from VBR streams in [14].

Some of the proposed algorithms also come up with multiple changes of streamout bandwidths over time, which allow even better buffer usage than a single streamout bandwidth for the whole video. Optimizations are done to reduce the number of necessary bandwidth changes or to prevent radical and massive changes to the streamout bandwidth.

All of the previously mentioned attempts on stream smoothing ignore unpredictable changes in network bandwidth due to the underlying best effort networks. They assume guaranteed services, either achieved by special networks like ATM or by intelligent routers in connection with software solutions like the Resource Reservation Protocol (RSVP) [15]. In all cases of re-calculating the changing streamout bandwidths, this always relies on renegotiation of quality of service (QoS) for network streamout, which is not available on best effort networks.

Since there is no way to guarantee a streamout bandwidth over a best effort network like the Internet, all pre-calculated analysis of the video stream can only be a good hint and starting point for intelligent streaming, not the final solution.

When the available network bandwidth decreases and falls below the a priori calculated streamout bandwidth, the streaming system has to react somehow. If the buffer fill level is high enough, it will ignore the bandwidth reduction and will just continue to send out data with lower bandwidth. This approach of modifying the streamout bandwidth according to network feedback has been discussed in [16]. Note that, if high bandwidth reduction is necessary because of congestions, video playout will still soon stop since there are not enough buffered $Vsecs$ in the client side buffer. The biggest downside of the approach of [16] is based on the authors' requirement to always keep the buffer in a medium fill state when the network bandwidth is at an acceptable level, so this results in very high and rapid fluctuations in the adjusted streaming bandwidth which also could distract competing streams like TCP connections.

To better reflect immediate and on demand streaming without complete knowledge of the full video stream (which is not available on eg. intermediate routers or network gateways), the needed streamout bandwidth can be detected in a certain window of time (eg. 10-60 seconds). Here, the streaming bandwidth has to be

recalculated whenever a new $Vsec$ is available in the smoothing engine [17], which increases the processing needs on intermediate nodes. This windowed approach still leads to fluctuations in the calculated streamout bandwidth, but – because of the window – those fluctuations are smoother in comparison to the previosly mentioned approach.

Nevertheless, if there is a forseeable buffer underflow because network bandwidth is too low to fill up buffers in time, we have to introduce new methods of preventing buffer underflow. The best way to do so is to *adapt* the stream *dynamically* by eg. dropping certain frames. [18] describes a way how to drop frames if a $VsecBS$ exceeds the given (but fixed) streamout bandwidth. Unfortunately, the authors do not take into account changing network bandwidths (which is one of the main problems within best effort networks) but assume a fixed (but maybe still too low) bandwidth for the full session time. They are preparing the frame discard scheme in an offline process before the streamout starts.

[19] tries to solve the problem of changing network bandwidth in best effort networks and reacts with frame dropping. To do so, it ignores coding dependencies, which are inherent in modern codecs. This means that *all* frames have to be coded as self-standing I-frames, which heavily increases the overall needed average bandwidth. The authors analyze the video in an offline process which also reorders large-sized frames whithin the stream, so when streaming is performed, frames are sent out of order, which introduces more latency and higher buffer requirements in general.

2.4 The Proposed Static Streamout Analysis Algorithm

To support a streaming server with good hints on the needed streamout rate, the following section describes an algorithm which calculates the exact bandwidth which is needed to keep a variable bitrate stream decodable and to avoid buffer underrun.

Static and a priori analysis of a video stream enables us to come up with the minimally needed streamout bandwidth, also taking advantage of prefetching time. The proposed algorithm also takes into account buffer size restrictions and a maximum amount of acceptable prefetch seconds.

Usually, the needed streamout bandwidth to fulfill all requirements on decodeability and buffer constraints, is noticeably higher than the average bitrate, since bandwidth variations have to be compensated. This is an important discovery, since it shows that it is not enough to stream out a video solely at its average bitrate, but most certainly it has to be streamed out with a higher bandwidth (the measurements on various streams done in this thesis at various average bitrates have shown a necessary increase of approx. 7 - 15% for the average streaming rate to avoid a buffer underrun).

Please note, that this work primarily focusses on today's Internet as a best effort network, so we only want to use statical analysis of variable bitrate streams as a basis for further adaptation hints in a dynamically changing streamout environment. In the following chapters, this offline calculated streamout bandwidth will be combined with intelligent dynamic frame dropping to fullfill buffer constraints even under changing network bandwidth, including viable solutions for packet loss.

Focussing on performance, our proposed algorithm was kept very simple and hereby fast. This simplicity makes it usable in various time-constrained scenarios like a highly frequented video streaming server. Given a pre-encoded video with highly varying $VsecBS_k$ (as shown in Figure 2.6(a)), an average bandwidth requirement of

$$avgBW = \frac{\sum_{k=1}^{numSec} VsecBS_k}{numSec}$$

Further, a given client is offering a certain buffer size and is accepting a configurable prefetching time.

We want to stream out a given video with one constant streamout rate ($streamBW$) for each $Ssec$ to a given client, without breaking client buffer constraints. Please note, that this algorithm is used as a basis, but not as the final solution to overcome bandwidth variations in best effort networks. As discussed in the following chapters, this concept will lead to satisfactory results only in connection with stream adaptation and stream switching.

2.4.1 Detailed Description of the Algorithm

To start the analysis, we initialize $streamBW$ with the average bit rate of the video and set the $prefetching$ time to zero. Within the first streamout second $Ssec_1$, we simulate a streamout of as much of $VsecBS_1$ as $streamBW$ allows. If the client side buffer is not full yet and there is some bandwidth of the $streamBW$ left, we continue with $VsecBS_2$ and so on. If the currently availabe $streamBW$ is used up or the client buffer is full, we have to delay the sending and continue later within $Ssec_2$.

Since we have no $prefetching$ time set, the client immediately starts to decode $Vsec_1$ after receiving the first $Ssec$ worth of data, which is called the playout second $Psec_1$. We deduct the size of $VsecBS_1$ from the client buffer, which virtually represents the decoding of $VsecBS_1$. After that, we enter $Ssec_2$ and continue to stream more video seconds worth of data ($VsecBSs$).

If we run out of $VsecBSs$ in the client side buffer, which means that we could not accurately decode the needed full $VsecBS_i$ at playout second $Psec_i$, we have to increase the $prefetching$ time and start all over again. By increasing the $prefetching$ time, we hopefully will be able to decode the full video from $Vsec_1$ up to $Vsec_n$ (see Section 2.2.1).

If the $prefetching$ time reaches the accepted maximum (eg. 8 $Ssecs$), we have to restart all calculations with an increased streamout bandwidth and with $prefetching$ time reset to zero again. Again, we hope that the increase of bandwidth will allow us to decode all $Vsecs$ of the video (see Section 2.2.2). If the decoding fails even with this higher streamout bandwidth and a zero prefetching time, we will try to increase the $prefetching$ time again until its maximum is reached. Then we will increase the stream bandwidth again. The loop ends with a valid $prefetching$ time and a $streamBW$ that allows accurate decoding of the entire video, or stops after detecting that this stream is not streamable under the given buffer constraints.

Assuming that the client side buffer is reasonably proportional to the average video bandwidth, this algorithm will find a solution. If variations in the video force a very high streamout bandwidth and prefetching, but the buffer cannot hold enough data, there might be no feasible streamout plan. This is not an error in the algorithm, it just shows, that this video *cannot* be displayed without any decoding problems under those buffer circumstances. The only way to find a fitting streamout bandwidth is to

increase the buffer or to reduce the average bandwidth of the video by eg. re-encoding or adaptation (see Section 2.2.3).

2.4.2 Implementation Results

The results of the proposed algorithm are discussed using the "Big Show Both" video with 520 $Vsecs$ (13000 frames), which has an average bitrate of 444 kbps, where $VsecBS$ range from under 200 kbits up to 700 kbits. It was encoded using the Microsoft reference software for MPEG-4 [20], which produces high bitrate variations over various $Vsecs$. The yellow line in Figure 2.6(a) nicely shows this high variation in the size of the actual $VsecBS$.

Our statical analysis calculates a needed streamout bandwidth of 495 kbps with a prefetching time of eight seconds. The average bitrate of 444 kbps is shown by the horizontal green line, the calculated streamout bandwidth is shown as the red line at 495 kbps. The really streamed bandwidth (magenta line) should equal the calculated streamout bandwidth, but since the granularity is on a frame per frame basis, it does not always match it. Anyway, it is always equal or below the calculated streamout bandwidth.

This algorithm for statical analysis was implemented as a C++ program, which takes an MPEG-4 video elementary stream as input.

There are some configurable variables:

- `MAX_PREFETCH_SECS` defaults to 8, but could be set according to the application's needs and/or according to the user's acceptance.
- `INC_FACTOR` is set to 1.01, so if the used prefetching time reaches `MAX_PREFETCH_SECS`, the $streamBW$ is multiplied by this factor and the prefetching time is set to zero again.
- `INC_BUF_FACTOR` is set to 1.05 and is used to climb up to the minimally needed clientside buffer, to get the video through (the exact description follows below).
- `MAX_STREAM_BW` gives an upper limit of the network bandwidth. If the calculated streamout bandwidth is higher than this value, then there is no feasible solution.

- `MAX_BUFFER_SIZE` gives an upper limit of the client side buffer. If the calculated buffer is higher than this value, then there is no feasible solution.

The software comes up with different statistics and draws gnuplot curves to visualize the buffer and streamout behaviour over the video time. There are four different situations/setups which are simulated by the software. These are discussed as follows.

2.4.2.1 Optimum Buffer with Minimum Streamout Bandwidth

In the first setup, the algorithm tries to find an optimum buffer size, so the streamout bandwidth is kept at a minimum and the buffer is never full or empty. Note that, whenever the buffer would be full, the streamout bandwidth has to be severely reduced to not flood the buffer. So basically, eliminating the buffer as a restraining factor favours a lower and constant streamout bandwidth.

The algorithm starts searching with a very small buffer size and gradually increases by `INC_BUF_FACTOR`. For the used test video, we need 16 Mbits of buffer and a streamout bandwidth of 495 kbps with an eight second prefetch (see Figure 2.6(a)). Keep in mind that the average bandwidth of this video was only 444 kbps!

2.4.2.2 Minimum Buffer with Minimum Streamout Bandwidth

In the next setup, we want to keep the already calculated *minimum streamout bandwidth*, but we also want to reduce the buffer to its minimum. To reduce the needed buffer size we accept that the buffer reaches a very high fill level (which results in a reduced streamout rate for the time of high fill level), but of course no buffer underflows. This setup comes up with the optimum buffer and bandwidth combination for this stream, since it requires low network bandwidth and optimally uses the buffer.

The algorithm keeps the minimum (and hereby optimum) streamout bandwidth and starts with a buffer set to zero. From here we gradually increase the buffer until the fixed streamout bandwidth does not endanger the buffer fill level to underflow.

Figure 2.7(b) shows the client buffer, which is restricted to 4854 kbits (607 KBytes). Due to the first eight prefetch $Ssecs$, the buffer is well filled with the fixed streamout bandwidth from before set to 495 kbps. When the decoder starts

playing $Vsec_0$ up to $Vsec_{180}$, which are encoded at bitsizes high above the calculated stream bandwidth, the buffer is drained to nearly its minimum (compare with the high bandwidth variations of the original stream (yellow line) in Figure 2.7(a)). $Vsec_{180}$ to $Vsec_{200}$ are small-sized, so the buffer refills quickly but later drains again. At $Vsec_{300}$, a sequence of small-sized $Vsecs$ allows the buffer to reach its maximum. This results in a reduction of the really streamed out bandwidth, since, to avoid buffer overflow, we can only send as much data as we release from the buffer within each $Vsec$. This is also shown in Figure 2.7(a), where the really streamed bandwidth massively goes down as long as the buffer is full.

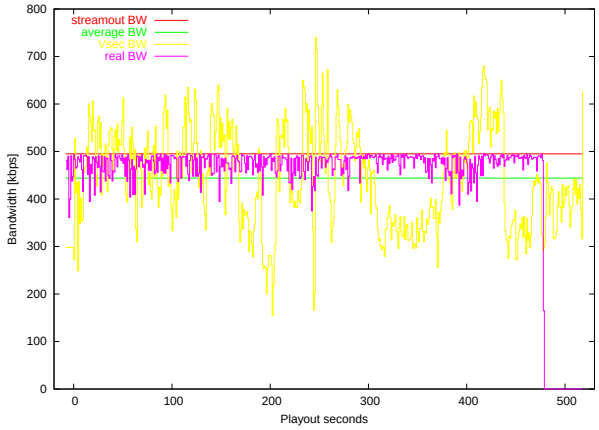
Figure 2.7(c) shows the client-side buffer tunnel with its maximum and minimum. The actual fill grade of the buffer obviously always stays in between those bounds. In close correlation to Figure 2.7(b), at $Vsec_{180}$ we nearly run out of buffer, whereas at $Vsec_{300}$ to $Vsec_{400}$, the buffer is filled to its maximum.

2.4.2.3 Absolute Minimum Buffer with High Streamout Bandwidth

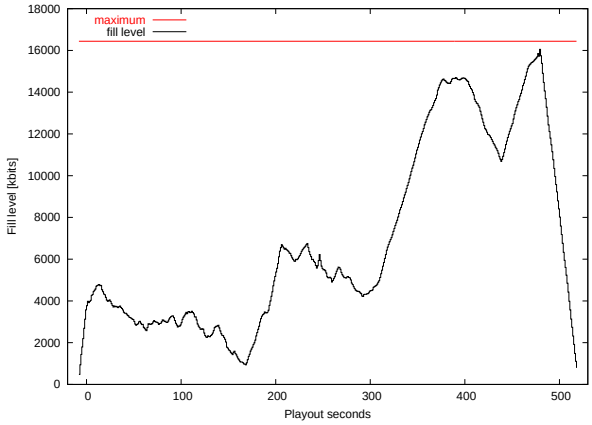
In this setup, the goal is to find the lowest possible client buffer requirement, when streamout bandwidth is not the bottleneck (Note that in the previous section the network was also taken into account). For this, the algorithm starts with a maximum buffer size which equals the average bandwidth of the stream, so for a constant bitrate stream, the buffer would be capable of storing exactly one $VsecBS$ in advance. This obviously will fail for a variable bitrate stream, so the algorithm will have to increase the buffer step by step. First it tries to find out a fitting streamout bandwidth, so the buffer does not underflow. If no solution was found, the buffer is increased by `INC_BUF_FACTOR`. Then a new attempt is initiated, looping until it finds a buffer level and streamout bandwidth, where problem-free streaming is possible. For the test video with an average bandwidth of 444 kbps, a minimum buffer of 760 kbits and a needed streamout bandwidth of 715 kbps with one second of prefetching time was calculated (see Figure 2.8(a)). The test shows that the buffer in Figure 2.8(b) is always at a very high fill level and this forces the needed streamout bandwidth to adjust to the highly peaky bitsize ($VsecBS$) variations. The calculated streamout bandwidth is even higher than the maximum peak of the variations, so this is not desirable for any streaming environment. It just minimizes the needed buffer.

2.4.2.4 Maximum Buffer with High Streamout Bandwidth

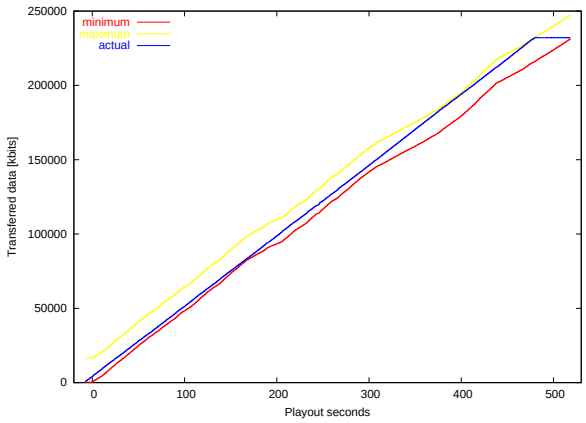
This scenario is not really applicable to real streaming environments, nor does it reflect real buffer scenarios. Still, it nicely simulates a burst scenario in an uncongested network. It just simulates a parameterize and very high `MAX_STREAM_BW` with a huge buffer `MAX_BUFFER_SIZE` and streams the video with this streamout bandwidth. The behaviour nicely shows how the buffer is fastly filled in the first seconds and then keeps full over the video life-time. The streamout rate massively drops when the buffer is full.



(a) Bandwidth variations over time

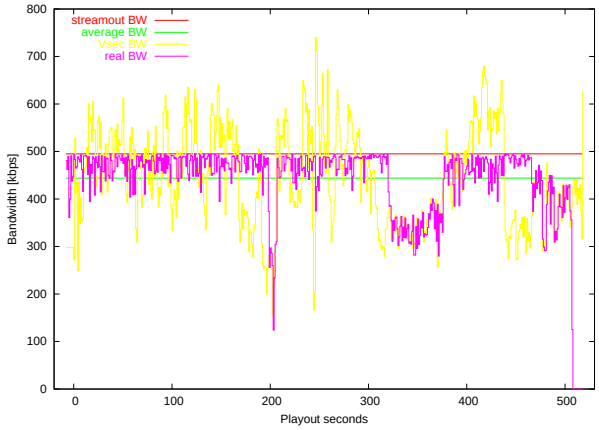


(b) Buffer fill level over time

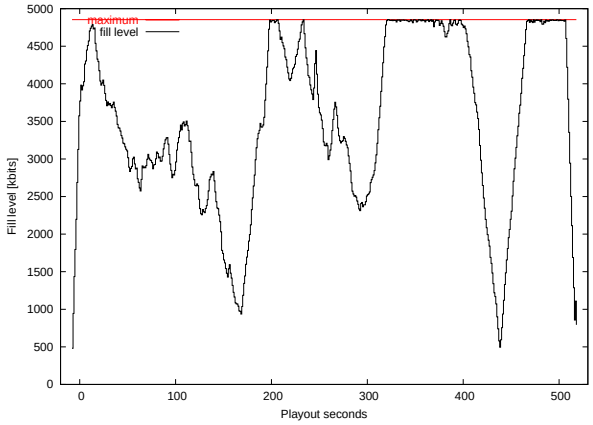


(c) Buffer fill level within the tunnel boundings

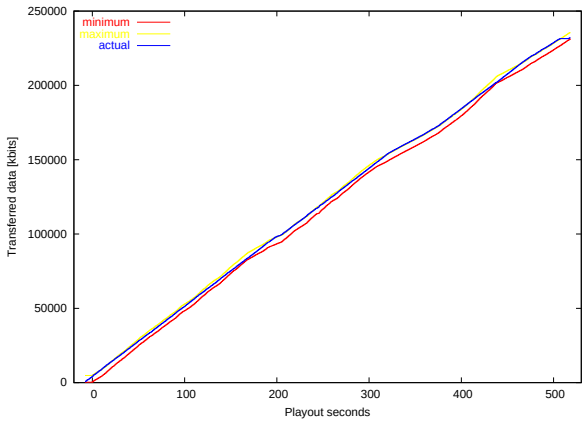
Figure 2.6: Optimum buffer and minimum streamout bandwidth



(a) Bandwidth variations over time

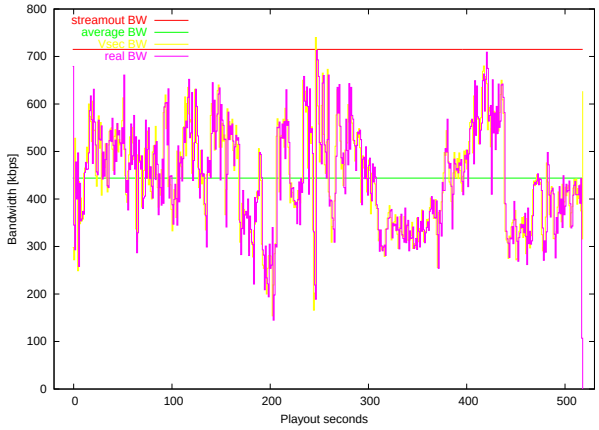


(b) Buffer fill level over time

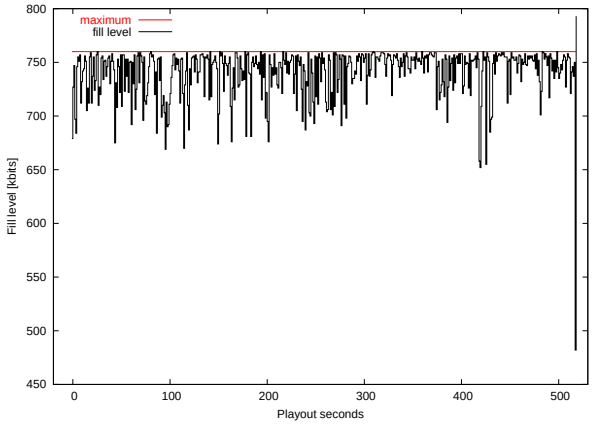


(c) Buffer fill level within the tunnel boundings

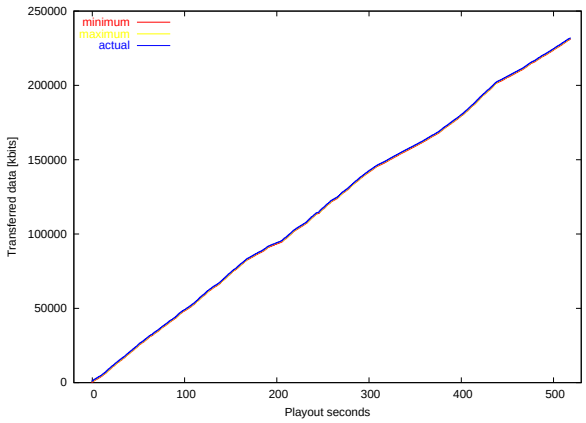
Figure 2.7: Minimum buffer and minimum streamout bandwidth



(a) Bandwidth variations over time

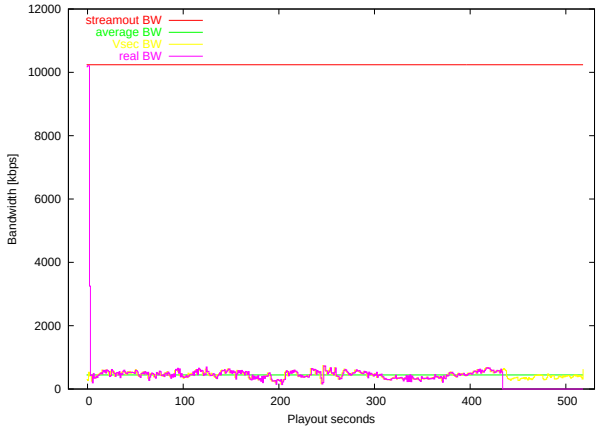


(b) Buffer fill level over time

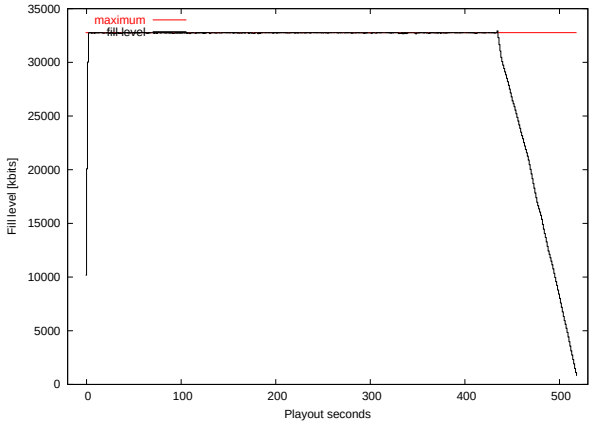


(c) Buffer fill level within the tunnel boundings

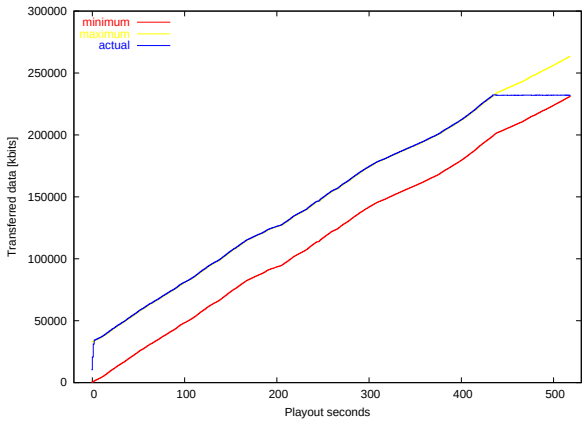
Figure 2.8: Absolute minimum buffer and high streamout bandwidth



(a) Bandwidth variations over time



(b) Buffer fill level over time



(c) Buffer fill level within the tunnel boundings

Figure 2.9: Maximum buffer and high streamout bandwidth

2.5 Conclusion and Future Work

The most important finding of the previous measurements is the fact, that the really needed streamout bandwidth is always higher than a measured average bandwidth (for different videos used within this work, coded at various average bitrates there was an increase of about 7 - 15%). This is triggered by the high $VsecBS$ variances of modern variable bitrate (VBR) codecs and the limited maximum prefetch time, which is acceptable in a certain user application environment.

When the proposed algorithm is used for stabelizing the streamout bandwidth (see Section 2.4.2.1) and optimizing the buffer fill level (see Section 2.4.2.2), the algorithm offers a fast way to calculate the minimum needed $streamBW$ to fulfill minimum client buffer constraints for a certain video stream. This results in a single $streamBW$ for the full video ($Vsec_1 \ldots Vsec_n$) but might also be applied to arbitrary subsequences, resulting in multiple $streamBWs$ for each subsequence.

So the video can be split up in eg. the first half $Vsec_1 \ldots Vsec_i$ and the second half $Vsec_{i+1} \ldots Vsec_n$, resulting in two $streamBWs$. This partitioning can be applied on different video scenes or just because the full video was not yet available in the streamout server or proxy. When used during streamout at an intermediate node, an intermediate buffer can be used as a basis for an estimation of the $streamBW$, which traverses the full video like a sliding window. The larger this buffer is, the more accurate the $streamBW$ can be calculated. This introduces, however, a certain latency, until this buffer is filled for the first time. From then, the data is forwarded in a FIFO (first in - first out) manner, so no more latency is added.

Future work will have to show if this sliding window approach has performance and accuracy impacts on the smooth streamout of a video. The evaluation of this is a very complex task and raises some further issues, which have to be taken into account. Most problematic is the fact that there are many other influences on the actually chosen and available bandwidth such as added congestion control, retransmissions and intelligent buffer management. Those will be described in the later chapters.

CHAPTER

3 Methods of Video Adaptation

3.1 Overview

Every pre-encoded video stream is bound by some input parameters at encoding time. This might be frame rate, SNR quality, or spatial resolution. The used algorithm (codecs like MPEG-4 or ITU-T H.263+) leads to special client-side requirements in buffer memory and processing power. Those requirements are further dependent on used codec optimizations within the stream like high-complexity frames, forward error correction, and enhanced segment separation for improved error resilience.

Sending video data through the network involves routers and different types of underlying networks, which are subjected to different load situations, with different latencies and packet loss rates.

We can overcome network constraints with the right amount of intelligence in the network or by some other means of packet retransmission and correcting mechanisms, so that the data *does* arrive at the terminal (client) side. Still, to be capable of receiving and finally decoding a stream without any errors, we need adequate support on the terminal side, like sufficient buffers and processing power, a fitting display resolution. Even color and frame rate capabilities (LCD might have lower update frequencies) play their role.

We distinguish between the following two *types of discrepancies* between a pre-encoded video and the destination terminal:

Static discrepancies are not changing over the session life time. They can be negotiated once before streaming starts, and so the video stream can be prepared/chosen

to meet the constraints introduced by them.

This includes

- *Terminal capabilities* like color depth, resolution, processing power, available buffer, or maximum display frame rates. Also the relationship between used processing power and available battery life might force a terminal to reduce processing power.
- *User capabilities* like the user is visually impaired, or he is not allowed (willing) to watch scenes with violent content.
- *Network capabilities* like known minimum latency and maximum jitter, given bandwidth constraints by the underlying medium.

Dynamic discrepancies will occur at any time during a session. This includes some degradation of one capability (like bandwidth) at some point in time, but also the strengthening of this capability back to the value that was proposed in the statical analysis.

This includes

- *Terminal capabilities* like the (handheld) terminal moves into glaring light or into darkness. Therefore the overall brightness/contrast of the video has to be adjusted. This could be done in the local hardware, but this would also decrease available processing power for decoding and also would consume more battery. Further, the display backlight will have to be turned on, which reduces battery life even further. As a long-term consequence, this will indirectly enforce a power-saving mode with less available processing power.
- *User capabilities* like the user prefers higher (more detailed) quality over more fluent frame rate for the next scene.
- *Network capabilities* like changes in the available bandwidth on best effort networks, caused by other cross traffic or temporal outages. Further, dynamic calculations could show that, for actual bandwidth fluctuations, the terminal will not provide a sufficient buffer fill level.

Taking into account all those obstacles, it is fairly reasonable to search for methods to perfectly fit a given (and ever-changing) video stream to a given capabilities scenario.

In the simplest case, this can be done by preparation of multiple streams with adjusted settings, which hopefully cover all possible variations. This obviously takes extra time and harddisk space, which should also be taken into account. Further, this hardly can be used to cope with dynamically appearing discrepancies.

To be resistant to either static or dynamic discrepancies, we are looking for methods, which perform some kind of *adaptation* on a video stream, so that it fits under the new constraints.

If processing power is not an issue, the video could be transcoded on the fly. So on the sender side, the by-passing frames are decoded, then adapted in the uncompressed domain, and are finally re-encoded with adjusted settings. By that, the sender could optimally fit the terminals' needs, at the cost of extra processing power, higher buffer requirements and added latency.

But even with always bigger and faster hardware on the market, whenever the amount of parallel transcodings is increased, all available resources will be used to full capacity again. Further, also the requested quality of multimedia presentation is growing in the future, so where people accept TV-quality today, the future requirement will be DVD-like quality, which means higher resolutions and hereby higher computational needs.

Scalable codecs try to enhance a pre-encoded stream by some means, so that highly efficient adaptation can be conducted on this stream. *Highly efficient* in this context means, that adaptation has to happen in the compressed domain with low processing and buffering requirements and nearly no added latency.

Unfortunately there are some static and dynamic discrepancies, which were not solved within a scalable codec yet, so they have to be resolved by transcoding. Open research topics on scalable codecs for the compressed domain are e.g. streams supporting visually impaired/normal-sighted people or fast adjustments in the color histogram.

In this chapter, we discuss *scalable codecs*. To read more about pre-encoded streams at different encoder settings, and how this still could be used to adjust to

different capabilities by *stream switching*, see Chapter 5.

3.2 Scalable Codecs

The following will give a quick overview of well-known types of scalability. Those scalability methods might be used to compensate either static or dynamic discrepancies between the pre-encoded video and the destination terminal. A more detailed overview is given in [21, 22, 23].

3.2.1 Temporal Scalability

The simplest and therefore most widely practiced method is to reduce the number of frames per second.

By dropping a frame at the server side, and hereby not sending it, we immediately decrease the needed bandwidth by the frames' size. Further, the destination terminal does not have to decode the unsent frames either. Frame dropping is only penalized with a more jerky visual experience. Increased visual experience can be provided by introducing special temporal filters at the client side decoder, which interpolates intermediate frames, like RealPlayer does for very low bit rates [24].

If the video is encoded with eg. Motion-JPEG, each frame is decodable on its own and totally independent from any other frame. Therefore those so-called *intra-coded* frames can be dropped arbitrarily.

Modern and more efficient video codecs like MPEG-4 or H.263+ use more sophisticated methods like *inter-coded* frames, which therefore have remarkably smaller bitrates for the same quality.

The following will only give a short introduction into different frame types, so for more details on MPEG-4 codec techniques and intra/inter-frame coding, refer to [25, 23].

Reference intra frames (so called I-frames) are intra-coded frames like in Motion-JPEG, and they can be decoded without any other information. Further predicted frames (P-frames) are referencing to the previous I-frames or P-frames. They save space and only store visual differences (deltas) or even only motion vectors of some picture regions, which have moved their position (think of a bouncing ball). A P-frame

is not decodable without the previous reference frame, but hereby is substantially reduced in its data size.

Obviously, deleting either a P-frame or an I-frame renders the video undecodable (or will cause at least severe display errors), until the next self-standing I-frame appears in the stream.

To further reduce frame sizes, bi-directionally predicted B-frames were introduced to diverse video codecs. B-frames are using the same coding ideas as P-frames, but can refer to both forward and backward frames, so their chances to find similarities or movement are even higher. But, and this fact makes them so interesting for adaptation, B-frames are never using other B-frames as references, but are always only taking into account P-frames or I-frames as reference points.

See Figure 3.1 for a visualization of the frame type dependencies.

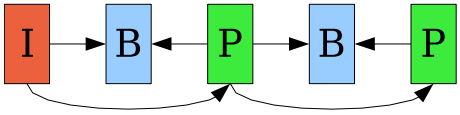


Figure 3.1: Frame types and their dependencies in MPEG-4

3.2.2 SNR Scalability

The quality of a still image can be measured by its Signal-to-Noise Ratio (SNR). Everything within the image, which does not correlate with the original image, is called “Noise”. Still, to reduce data rates, every lossy codec introduces discrepancies between the original and the result image. The SN-Ratio decreases (so the signal converges more and more to noise) with the level of compression, which obviously introduces information loss.

A *bit-rate reduction* can only be achieved by decreasing the quality and hereby decreasing the Signal-to-Noise Ratio. This might be necessary in terms of processing needs, but mainly because of storage/streaming bandwidth needs.

The easiest but most expensive way is again transcoding. But instead of completely do *cascaded full decoding/re-encoding*, work has been done to increase SNR transcoding efficiency by re-using already calculated motion data [26, 27, 28].

The MPEG-4 Advanced Scalable Profile [29] allows the separation of the encoded video into two streams: a base layer for low quality and an optional enhancement layer with the high quality information (see Figure 3.2). The base layer is decodable on its own, but the enhancement layer is only usable with the base layer at hand.

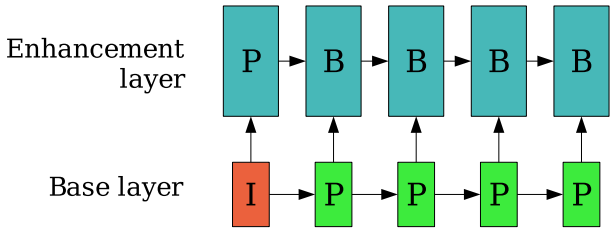


Figure 3.2: MPEG-4 scalable profile provides a base and an enhancement layer

Fine grained SNR scalability can be provided by a *progressive video codec*, which means that the displayed quality can be refined step by step, depending on the amount of already available data. Like illustrated in Figure 3.3, the enhancement layer can be truncated at any position, and the remaining data is used to get the best possible quality.

Figure 3.3: The enhancement layer can be truncated arbitrarily

3.2.2.1 Wavelet Transformation

Wavelet transformation is an advanced method for SNR scalability. Wavelet-based codecs are progressive and, contrary to the Discrete Cosine Transformation (DCT), they use the so-called Discrete Wavelet Transformation (DWT), which allows a higher compression ratio and better bitrate adjustment for still images. The newest JPEG 2000 standard by the ISO/IEC JTC1/SC29/WG1 (JPEG) committee is completely based on wavelet techniques, and highly outperforms the older JPEG algorithm on quality and size.

MPEG-1/2/4 codecs use JPEG-based DCT, but independent attempts were made to exchange all JPEG parts with the newer JPEG 2000 standard [30].

Also the official ISO/IEC JTC1/SC29/WG11 (MPEG) committee has installed a video subgroup, which, categorized as MPEG-21 Part 13, is working on standard-conformant integration of wavelet-based (and other) scalable video codecs for MPEG-4.

An alternative video codec, fully based on wavelet technology, and highly optimized for wireless networks, is presented in [31].

3.2.2.2 Fine Granularity Scalability (FGS) in MPEG-4

Fine Granularity Scalability (FGS), which was introduced four years ago to the MPEG-4 standard, also offers progressiveness in the enhancement layer, so it can be truncated at any arbitrary position and the FGS decoder will use the remaining bits for decoding (see Figure 3.3). But for coding simplicity, the MPEG committee decided on incrementally stored DCT coefficients, organized in multiple bit-planes, instead of wavelet coding. The more bit-planes are received, the more quantization errors between the original stream and the base layer can be eliminated [32].

Those (normally seven) bit-planes are concatenated together and can be seen as one single enhancement layer. The lower bit-planes are smaller but bring a higher gain in quality [33].

Figure 3.4 shows I, P and B-frames with low quality base layers and their enhancement layers. These enhancement layers only use each corresponding frame's base layer for prediction, so the enhancement layer can be dropped or arbitrarily truncated on a per-frame basis.

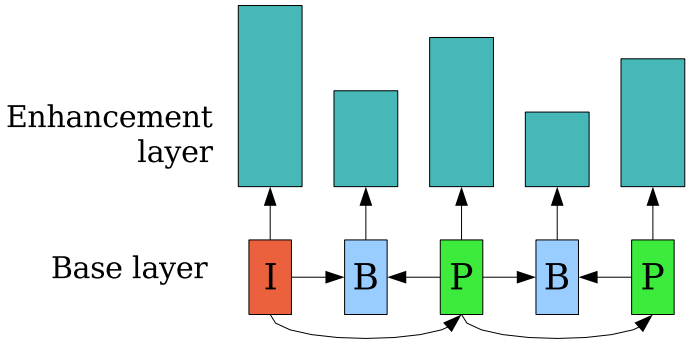


Figure 3.4: FGS frames with their base and enhancement layer

Though arbitrary truncating is allowed, better results can be gained, when the cuts are closer to bit-plane boundaries [34]. This motivates the question, how fine granular FGS really is. To take advantage of both the knowledge of better results at boundaries and the possibility of cutting everywhere, close cuts at boundaries should be favoured but, if the bit-plane boundaries are too far off, a cut inbetween can be performed.

The decision taking on the cutting point could be implemented with some weighting on bit-plane boundaries, which prefers a cutting point alignment to close boundaries.

It is known, that FGS sacrifices display quality up to 2 dB PSNR, if compared to a non-scalable codec at a given bandwidth [32]. Still, trying to ignore a progressive codec as FGS, and falling back to a simple solution with storing a feasible number of multiple versions of a non-scalable video, will fail, because those versions will hardly exactly fit the ever-changing network conditions at all times.

Nevertheless, the issue of coding overhead is highly discussed in the MPEG consortium and there are proposals on various extensions to standard FGS to reach comparability to non-scalable codecs in terms of coding efficiency, eg. by adding motion compensation [35] or implementing motion compensation from prior reference enhancement layers [36].

There is a slow implementation available by the MoMuSys Project [37], and some not publicly available real-time encoders/decoders are emerging in Microsofts research labs.

3.2.3 Spatial Scalability

Adjusting a video to a terminal's given maximum resolution could be done by receiving *any* resolution video and doing the up-scaling or down-scaling after decoding the video. This obviously consumes processing power and (in the down-scaling scenario) wastes network bandwidth for unneeded data.

Scaling a picture resolution by half in both axes could reduce encoded frame size and processing needs close to a factor of four.

The MPEG-4 Advanced Scalable Profile [29] also supports spatial scalability with two layers: a base layer for the low resolution and an optional enhancement layer

with the high resolution information (see Figure 3.2).

Besides that, other approaches have been made to allow spatial scalability. [38] describes a scheme to extend two hardware implementations of non-scalable video codec profiles (like MPEG-2 or MPEG-4 simple profile) with added support for spatial base and enhancement layers. This allows cost-efficient introduction of spatial scalability to already available settop boxes, and is within comparable qualitative results.

Nevertheless, this rigid scheme with a base and an enhancement layer only provides two different resolutions, which might not fit the given terminal. To fit any display resolution, more fine-grained spatial scalability is needed. This could be achieved by a proposed extension to MPEG-4 FGS on hybrid spatial and SNR scalability [39] (see Figure 3.4).

High-quality Picture Scaling To generate the lower resolution base layer, any of the previously mentioned codecs has to do down-sampling of the original video in the uncompressed domain. Also, if on-the-fly transcoding is chosen, CPU-intensive rescaling has to be done.

If the network is only capable of a base layer in a low resolution, but the terminal could provide a higher one, the terminal has to do up-sampling of the already decoded video. Various interpolation methods can greatly improve displaying quality by eliminating blocking effects. Taking up this idea, it could be used as a kind of scalable codec on its own.

Whenever image scaling is performed, different algorithms can be applied. According to [40], scaling could be addressed by the following methods (see Figure 3.6 for visual comparison (images taken from [40])), which vary in performance and quality issues:

- *Nearest neighbour interpolation*, which is the computationally fastest but results in the most distorted scaled image. In the upscaling case, it is simply duplicating one source pixel to the (upscaled) destination pixels. So an $X \times Y$ upscaling by a factor of 2 results in duplicating one source pixel to four destination pixels of the same value (see Figure 3.5). When performing downscaling by eg. factor 2, just one pixel out of four source pixels is chosen, ignoring the other three. The

pixel position is determined by the following formula:

$$newImage[x, y] = oldImage[round(x \times scaleFactor), round(y \times scaleFactor)]$$

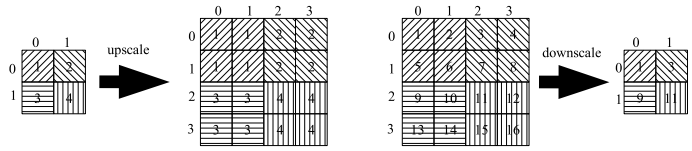


Figure 3.5: Nearest neighbour interpolation for upscaling and downscaling

- *Bilinear interpolation* calculates a linear weighted sum of the four closest neighboring pixels.
- *Bicubic interpolation* uses polynomials of higher powers and B-splines. Despite higher computational needs, it blurs/smears the picture, so it should only be used for upscaling.
- *Martinez-Lim interpolation* is based on a line-shift model for de-interlacing PAL or NTSC transmissions. It uses interpolated in-between lines, to receive even better scaling results.
- *Frequency-domain techniques* can be used in the compressed domain, more precisely in the DCT frequency domain. Arbitrary rescaling purely in the compressed domain is hard to implement because of fixed block sizes like 8x8 or 16x16. Still, if those limitations of block sizes are acceptable, highly efficient and distortion-free conversion is possible to multiples of 2.

Choosing the right interpolation method with the highest possible quality within given computational bounds will be important for efficient *stream switching*. This enables switching down to low resolution stream versions and displaying them in an upscaled manner (see Section 5). Unfortunately, only *nearest neighbourhood*, which gives worst results, is feasible in real-time and in pure software, *DCT domain resizing* is not supported by available MPEG-4 decoders, and only some modern graphics cards offer *bilinear* interpolation support. Taking into account this drawback, in further chapters, scaling is always performed with *nearest neighbour interpolation*.

(a) Original frame

(b) Nearest neighbour interpolation

(c) Bilinear interpolation

(d) Bicubic interpolation

(e) Martinez-Lim interpolation

(f) DCT domain resizing

Figure 3.6: Interpolation method comparison for image downscaling

3.2.4 Object Based Scalability

Superior to other video codecs, MPEG-4 not only offers support for one single rectangular video, but also for arbitrarily shaped videos [41]. It also allows the composition of a scene out of multiple object streams [42], so it is possible to have a rectangular background video with a bouncing ball as a shaped object stream and further, two persons encoded as two separate object streams.

Shaped objects are encoded with a given alpha mask for the whole image, where every pixel is accompanied by an alpha value in the range from zero to 255. This allows fine-grained steps from completely opaque (alpha value is zero), over various steps of shadiness up to total transparency (alpha value is 255).

The creation of video content with multiple video objects could either happen during:

- *Pre-Production* at the studios, where foreground objects like persons are filmed in front of blue screens and are then overlayed with the background, or
- *Post-Production*, with different algorithms which try to extract and separate the different visible objects. Though this is in the early stages of research, there are some interesting attempts. [43] exactly separates moving objects from a static background, which is especially useful for surveillance cameras. [44] detects moving rectangular areas, even if the background is also moving. This could eg. separate a car from its background in a typical car chase scene.

If the video content is composed of multiple streams, it is easy to gain scalability by just dropping the less important streams like the background [45]. To identify the importance of a stream, manually added metadata stored in MPEG-7 [46] could be used.

3.2.5 Region of Interest Scalability

If full dropping of a single stream is not needed (or no separated background and foreground streams are available), different methods of scalability could be applied to single streams, varying for certain regions. For example, we could increase the visual

quality of the important foreground speaker (eg. lower quantization of the according macro blocks) and decrease the quality of the surrounding background.

This last example uses a well-known phenomenon of the human vison: only about two degrees in our about 140 degrees vision span has sharp vision [47]. If an encoder would know, where the watching person is looking at, it could increase the quality in the *region of interest* by penalizing the background areas. In a real-time environment for wide-screen video, a head-mounted eye-tracker could be used to extract the gaze area [48]. For off-line preparation, a less personalized algorithm could be used. If the region of interest is rather obvious (eg. for video telephony or a news speaker), the encoder could be combined with a face recognition system and then focus on this area [49].

3.2.6 Complexity Scalability

If a video stream has to be decoded on a weak client terminal, there some restrictions on available processing power or memory may apply. Scalability could be used to reduce complexity for weak terminals.

One example is the decoding of B-frames, including forward- and backward-referencing for detected motion. This implies, that all three affected frames have to be available in the decoder's buffer, which increases memory needs. Further, this implies a reordering of the frames when they are sent out. So if the *display order* of a video sequence is `IBPB...`, the frames have to be reordered to *network order*, which is `IPBPB...` (see Figure 3.7).

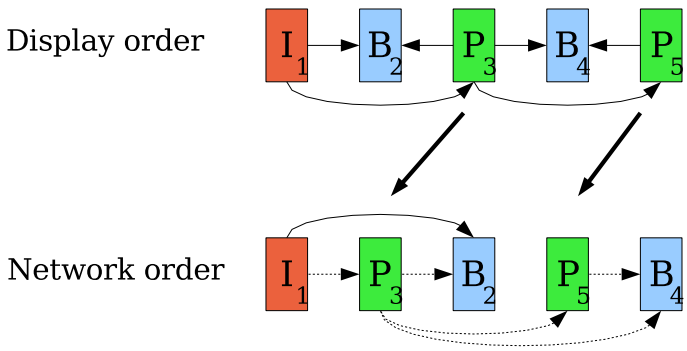


Figure 3.7: Reference frames are sent before referencing frames

When frames are sent over the network, each packet arrives with a certain latency. After reordering frames, decoding of a B-frame will be delayed until all affected reference frames have been received at the client terminal. This might conflict with real-time streaming or render impossible on very slow and high-latency networks.

The more references are allowed (eg. MPEG-4 AVC/H.264 can use up to 5 reference frames for one B-frame), the more buffer is occupied and the more latency is introduced by network reordering.

Complexity scalability enforces either a reduced encoded framerate or a re-encoding of B-frames into P-frames, since P-frames require less buffer and computational needs. Further, scalable codecs with enhancement layers (spatial, SNR, fine-granular SNR) always will demand higher processing power. Therefore, those enhancement layers should be dropped in advance, before being sent out.

3.3 Combined Usage

3.3.1 The Universal Scalable Video Codec

Universal scalable video coding is an attempt to build a scalable codec where fine granular spatial, temporal and SNR scalabilities are integrated together to provide ubiquitous video access on any device through any network.

First steps into that direction have been done with proposals on *hybrid fine-grained temporal and SNR scalability*, which not only introduces B-frames to the base layer, but also adds an extra FGS enhancement layer between two base layer frames, which hereby can be interpreted as intermediary B-frames [50]. Further proposals were made for the *hybrid fine-grained spatial and SNR salability* codec [39].

Combining those two approaches enables scalability in the spatial, temporal and SNR domain with MPEG-4 FGS [51]. Adaptation by various FGS codecs is evaluated with the FGS-based streaming testbed [52], which is part of the MPEG-4 standardization process. Still, severe coding overhead exists and further research has to be done in this direction.

Assuming this universal scalable video codec will exist with good performance in the future, another work analyzes the multi-criteria optimization problem which arises, when more than one way of adaptation is possible within the same stream [53].

3.3.2 Chaining of Multiple Codecs

As long as the above mentioned *universal scalable video coding* based on FGS will not overcome its problems in coding overhead and this codec is not available to the public, other solutions have to be applied. Further, much more adaptation is possible, than just in the temporal, spatial or SNR domain. There could be the need to add digital watermarks and encryption, specially configured for a certain receiving terminal with its unique key. Driven by *MPEG-21 user capabilities [54]*, it could be possible to drop certain, eg. violent scenes [55], which were first described in MPEG-7 metadata [46].

A given scalability framework (eg. the FGS testbed [52], or ViTooKi [56]), which provides a client with an adjusted video stream, can access several different scaling methods, so called *adaptors*, which might also be chained to receive a multi-dimensional adaptation (eg. temporal and spatial). Those *adaptation engines* have to be hinted by a client's capabilities and quality of service (QoS) information [33].

Figure 3.8 shows, how an input stream is adapted by various consecutive adaptors.

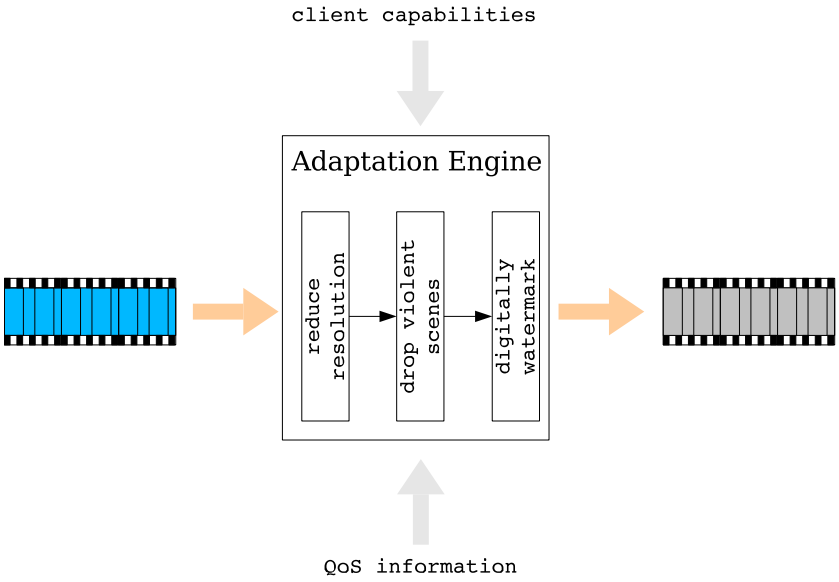


Figure 3.8: Adaptation by a chain of adaptor plugins

ViTooKi – The Video ToolKit [56] incorporates such an adaptation engine and is presented in this work later in Chapter 8. It is based on ideas discussed in [57, 58], which outline a framework for adaptive proxy-caching of MPEG-4 videos.

Decisions on feasibility, necessity and the ordering of different adaptor plugins have to be taken by mathematical rules or long-term heuristics [59]. Some work has been done to feed a Prolog-based planner with descriptions of the input and output stream properties, along with descriptions of many available adaptor plugins. This planner then returns a chain of adaptation steps, which are always valid. It is possible to use different heuristics to achieve optimal results in various domains like computational needs or quality [60].

3.3.3 User-Centric Adaptation

Most of the above mentioned approaches try to find the optimum solution of needed adaptation solely on mathematical or (in the best case) personal heuristics. Superior user acceptance can be reached, if also subjective measurements are taken into account, which give a better hint on human preferences. This especially helps to prioritize adaptation combinations within adaptation domains like temporal or spatial adaptation.

3.3.3.1 Subjective Tests

[61] has incorporated the subjective measurements of human users into an adaptation engine. People were allowed to rank combinations of SNR quantization, frame dropping and spatial resolution with marks between 1 (poor) and 5 (excellent). Those results where evaluated using the *mean opinion score (MOS)*. The engine uses the results to offer better combinations of scalability to fulfill a given bandwidth constraint. Problems in the conducted tests from [61] are the small number of combinations and the extreme scalability settings. Eg. the resolution might be 640x480 or 320x240, with either 30 or 10 fps. Given this, a user might only decide between very large but choppy displaying or small with fast displaying.

Still, those results correspond with other work, which performed some user tests on various video genres with changing frame rates and spatial resolution [62].

3.3.3.2 Monetary Issues on Quality

Adaptation is not only necessary because of hardware or network constraints, but could also be used to incorporate differentiated pricing schemes for multimedia content. For preview purposes, a low quality base layer could be sent out freely, but each extra step in enhanced quality is combined with increasing payments.

Subjective user tests with given budgets showed, that people prefer predictable and consistent lower quality over higher quality with fluctuations. Further, their results suggested that, when users accept a pricing scheme they develop strategies to optimize their use of a limited resource [63].

Also MPEG-21 offers new means of describing special user preferences or digital rights management. Future research and tests in real user environments will have to prove the effectiveness and completeness of MPEG-21 digital rights management, since it is quite forseeable that future multimedia consumption over the Internet will not be free of charge [64].

CHAPTER

4 Detailed Analysis of Temporal Scalability

4.1 Overview

Most of today's widely available video codecs, which are used in diverse applications like Microsoft's MediaPlayer, Apple Quicktime or Real One, do not support enhanced scalability methods for spatial or SNR adaptation. The same is valid for open source multi-codec libraries like FFmpeg [65] or XViD [66]. Still, all of them offer broad support for simple temporal scalability by generating B-frames. The bi-directionally predicted B-frames offer substantially reduced frame sizes and therefore make the overall stream smaller and more robust to packet loss [67].

As discussed in Section 3.2.1, the only really easy and efficient temporal adaptation is to drop B-frames. Simple and arbitrary frame dropping could accidently eliminate I-frames or P-frames, which would cause severe decoding errors or drift.

According to [68, 69], novel approaches are presented, how the missing information of dropped I- and P-frames could be approximated and motion vectors could be adjusted in the compressed domain. This happens either by interpolating the four adjacent motion vectors or by finding out the dominant motion vector. Still, doing so implies extra algorithmic calculations and hereby increases the needed processing power and the achieved quality is still not optimal. If there are enough B-frames in each video second, it should not be necessary to drop any reference frames like I-frames and P-frames. So in the following, we exclusively concentrate on B-frame dropping.

Further, compared to any other type of adaptation (eg. spatial or SNR), dropping

B-frames is always the fastest. Also, studies have shown that:

> ...from a viewer's perspective, reduction of frame rate provides the best results regarding to the viewer's comprehension of the video. Severe degradation in spatial resolution or SNR will result in frames either too small or too blurred for a viewer to perceive enough details, and even worse, will distract viewers' attention and harm the comprehension of the entire video [70].

4.2 Frame Patterns

To take the highest possible advantage of temporal scalability, a video stream has to be prepared to offer a maximum number of B-frames. This section will show, that this maximum number is limited by various constraints and will come up with reasonable values for various adaptation needs.

Available MPEG-4 encoders like the MPEG-4 Microsoft reference software [20] or FFmpeg [65], allow us to choose between *fixed* and *dynamic* frame patterns.

If we decide on dynamic frame patterns, the encoder chooses when it is best to use I-, P- or B-frames arbitrarily over the video stream and might come up with patterns like `IPPPPPBPPPBPPIPIIIPPBBP...`. This decision taking increases coding time, but will offer us a smaller bandwidth video. The downside of this is that we will not have many B-frames in the stream, since the encoder preferrably decides on P-frames to reduce buffering requirements and decoding complexity. The reason for favoring P-frames is stated in the fact that consecutive P-frames not only decrease decoding complexity, but also lessen buffer requirements. When comparing display versus network order of frames (see Figure 3.7), it becomes evident that, for decoding a B-frame, both reference frames of the B-frame have to be received in advance and therefore also have to stay in the buffer until the next reference frame arrives.

If we specify a fixed encoder frame pattern, it will be repeated over the whole video stream. Its length is defined by the frequency of intra-coded I-frames. Although the frequency can be an arbitrary number, it is reasonable to be set to the video's frame rate (25 for PAL, 30 for NTSC, 15 for video telephony) or a multiple of it. The rest

of the frame pattern will be filled up with alternating P and B-frames, where we can define the number of B-frames between two reference frames (see Figure 4.1).

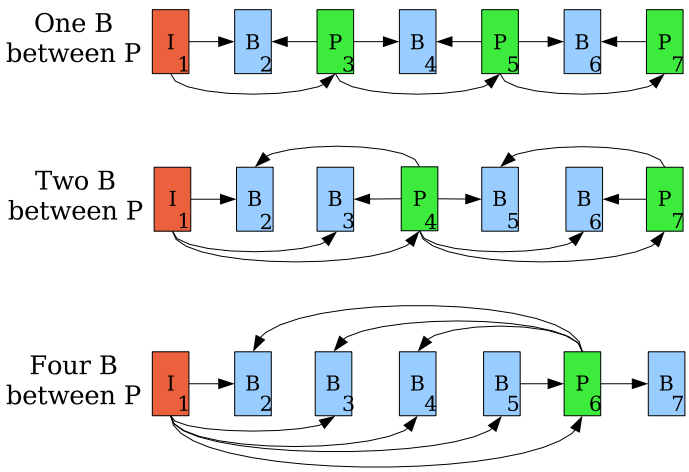


Figure 4.1: The more B-frames between P, the farther their reference frames

Note that, if P-frames are lost, the video will be undecodable until the next I-frame arrives. So for better error resilience, as a good tradeoff between coding efficiency and fast resynchronization, a video should have an I-frame at about every second. If bandwidth constraints are crucial, other reasonable values could be every 2, 5 or 10 seconds. A good reason for more frequent I-frames is, besides better error resiliance, their exploitation as entry points for random access [24], or switching points for different streams (find more about *stream switching* in Chapter 5).

Obviously, the more B-frames are in the stream, the more frames (and hereby bytes) can be dropped without any consequence, except that a high drop rate will lead to a more jerky visual experience.

With a frame pattern of 30 frames between each I-frame and with an increasing number of B-frames between P-frames, Table 4.1 shows how the temporal scalability increases and the possible minimum framerate decreases assuming that no I- or P-frames are dropped.

pattern	definition	I+P-frames	B-frames	B-percent	lowest achievable fps (from 30 fps)
IBPBPBP...BPBPB	$P \overset{1B}{\rightarrow} P$	1+14	15	50%	15
IBBPBBP...BBPBB	$P \overset{2B}{\rightarrow} P$	1+9	20	67%	10
IBBBBPB...PBBBB	$P \overset{4B}{\rightarrow} P$	1+5	24	80%	6

Table 4.1: Temporal scalability gains with increasing number of B-frames at 30 fps

4.3 Distances between Reference Frames

Further note that an increase of the number of B-frames also increases the distance between each P-frame. Also each B-frame is farther away from its two reference frames (see Figure 4.1). Obviously, this introduces larger picture differences (deltas), since prediction has to be performed across multiple frames. Consequently, both, more motion is detected and/or more quantized data has to be stored, which increases the needed frame size. Obviously, this also increases the overall filesize.

For the following discussions, the measured video was "Big Show Both" in CIF resolution (352x288) with 13000 frames, in a 30 fps scenario with a fixed pattern with an I-frame every 30 frames. This gives 434 patterns. The used encoder was the Microsoft reference software for MPEG-4 [20].

The average bandwidth per frame pattern can be calculated as follows: AF denotes the set of all available frames and $numFrm(AF)$ denotes the number of frames in the given set. All frames f_i are summed up and divided by the number of patterns $numPat$.

$$avgPatSize = \frac{\sum_{i=1}^{numFrm(AF)} size(f_i)}{numPat}$$

Each pattern consists of fps number of frames, and each frame f_i within a pattern p_j has a different size. Seen over all patterns, each frame at a certain position i within a pattern, has an average size over all patterns:

$$avgFrmSize_i = \frac{\sum_{p=0}^{numPat-1} size(f_{p*fps+i})}{numPat}, \forall i = 1 \ldots fps$$

Figure 4.2 shows, how much the average frame size at each position i within a

pattern contributes in percent to the average frame pattern bandwidth. Analysis has been done for fixed patterns of one B-frame between each P-frame ($P \overset{1B}{\rightarrow} P$), two ($P \overset{2B}{\rightarrow} P$) and four ($P \overset{4B}{\rightarrow} P$). To denote the alternating number of B-frames in the following figures, the abbreviation $xBbP$ is used instead of $P \overset{xB}{\rightarrow} P$, where $x = \{1, 2, 4\}$.

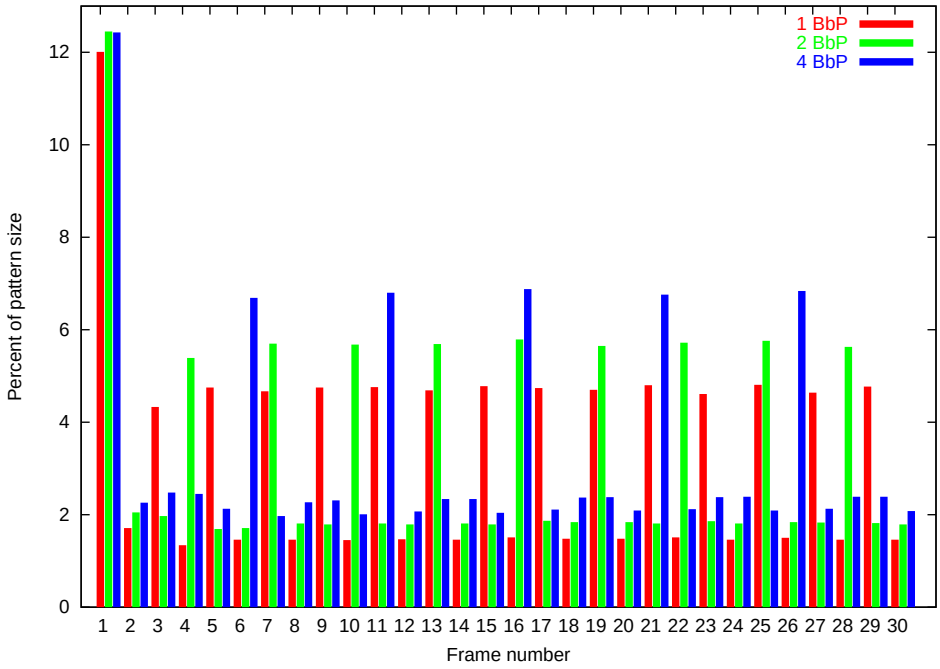


Figure 4.2: P-frames grow proportionally to their distance to each other

The first frame is always an I-frame, which accounts for 12% of one pattern. Before analysing in-between B-frames, we concentrate on the larger-sized P-frames, which grow significantly with increasing distance between each other.

With one B-frame between each P-frame ($P \overset{1B}{\rightarrow} P$), we have 14 P-frames, each using about 4.5% of the full pattern size. If two B-frames ($P \overset{2B}{\rightarrow} P$) are chosen, the number of P-frames is reduced to nine and their proportionate size grows to over 5.5% each, since their distance to each other increases. If we use four B-frames ($P \overset{4B}{\rightarrow} P$), there are only five P-frames left, with proportionate sizes of about 7% each.

Understanding the drawback of reference distance, it further becomes clear that in all cases where $P \overset{xB}{\rightarrow} P$, and $x \geq 3$, the innermost B-frames are the farthest away to their reference frames and hereby also become recognizably large. This is nicely

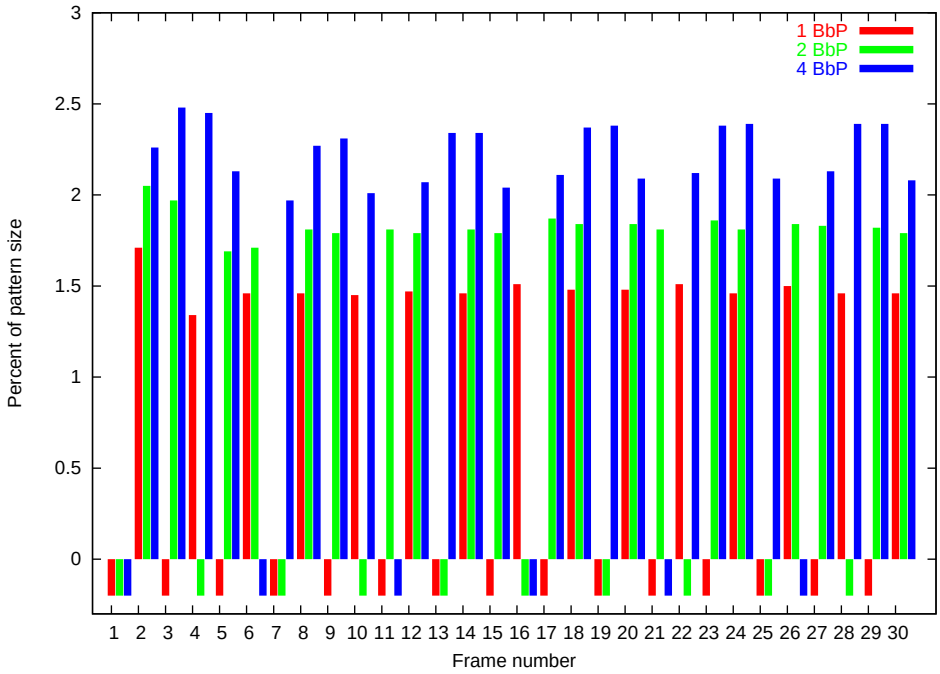


Figure 4.3: Innermost B-frames need more bytes to code deltas

shown in Figure 4.3, where only the B-frames are drawn. Note that I- and P-frames are blended out by only showing them as negative values. For absolute comparison, use Figure 4.2.

For the $P \overset{4B}{\rightarrow} P$ case, the frames $2, 3, 4, 5$ are coded as B-frames, where the two innermost B-frames $3, 4$ have increased sizes. This symptom is nonexistent for the $P \overset{2B}{\rightarrow} P$ case, since both B-frames have the same distance to both of their references (either their left or right reference frame is closer). The same holds for the single B-frame in $P \overset{1B}{\rightarrow} P$. So we come to the conclusion, that it is not useful to code too many B-frames between P-frames. Available standard codecs default to use a maximum of $P \overset{2B}{\rightarrow} P$, which, under the above given circumstances, seems reasonable. Still, if we want to gain a maximum of scalability, we shall accept the consequences and code our video with a higher number of in-between B-frames $P \overset{xB}{\rightarrow} P$, and $x \geq 3$.

4.4 Quantization Steps for Different Frame Types

In this section, we want to examine the coherence of quantization, filesize, and the number of B-frames between P-frames. We always compare three streams of "Big Show Both, CIF" with each other, with $P \overset{1B}{\rightarrow} P$, $P \overset{2B}{\rightarrow} P$ and $P \overset{4B}{\rightarrow} P$. Since $P \overset{2B}{\rightarrow} P$ is qualitatively always somewhere in between the other two, we silently ignore it in the following descriptions, though it is still plotted in the comparing figures.

When using MPEG-4 or other DCT (Descrete Cosine Transformation) based video encoding algorithms, each pixel block is transformed into the frequency domain. A calculated coefficient is assigned to each frequency. This coefficient can be mapped to a limited range of integer values (for MPEG-4 it is 0 – 255). The range of the available integer values can further be reduced by the level of *quantization*. Quantization is defined in levels between $quant = 1$ (no loss at all) and $quant = 32$ (heavy degradation). With an increasing quantization value, a reduction of possible mappings from the original coefficient to the available pool of integers is achieved. The more coarse-grained this mapping gets, the lower the decoded quality will be, but this also severely reduces bandwidth. Allthough a high quantization level decreases a frame's data size, each frame still has some header overhead and codes the detected motion vectors. Still, quantization is offering the highest gains in bandwidth reduction.

If we keep the I-frame and P-frame quantization fixed and only change the B-frame quantization, overall bandwidth requirements substantially decrease with increasing quantization of B-frames and the number of available B-frames. Keeping I- and P-frames untouched offers the advantage that we can rely on high quality reference frames. Figures 4.4, 4.5 and 4.6 show changing B-frame quantization with fixed I- and P-quantization of 16, 12 and 8 respectively.

For a fixed I- and P-quantization of 16, Figure 4.4(a) shows that if the stream with $P \overset{4B}{\rightarrow} P$ is exposed to B-frame quantization between 16 and 32, it drops fastest in quality, since more frames (there are 24 B-frames) suffer from quantization. But even with increasing B-frame quantization, the so-called base layer with all statically quantized I- and P-frames stays at the same size (see Figure 4.4(b)). Note, that at very high quantization levels, in comparison to the previous quantization level, there is not much more data reduction possible, so all quantization curves represent a logarithmic shape. The remaining distance between the overall bandwidth and

the corresponding baselayer is related to the available number of B-frames and their coded motion vector parts and headers.

With a low quantization of B-frames (eg $quant = 16$), the bandwidth requirements are the highest for the $P \overset{4B}{\rightarrow} P$ stream. Still, the $P \overset{1B}{\rightarrow} P$ stream rapidly converges to the minimally achievable B-frame size. The $P \overset{4B}{\rightarrow} P$ stream has more B-frames, so there is more overall bandwidth reduction possible by higher B-frame quantization.

From this analysis, it becomes obvious that the more B-frames exist, the more bandwidth can be reduced by dropping them. This bandwidth reduction is called *bandwidth scalability*, which defines the ability to reduce the needed bandwidth of a video by deleting B-frames. This only introduces a lower frame rate (and hereby choppiness in displaying) but has no negative inpact on decoding, as would be introduced by dropping I- or P-frames.

Figure 4.4(c) shows the possible bandwidth reduction in percent if all available B-frames are dropped. The figure points out that $P \overset{4B}{\rightarrow} P$ hereby offers the highest bandwidth scalability, even though with increasing B-frame quantization the bandwidth scalability is decreasing for all three streams, since B-frames are coded with fewer bits.

4.4.1 Absolute Versus Relative Gain of Scalability

As described in Section 4.3, the more B-frames are used, the higher all distances to reference frames become. So increasing the bandwidth scalability by B-frames also leads to an overall increased bandwidth. If we want to compare the bandwidth scalability of $P \overset{4B}{\rightarrow} P$ and $P \overset{1B}{\rightarrow} P$ fairly, we have to first adjust the $P \overset{4B}{\rightarrow} P$ to lower bandwidth needs by reducing the needed bandwidth of $P \overset{4B}{\rightarrow} P$ down to the bandwidth of $P \overset{1B}{\rightarrow} P$ by deleting some frames. This not only reduces the frame rate, but also reduces the overall bandwidth scalability. Figure 4.4(d) shows, that the $P \overset{4B}{\rightarrow} P$ stream at higher quality B-Frames (lower quantizations) has to drop more B-frames to reach the $P \overset{1B}{\rightarrow} P$ bandwidth. At a certain point of quantization, the $P \overset{4B}{\rightarrow} P$ B-frames are getting small enough so that no dropping is needed anymore and $P \overset{1B}{\rightarrow} P$ is automatically reached. This fact is also shown in Figure 4.4(b) at the streams' crossing points, where the $P \overset{1B}{\rightarrow} P$ stream's B-frames fastly converge to their minimal size and the $P \overset{4B}{\rightarrow} P$ stream is far from converging since it has more

B-frames.

Because of the high number of B-frames, $P \overset{4B}{\rightarrow} P$ keeps its relative scalability (after adjusting the overall bandwidth to $P \overset{1B}{\rightarrow} P$), even at high quantizations. Figure 4.4(e) shows, that for a low B-quantization, $P \overset{4B}{\rightarrow} P$ has to drop many B-frames, so the relative bandwidth scalability is "only" 55% with a severe frame loss of about 5 frames per second (see Figure 4.4(d)), instead of the absolute bandwidth scalability of close to 70%. The higher the quantization, the more frames can be displayed (at the same bandwidth as $P \overset{1B}{\rightarrow} P$), so the absolute and relative scalability come close together.

4.4.2 Optimum Encoder Settings

Concluding on the previous analysis, in terms of scalability and quality, we search the optimum stream at $P \overset{4B}{\rightarrow} P$ with I- and P-quantization set to 16, by setting the B-quantization to 25. This gives the same bandwidth as $P \overset{1B}{\rightarrow} P$, also with a B-quantization of 25. So with a bandwidth of about 420 kpbs, it offers nearly 56% absolute bandwidth scalability and 24 droppable B-frames. In comparison to that, the $P \overset{1B}{\rightarrow} P$ stream offers only 24% absolute bandwidth scalability and 15 droppable B-frames at the same bandwidth. Still, in all the consideration we ignore a qualitative reduction of -0.44 dB PSNR from a maximum of 28.38 dB, but this reduction is invisible even to experts.

Figure 4.5 shows the same comparisons as Figure 4.4, but with a better I- and P-quantization set to 12. Here, the PSNR quality is in higher ranges, but also the base layer of I- and P-frames is larger. Here, the optimal $P \overset{4B}{\rightarrow} P$ stream can be found with a B-quantization set to 16, wich offers 59% absolute bandwidth scalability and 24 B-frames at 615 kbps. At the same bandwidth, $P \overset{1B}{\rightarrow} P$ only offers 25.5% scalability and 15 B-frames with an ignorable increase of quality by $+0.38$ dB PSNR of a maximum of 29.81 dB.

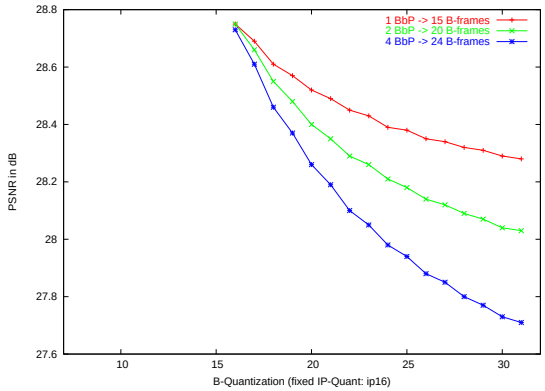
At an even better I- and P-quantization set to 8, all subfigures in Figure 4.6 show similar results but with shifted PSNR quality ranges. The streams already intersect at a B-quantization set to 10 with a bandwidth of 1064 kbps. Increased absolute bandwidth scalability of 62% with 24 droppable B-frames in $P \overset{4B}{\rightarrow} P$ can be exploited in comparison to only 24% scalability with 15 B-frames for $P \overset{1B}{\rightarrow} P$. This again costs

about −0.37 dB PSNR of a maximum of 31.95 dB.

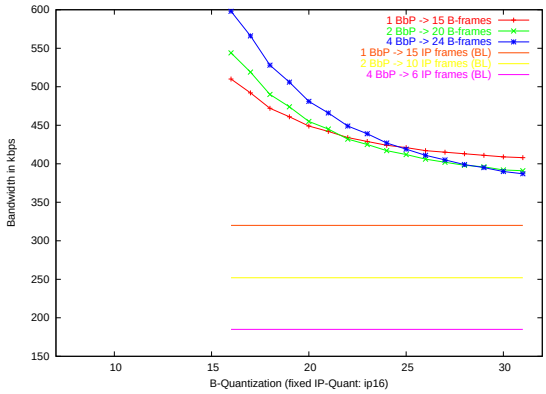
This leads to three streams with $P \overset{4B}{\rightarrow} P$ and fixed I- and P-quantizations of 16, 12, and 8 (see Table 4.2). All streams offer a maximum dropping of 24 B-frames (from 30 fps down to 6 fps) with an approx. absolute bandwidth scalability of at least 55%. So to fit varying bandwidth capacities, it is possible to adapt each stream to its maximum and then switch to the next stream. Advantages and caveats of *stream switching* will be discussed in detail in Chapter 5.

kbps	I-/P-quant	B-quant	absolute bandwidth scalability	overall PSNR [dB]
420	16	25	55.9%	27.94
615	12	16	58.6%	29.43
1064	8	10	62.2%	31.58

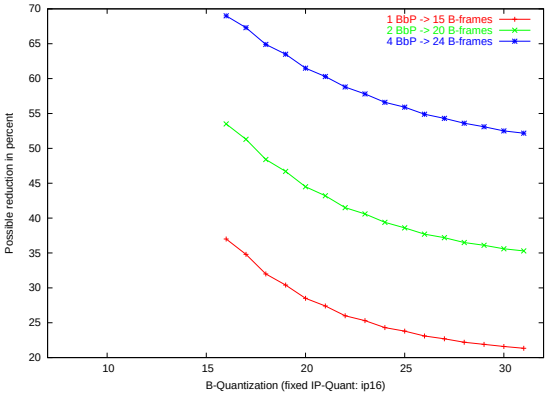
Table 4.2: Optimum streams with $P \overset{4B}{\rightarrow} P$ and I-/P-quantizations 16, 12, and 8



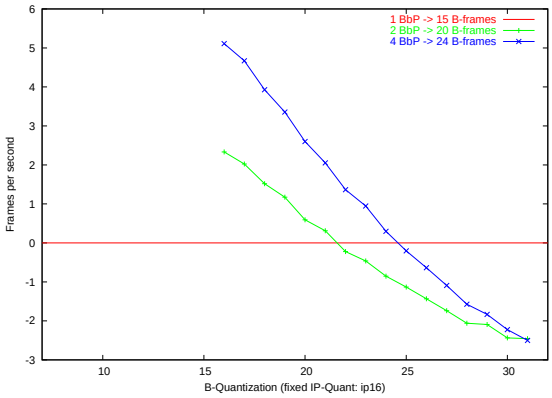
(a) Quality reduction by increasing B-frame quantization



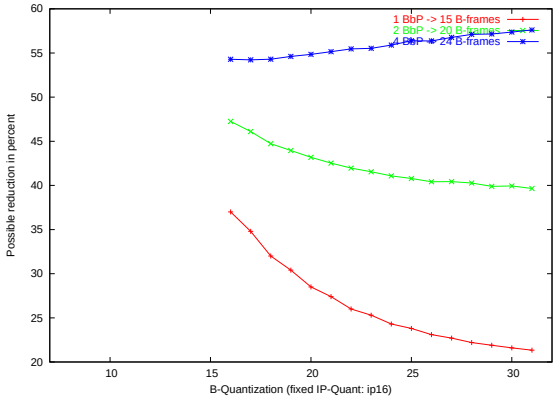
(b) Bandwidth reduction by increasing B-frame quantization



(c) Absolute overall bandwidth scalability

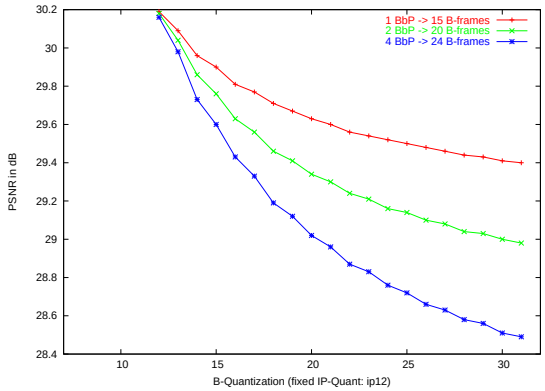


(d) Needed framerate reduction to reach $P \overset{1B}{\rightarrow} P$ bandwidth

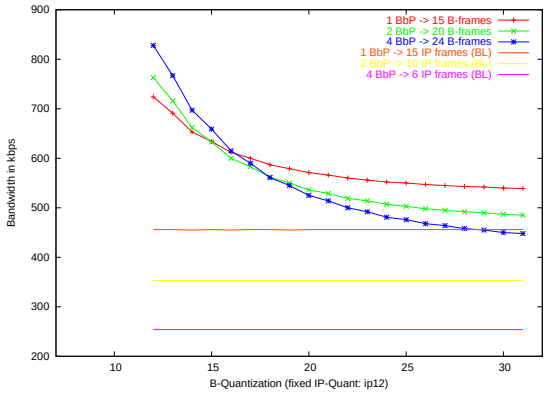


(e) Relative overall bandwidth scalability, adjusted to $P \overset{1B}{\rightarrow} P$

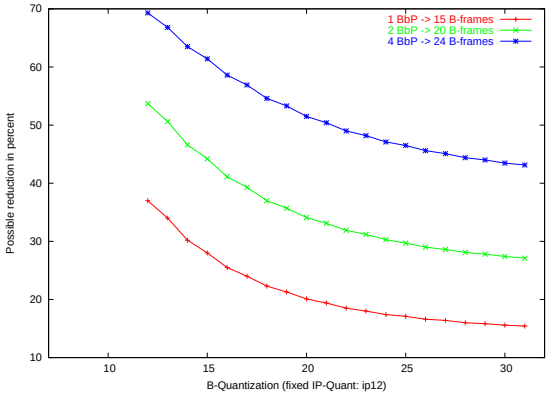
Figure 4.4: Dynamic B-frame quantization with static I-/P-quantization 16



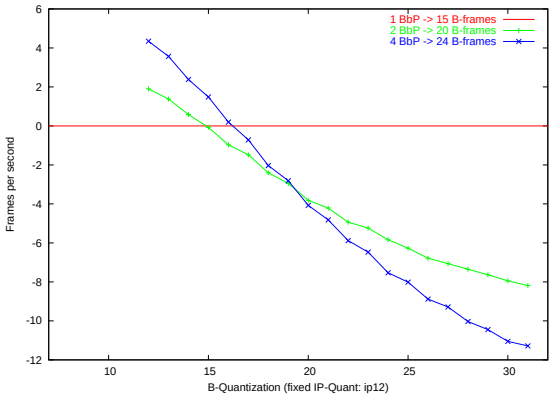
(a) Quality reduction by increasing B-frame quantization



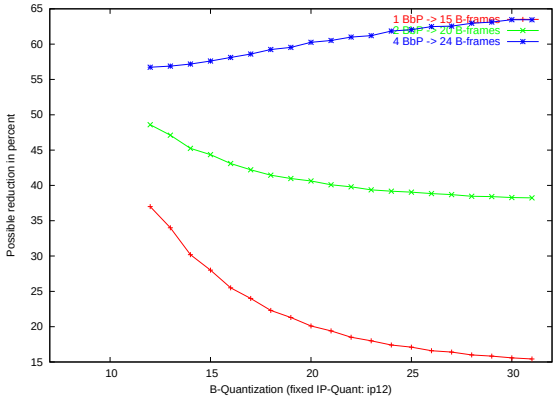
(b) Bandwidth reduction by increasing B-frame quantization



(c) Absolute overall bandwidth scalability

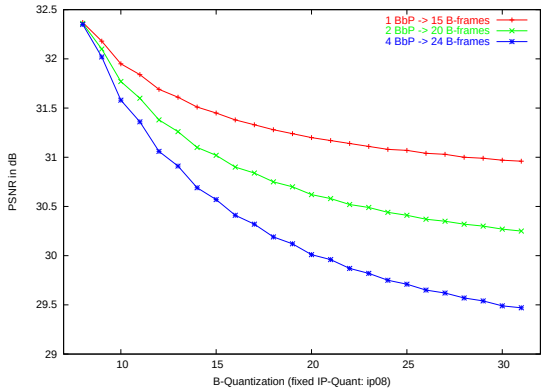


(d) Needed framerate reduction to reach $P \overset{1B}{\rightarrow} P$ bandwidth

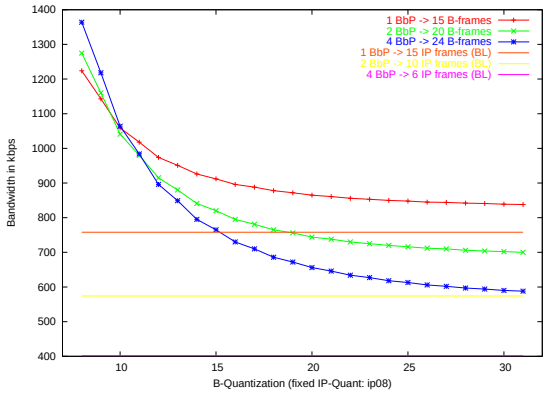


(e) Relative overall bandwidth scalability, adjusted to $P \overset{1B}{\rightarrow} P$

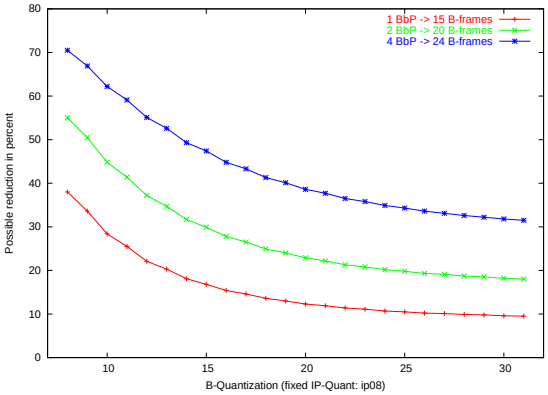
Figure 4.5: Dynamic B-frame quantization with static I-/P-quantization 12



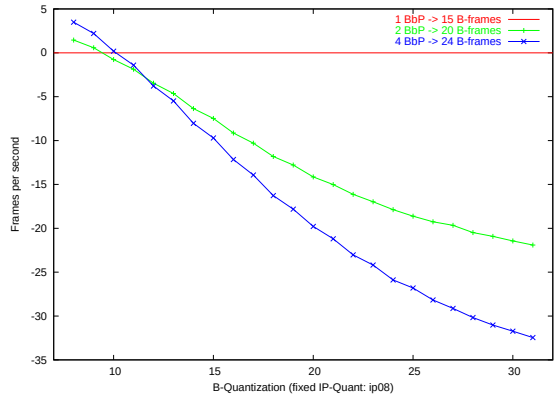
(a) Quality reduction by increasing B-frame quantization



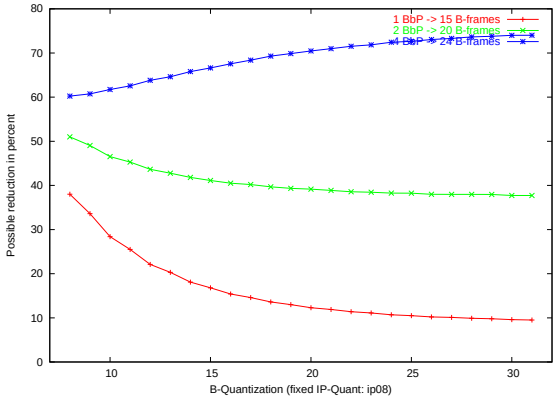
(b) Bandwidth reduction by increasing B-frame quantization



(c) Absolute overall bandwidth scalability



(d) Needed framerate reduction to reach $P \overset{1B}{\rightarrow} P$ bandwidth



(e) Relative overall bandwidth scalability, adjusted to $P \overset{1B}{\rightarrow} P$

Figure 4.6: Dynamic B-frame quantization with static I-/P-quantization 8

4.5 Prioritization of B-Frames

If bandwidth or terminal constraints force the server to deliver a lower frame rate, then – given a sufficient number of B-frames – it is the sender's decision which B-frames to drop. In the following, we will always assume a "semi-intelligent" server, which is capable of distinguishing between I-, P- and B-frames.

The following terms will be used in the following:

- *streamBW* is the bandwidth which *should* be streamed out, so it is assumed to be fixed to the average bandwidth of the video stream (or might be further refined and determined according to Chapter 2). In the following formula, AF denotes the set of all available frames in the stream (AF = All Frames). f_i iteratively references each of the available frames in AF.

$$streamBW = avgBW = \frac{\sum_{i=1}^{numFrm(AF)} size(f_i)}{numFrm(AF)} * fps, \forall f_i \in AF$$

- *netBW* is the really available bandwidth, to which each video second's bitsize ($VsecBS$) has to be adjusted, so that the overall video fits through the network under those hardened network constraints. In the following formula, DF denotes the set of all dropped frames from the stream (DF = Dropped Frames).

$$netBW = streamBW - \frac{\sum_{j=1}^{numFrm(DF)} size(f_j)}{numFrm(DF)} * fps, streamBW \geq netBW, \forall f_j \in DF$$

- The amount of data reduction determined by the lower *netBW* in comparison to the full *streamBW* within one $VsecBS$ is described by a percentage value stored in the *dropRate*, which is defined as follows:

$$dropRate = 100 - \frac{netBW}{streamBW/100}$$

- In the following discussions, an inverse value of the *dropRate* is used to describe the amount of data which is not dropped, but which is kept and finally sent

out by the server. By that, this *keepRate* is the percentage of still available *streamBW* in $VsecBS$ after applying the necessary adaptation, which leads to *netBW*, so it is defined as

$$keepRate = \frac{netBW}{streamBW/100}$$

4.5.1 "Bad" Prioritizations

A rather "unintelligent" server will stream out *netBW* and then realizes, that, after sending *keepRate* percent of the *streamBW*, no more data can be sent out for this $Vsec$. So the still outstanding B-frames of this $Vsec$ (denoted as *dropRate* percent of the *streamBW*) are dropped. A possibly resulting frame pattern for a $Vsec$ with 30 frames might then look like this: `IBBBBPBBBBPBBBBPBB--P----P----`, where the dashes (`-`) denote the dropped frames. So the applied adaptation is non-uniformly distributed in the frame pattern. This obviously gives extremely choppy results, since the human eye and brain are trying to track the smooth motion of objects [71], and are therefore highly irritated.

The choppiness can be reduced, maybe even avoided by distributing the missing frames over the whole $Vsec$ [24]. To identify which frame should be dropped, we have to introduce *priority values* for each B-frame. Note, that this implies algorithmical effort, since we have to tag each frame either a priori, which means according to a hint file, or directly within the sending routine. Assuming, that each frame already has its priority stored, the code fragment in Table 4.3 shows how the streamout routine easily decides, if a certain frame's priority fits into the bounds of *netBW*. This bound is defined as *maxPrio*, which is simply a mapping of the *keepRate* percentage on the possible frame priority range from $1 \ldots fps$ for one $Vsec$, where fps is the static number of frames per second all over the video.

Simply assigning random numbers as priorities would render impossible a good evaluation of the quality of the chosen priority values, which might randomly be equal or very similar to the above mentioned, unattractive non-uniformly distributed adaptation pattern. Further, when using random priorities, we would have to make sure that no number is used twice within one $Vsec$, because then we would wrongly drop too many frames in this $Vsec$.

```
maxPrio = fps * keepRate/100;
while (! eof(stream)) {
  frm = getNextFrameFromStream();
  if (frm.type == B_VOP) {
    if (frm.prio < maxPrio)
      send(frm);
    else
      drop(frm);
  }
}
```

Table 4.3: Temporal adaptation in case of prioritized frames

For the following, we use B-frame prioritization and dropping only based on decisions of the server-side, hinted by client feedback on network behaviour. To further gain more immediate reaction to network problems, this priority-based dropping can also happen *within* the network, on specialized multimedia routers [72] [73]. Since those routers unfortunately are not yet widely available on normal IP networks, we refrain to use them in our measurements. Still, the availability and use of such technology will even more increase the value of good prioritization algorithms under certain circumstances [74].

4.5.2 Timely Uniform Distribution

If dropped B-frames would be uniformly distributed within the $Vsec$, the resulting frame pattern with frame drops should look like `I-B-BP-B-BP-B-BP-B-BP-B-BP-B-B`. This is called *timely uniform distribution.*

In Appendix A.1, we will present an algorithm, which sets up a table with frame priorities. Basically, it implements a recursive depth-search, which is limited to a certain depth. At each depth, left and right traversals are started, which sets ascending priority numbers to the alternating tree halves.

The following example in Table 4.4 shows the needed three depths for a pattern size of seven frames, and how the new priorities are uniformly assigned for each depth. In the first level, no priorities are assigned to the table of frames yet, which is denoted

as double dots (..). Only the center frame gets the highest number, so this frame will be dropped first, when priority-based adaptation is necessary. After the algorithm has worked down to the third level, all frames have assigned priorities, which are perfectly timely uniform distributed.

For further illustration, Table 4.5 shows a pattern size of 15 frames, with a four-step priority assignment, which also comes up with a timely uniform distributed prioritization scheme on level four.

```
1:  ..-..-..- 7-..-..-..
2:  ..- 6-..- 7-..- 5-..
3:   4- 6- 2- 7- 1- 5- 3
```

Table 4.4: Building tree for timely uniform distributed prioritization for 7 frames

```
1:  ..-..-..-..-..-..-..-15-..-..-..-..-..-..-..-
2:  ..-..-..-14-..-..-..-15-..-..-..-13-..-..-..-
3:  ..-12-..-14-..-10-..-15-..- 9-..-13-..-11-..-
4:   8-12- 6-14- 4-10- 2-15- 1- 9- 3-13- 5-11- 7-
```

Table 4.5: Building tree for timely uniform distributed prioritization for 15 frames

After the table is filled up and stored, the priority is looked up in this table for the current frame's position. This priority can be applied to each frame, shortly before it will be sent out or dropped, without any dependencies to prior or future frames (see Table 4.6). The decision of dropping is taken, if the current frame's priority exceeds a maximum priority level which was derived from the *keepRate*.

Table 4.7 shows different *keepRates* and the resulting number of dropped frames denoted as double dots (..). The lower the *keepRate*, the more frames are dropped and less frames are "kept" for sending, so a 100% *keepRate* means no adaptation, a 80% *keepRate* in this example means the sendout of 20 frames out of a 25 fps pattern. The patterns nicely demonstrate, that the remaining frames are always uniformly distributed at every level of adaptation.

```
currentFrameNo = 0;
maxPrio = fps * keepRate/100;
while (! eof(stream)) {
   frm = getNextFrameFromStream();
   if (frm.type == B_VOP) {
        frm.prio = prioTable[currentFrameNo % fps];
   } else // I- or P-frame
        frm.prio = 0;
if (frm.prio < maxPrio)
   send(frm);
else
   drop(frm);
currentFrameNo++;
}
```

Table 4.6: Efficient priority assignment and adaptation of frames

keepRate	resulting frame pattern
100%:	10-18- 8-22-16- 6-24-14- 4-20-12- 2-25-11- 1-19-13- 3-23-15- 5-21- 7-17- 9-
96%:	10-18- 8-22-16- 6-24-14- 4-20-12- 2-..-11- 1-19-13- 3-23-15- 5-21- 7-17- 9-
92%:	10-18- 8-22-16- 6-..-14- 4-20-12- 2-..-11- 1-19-13- 3-23-15- 5-21- 7-17- 9-
88%:	10-18- 8-22-16- 6-..-14- 4-20-12- 2-..-11- 1-19-13- 3-..-15- 5-21- 7-17- 9-
84%:	10-18- 8-..-16- 6-..-14- 4-20-12- 2-..-11- 1-19-13- 3-..-15- 5-21- 7-17- 9-
80%:	10-18- 8-..-16- 6-..-14- 4-20-12- 2-..-11- 1-19-13- 3-..-15- 5-..- 7-17- 9-
76%:	10-18- 8-..-16- 6-..-14- 4-..-12- 2-..-11- 1-19-13- 3-..-15- 5-..- 7-17- 9-
72%:	10-18- 8-..-16- 6-..-14- 4-..-12- 2-..-11- 1-..-13- 3-..-15- 5-..- 7-17- 9-
68%:	10-..- 8-..-16- 6-..-14- 4-..-12- 2-..-11- 1-..-13- 3-..-15- 5-..- 7-17- 9-
64%:	10-..- 8-..-16- 6-..-14- 4-..-12- 2-..-11- 1-..-13- 3-..-15- 5-..- 7-..- 9-
60%:	10-..- 8-..-..- 6-..-14- 4-..-12- 2-..-11- 1-..-13- 3-..-15- 5-..- 7-..- 9-
56%:	10-..- 8-..-..- 6-..-14- 4-..-12- 2-..-11- 1-..-13- 3-..-..- 5-..- 7-..- 9-
52%:	10-..- 8-..-..- 6-..-..- 4-..-12- 2-..-11- 1-..-13- 3-..-..- 5-..- 7-..- 9-
48%:	10-..- 8-..-..- 6-..-..- 4-..-12- 2-..-11- 1-..-..- 3-..-..- 5-..- 7-..- 9-
44%:	10-..- 8-..-..- 6-..-..- 4-..-..- 2-..-11- 1-..-..- 3-..-..- 5-..- 7-..- 9-
40%:	10-..- 8-..-..- 6-..-..- 4-..-..- 2-..-..- 1-..-..- 3-..-..- 5-..- 7-..- 9-
36%:	..-..- 8-..-..- 6-..-..- 4-..-..- 2-..-..- 1-..-..- 3-..-..- 5-..- 7-..- 9-
32%:	..-..- 8-..-..- 6-..-..- 4-..-..- 2-..-..- 1-..-..- 3-..-..- 5-..- 7-..-..-
28%:	..-..-..-..-..- 6-..-..- 4-..-..- 2-..-..- 1-..-..- 3-..-..- 5-..- 7-..-..-
24%:	..-..-..-..-..- 6-..-..- 4-..-..- 2-..-..- 1-..-..- 3-..-..- 5-..-..-..-..-
20%:	..-..-..-..-..-..-..-..- 4-..-..- 2-..-..- 1-..-..- 3-..-..- 5-..-..-..-..-
16%:	..-..-..-..-..-..-..-..- 4-..-..- 2-..-..- 1-..-..- 3-..-..-..-..-..-..-..-
12%:	..-..-..-..-..-..-..-..-..-..- 2-..-..- 1-..-..- 3-..-..-..-..-..-..-..-
8%:	..-..-..-..-..-..-..-..-..-..- 2-..-..- 1-..-..-..-..-..-..-..-..-..-
4%:	..-..-..-..-..-..-..-..-..-..-..- 1-..-..-..-..-..-..-..-..-..-
0%:	..-

Table 4.7: Prioritized frame pattern exposed to increasing level of adaptation

Varying Frame Sizes Note, that the simple algorithms in Table 4.3 and 4.6 assume frames which are of equal byte sizes. Doing so, a calculated maximum priority level ($maxPrio$) according to a certain $keepRate$ will lead to a bandwidth reduction from $streamBW$ down to $netBW$ for each $Vsec$. For the following measurements, a more sophisticated algorithm is used, which also addresses the problems of different frame sizes.

This adaptation process introduces buffer requirements for at least one $Vsec$. This $Vsec$ has to be adapted to fit the $keepRate$, and then streamed out with the adjusted priority level, where some frames are dropped.

As an example, in Figure 4.7 a $streamBW$ of 232 kbps is presumed, but the measured network bandwidth is only 198 kbps. The available $netBW$ defines a $keepRate = 85\%$, so we have to reduce data by 15%. We first select all frames from $Vsec_{34}$ to fill the $streamBW$ and then sort by priorities. Finally, we chop off the highest priority numbers until we can satisfy the $netBW$.

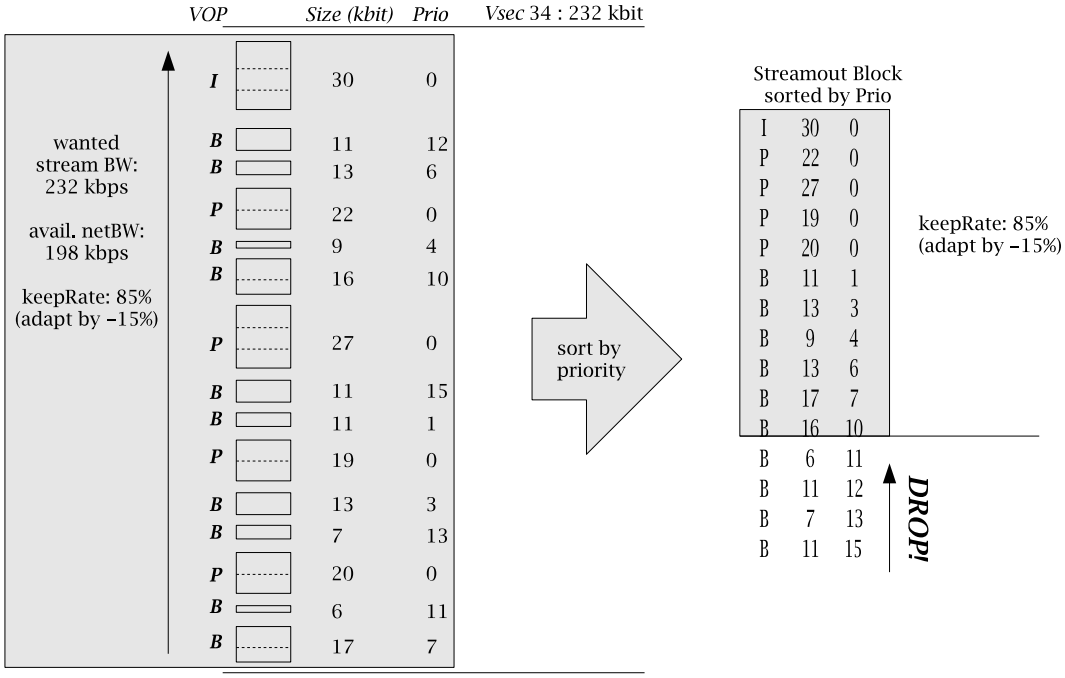


Figure 4.7: Prioritization within one $Vsec$ with varying frame sizes

4.5.3 Varying Motion Energy within a Pattern

Though timely uniform distribution performs well on the average over a full stream, the following subsections will describe more sophisticated methods of gaining priority values which even more reflect the actually displayed video content.

If the motion of a certain $Vsec$ is not uniformly distributed, but the previously discussed "timely uniform distribution" algorithm prioritizes the frames in an evenly distributed manner, the resulting priorities are not optimal in terms of choppiness. So if the first half of a certain $Vsec$ shows a high motion scene and then switches to a nearly still image for the remaining half second, we would prefer to give higher priorities to the first half, which would result in a smoother presentation of the high motion and unchanging quality for the still scene, since it does obviously not matter, if a still image is coded using one or 15 frames.

Unfortunately, there is no obvious and reliable way of measuring exactly this motion, that is important for a user. This would mean to extract the visual object of interest within the video and then measure its motion, which also leads to transformation of the visible object shape (think of a person running and turning).

In the compressed domain, motion vectors on macroblocks stored in each B-frame, can be used to come up with motion models. Liu et al. calculated those motion models, which then were transformed to priority values and evaluated by subjective tests [75]. Results have shown, that the motion models and related priorities provide high correlation with the human perceived motion energy.

Similar work on evaluating the importance of motion vectors has been done in [76]. But here, a frame's priority is also increased by the number of slice level references, which means, on an MPEG-4 macroblock level, how many other macroblocks are referencing this one for prediction. The more important macroblocks a frame owns, the higher its priority will be.

4.5.4 Scene Cuts and Key Frames

Even if motion is uniformly distributed over a $Vsec$, there might occur hard scene cuts, which might be detected automatically or are described using MPEG-7 temporal regions. It is important, not to drop those frames which are close to the scene

cut, since they might offer important visual information for the user or for audio synchronization. Think of a car explosion, accompanied with rumbling thunder, then it is expected by the user, to see the car going up into smoke, immediately when the sound is audible. If we would drop those frames, the intact car would still stand around, when loud explosion sounds are played.

The prior scenario could be detected but also hinted by an editor. Key frames of certain scenes are very valuable and should not be arbitrarily dropped. They should be treated like reference frames[1].

4.5.5 PSNR-Based Quality Detection of Drop Patterns

The most exact detection of frame importance can be conducted in the uncompressed domain, based on each frame's PSNR value. To be more specific, we measure the loss of PSNR quality all over a frame, if a certain frame is dropped and the preceeding frame has to be replayed again. This means that, if a frame is dropped, the old frame keeps being displayed, and therefore causes a loss of information, quality and motion. This is compared to the original frame at this time point and gives a severely decreased PSNR value, which is then used for prioritization.

Table 4.8 demonstrates this behavior on randomly dropped B-frames, where eg. frame number 3 was dropped, so frame number 2 was replayed, which resulted in a quality reduction from 33.984 dB down to 23.984 dB. Unfortunately, the further frames 4 and 5 were also dropped, which resulted in replaying frame 2 even longer. PSNR values dropped from 34.191 dB to 19.191 dB, and from 34.432 dB down to 14.432 dB respectively. The quality loss is growing with each frame, since the error is propagated due to continuing motion in the video. Find a more formal definition of quality estimation in [21].

Quality Controlled Temporal Video Adaptation (QCTVA) [77] generates all possible combinations of dropping patterns and stores them in a graph, the so called *modification lattice*, which starts at the full frame pattern and then, for each

[1] Eg., the movie *Fight Club* contains scenes of the schizophrenic leading character, where his true identity is revealed and shown for a small fraction of a second. This fact is well known to many fans of this DVD video.

No	Type	PSNR [dB]	No	Type	PSNR [dB]
1	I	35.342	1	I	35.342
2	B	33.993	2	B	33.993
3	B	33.984	***2***	***B***	***23.984***
4	B	34.191	***2***	***B***	***19.191***
5	B	34.032	***2***	***B***	***14.032***
6	P	35.561	6	P	35.561
7	B	34.432	***6***	***P***	***14.432***
8	B	34.331	8	B	34.331
9	B	34.531	9	B	34.531
10	B	34.667	***9***	***B***	***14.667***
11	P	35.123	11	P	35.123
Original quality		34.562	After dropping		26.835

Table 4.8: PSNR estimation in case of randomly dropped B-frames

level, simulates the dropping of one B-frame (see Figure 4.8). For all resulting nodes, where one B-frame was dropped, further frame dropping is applied, so again all possible combinations are generated.

Since multiple nodes, after dropping another B-frame, might result in the same subnode, there do exist subnodes, which can be reached from more than one parent.

This lattice is then traversed in a best first manner for the highest quality result at each horizontal level. So, starting at the top-level node of the unadapted frame pattern, the best performing subnode of the next level is chosen. Best performing is simply defined by the highest PSNR value in this level. For the example pattern `IBBPBB`, Figure 4.9 shows how QCTVA examines each level in a best first expansion manner.

Since every level always symbolizes one addtional dropped B-frame, priority levels are assigned to those frames according to the level, where they where dropped. Priorities are counted down from the height of the lattice, which is equal to the number of B-frames in the pattern. I- and P-frames always get the priority value zero (0), which denotes their extremely high importance. In Figure 4.9, the resulting priorities would be `0-4-1-0-2-3`. Zero (0) is assigned for the I-frame, then the lowest priority (4) is assigned to the first B-frame, since in the first level, the B-frame at position 2

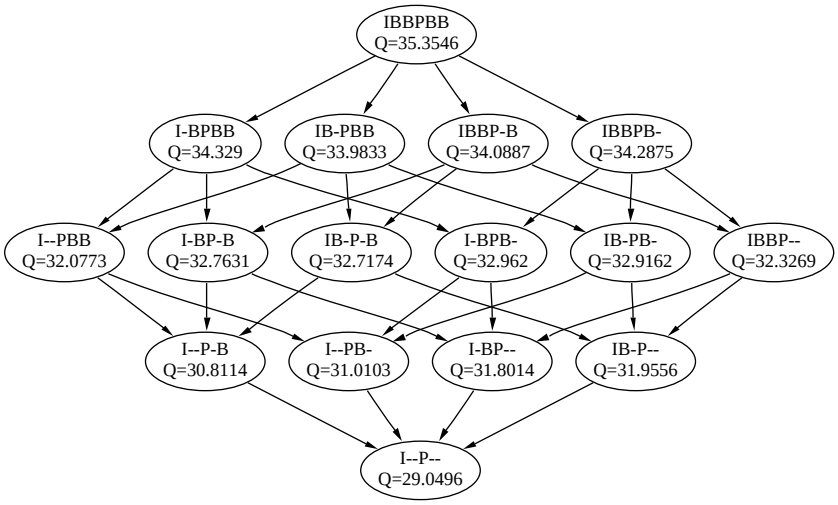


Figure 4.8: Modification lattice including quality measures

in the pattern is dropped first.

The next level is only entered via a direct link, so QCTVA might marginally miss the next optimal modification. For every level, the dropped frame is identified from the actual pattern and the next priority value is assigned accordingly.

According to Figure 4.9, on level three, the technique misses the absolute optimum, since the decision in the upper level offers no direct link. Still, those dependencies between parents and subnodes are necessary to provide unique priorities and further severely speeds up the path finding process, since for best first expansion, only the directly linked subnodes have to be expanded. Please also note, that on a regular best effort network like the Internet, adaptation needs are expected to be within $10 - 30\%$, or other means of adaptation like *stream switching* have to be addressed. Further, if more than 30% adaptation has to be performed on a $25fps$ stream with use of B-frame dropping, this will result in unacceptable frame rates below 15 frames.

For simple adaptation needs, only the upper levels of the lattice have to be traversed. But propagation errors because of missing links between an already chosen parent node and the next optimal subnode will only occur at lower levels (if at all) [21].

Since the QCTVA algorithm can easily be changed from best first expansion (BFE)

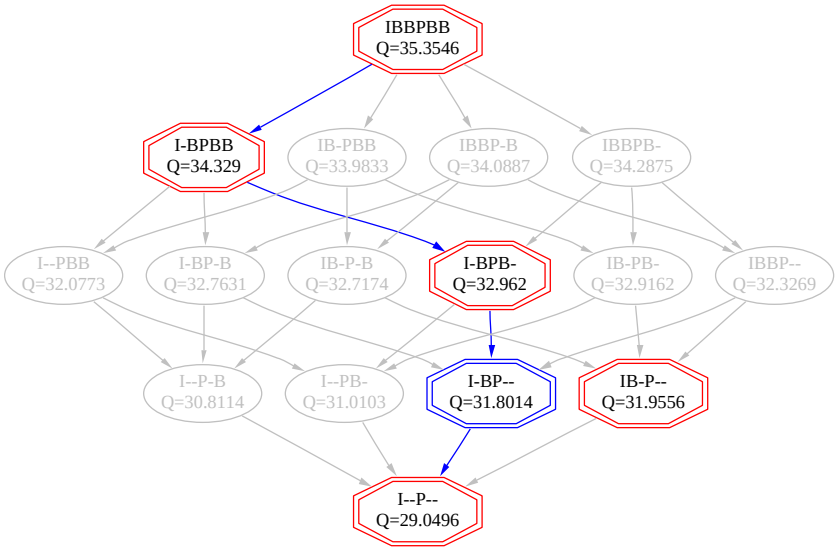


Figure 4.9: QCTVA top-down searches linked patterns with highest quality

to worst first expansion (WFE), also the qualitatively worst priorities can be calculated. This QCTVA WFE will be used for further comparisons.

4.6 Evaluation of the Different B-Frame Prioritization Schemes

The following PSNR comparisons consider the QCTVA BFE (best first expansion), timely uniform distribution and QCTVA WFE (worst first expansion) prioritization algorithms. Random and motion-based prioritization are assumed to lie in between BFE and WFE, which obviously denote the highest and lowest qualitative limits.

Hand-annotated metadata prioritization is out of scope for this comparison, since eg. for a key frame, importance is not measurable via PSNR. If annotation is performed well, it would always outperform any other algorithm, but hand-annotating videos is very time consuming so it is normally not feasible in this detail.

The tests were conducted on MPEG-4 CIF reference streams defined by the MPEG consortium (see Table 4.9). All video sequences were coded with an I- and P-quantization of 16 and a B-quantization of 28 at a frame rate of 30 fps and a fixed

frame pattern with $P \overset{4B}{\rightarrow} P$, using the Microsoft MPEG-4 reference software [20]. Note that, depending on the video, more or less motion has to be coded, even though fixed quantization values for all frames are used. This results in severely different bandwidth requirements for the test sequences to reach about similar average PSNR values.

sequence	frames	overall PSNR [dB]	kbps 100%	kbps 85%	kbps 70%	description
ice	236	32.53	243	207	170	about 15 people are ice skating, each person moves at different speed
football	256	29.53	591	502	414	football game with still camera on game play, then a jerky high motion camera pan follows a thrown ball
city	296	29.49	241	205	169	smooth and slow camera pan over city skyline, recorded from an airplane

Table 4.9: MPEG-4 reference streams used for B-frame prioritization comparison

The video sequences were streamed via RTP/UDP with retransmissions enabled [78] and the *keepRate* (simulating the available network bandwidth) was set to 85% and 70% of the average video bitrate, respectively. This means a reduction of each video second's bitsize ($VsecBS$) by either 15% or 30%.

An average PSNR calculation across all frames (either dropped or kept) would lead to minimal differences between the proposed algorithms, since frame droppings are rather seldom and distributed over the video. So Figure 4.10 only shows the quality reduction averaged over all really dropped frames in comparison to the original quality. Focussing on only the dropped frames reflects the human perception better, since humans do not watch a video as an "average", but are distracted by each hick-up or short quality loss.

As expected, QCTVA BFE and WFE sketches the high and low limits and QCTVA BFE also always outperforms timely uniform distribution for all tested video sequences. For a *keepRate* of 85%, QCTVA BFE performs best with highly optimal

results for the upper layers of the modification lattice. Still, the more adaptation is needed (eg. *keepRate* = 70% or lower), the more QCTVA BFE or any other prioritization scheme converge to the base layer of I- and P-frames and therefore deliver even more similar results.

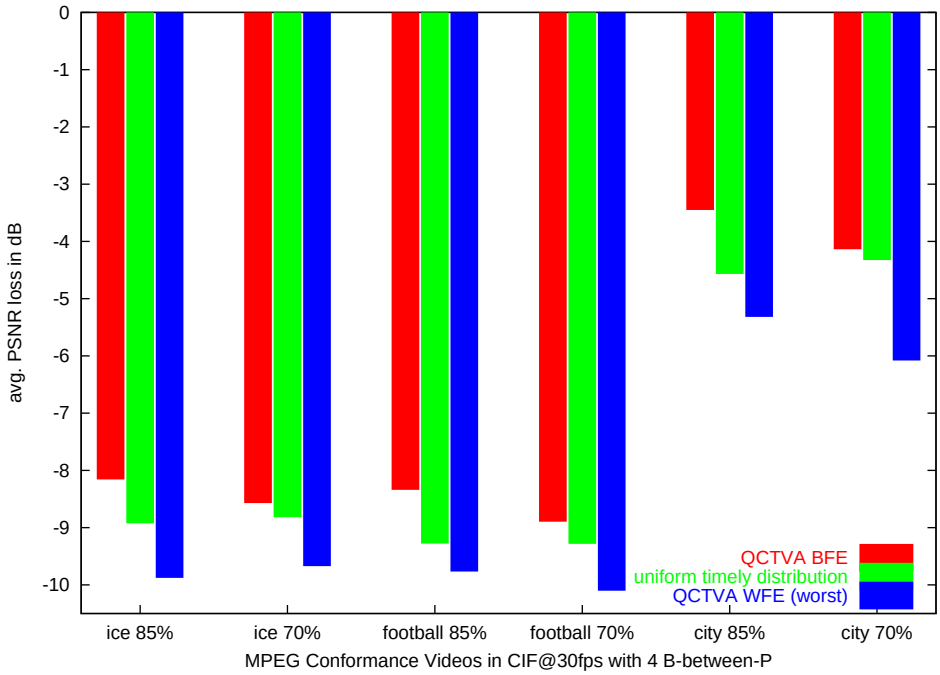


Figure 4.10: Quality reduction on dropped frames for *keepRate* 85% and 70%

For the test sequence *ice* with a *keepRate* of 85%, timely uniform distribution is exactly in between QCTVA BFE and QCTVA WFE. The video shows different motion patterns within a frame for each person sliding over the ice. Since PSNR changes are inherent all over the video, but not directly related to frame-per-frame motion, a timely uniform distribution performs reasonably, though not perfectly.

For the *football* scene, QCTVA BFE outperforms timely uniform distribution even more, since the motion is unevenly distributed over the video. First, there is a still camera on the melee of players. Here, the same rules as for *ice* apply. But then, all of a sudden, a player makes a pass and the ball flies all over the field. The camera tries to follow the ball, which introduces high motion for a short time. QCTVA BFE is capable to optimize priorities for this fast scene and drops frames at the slow motion scene. Timely uniform distribution drops frames uniformly over both scenes, which

causes undesired frame loss also in the fast scene.

For the all-time slow-motion video *city*, the average PSNR loss is very low for all three algorithms. Still, since the scene is recorded with a hand camera from within an airplane, there are some bumpy camera moves. Those are better detected and prioritized by QCTVA BFE than timely uniform distribution.

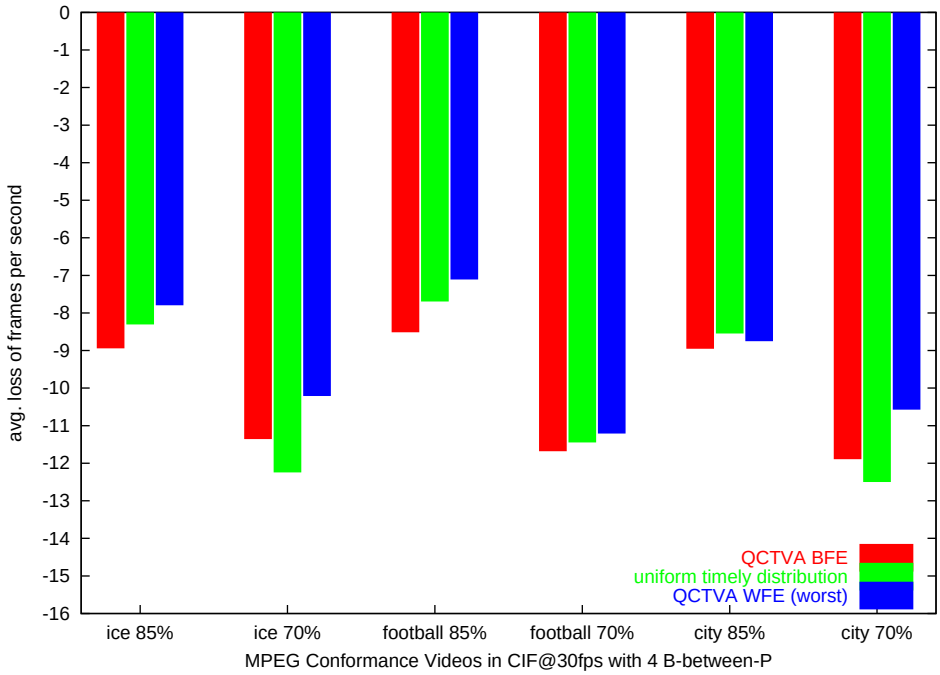


Figure 4.11: Varying frame rates for *keepRate* 85% and 70%

Though the above *keepRate* is always applied to an average bandwidth and each pattern is adapted to the same value, it is a well-known fact, that MPEG-4 is more of a variable bitrate coding scheme than a constant one. So the average bandwidth does not represent each single $VsecBS$, and is even less apportionate to each frame. B-frame sizes vary because of varying image differences to their reference frames (eg. amount of motion between frames) and further by their distance to their reference frames (see Section 4.3). So whenever different B-frames were chosen for dropping, they offer a varying proportion of bandwidth scalability.

Figure 4.11 demonstrates, that the frame rate is not directly connected with adaptation quality. Note, that the frame size of an important B-frame will be proportionally larger, since it has to code more information.

Timely uniform distribution is more or less random on frame sizes, since it only adheres to the mere goal of uniform dropping over time. So it will drop small-sized but also large-sized frames at will, regardless of their importance. This fact makes it impossible to relate frame rate with the qualitative results.

QCTVA, in either BFE or WFE manner, selects priorities only based on their PSNR importance, which ignores frame sizes. QCTVA BFE will always prefer dropping unimportant ones, so it will therefore have to drop more than QCTVA WFE, because BFE prioritizes important frames, offering higher bandwidth scalability.

QCTVA also ignores such frames that are just blown up because of high reference distances. Given that such an overly large frame does not offer high PSNR importance, QCTVA will drop it and accidently gain higher bandwidth scalability, or vice versa.

To further optimize prioritization results, fast prioritization algorithms have to be found which try to optimize visual performance in terms of PSNR but also keep in mind the varying frame sizes and the resulting low frame rates.

4.7 Conclusion and Future Work

The previous section showed the inherent superiority of QCTVA BFE, which is based on the fact, that QCTVA BFE can choose the high priority frames out of a large pool of B-frames. So with $P \xrightarrow{4B} P$ and a *keepRate* of at least 85%, it is just a matter of computational power to prioritize B-frames for perfect PSNR results.

As discussed in Section 4.3, the number of B-frames increases the needed bandwidth to keep a certain PSNR quality or – to keep bandwidth – slightly decreases quality (see Table 4.10).

As stated above, QCTVA outperforms other prioritization algorithms, only if there is a large pool of B-frames and effectively only a few have to be dropped. Otherwise, all algorithms will converge to the baselayer of remaining I- and P-frames.

For a real streaming scenario, we can choose temporal adaptation only for a rather high *keepRate* of eg. $70-85\%$, which gives the best results for QCTVA. If the network bandwidth falls below those 70%, we should better switch over to another stream at a lower bandwidth with lower quality but standard frame rate again.

So if stream switching is possible in the given environment and the switch ranges are at about 30%, only a maximum of 30% of frame dropping is needed. For this, we might also take into account timely uniform distribution. Here, timely uniform distribution could easily be used with a frame pattern of $P \overset{1B}{\rightarrow} P$, since this still gives enough bandwidth scalability.

kbps	pattern	I-/P-quant	B-quant	bandwidth scalability	droppable B-frames	overall PSNR [dB]
241	$P \overset{1B}{\rightarrow} P$	16	15	38.2%	15	32.69
243	$P \overset{4B}{\rightarrow} P$	16	28	61.5%	24	31.63

Table 4.10: Comparing sequence *ice* with $P \overset{1B}{\rightarrow} P$ and $P \overset{4B}{\rightarrow} P$ at same bandwidth

Exemplarily shown on the conformance stream *ice*, this change to the $P \overset{1B}{\rightarrow} P$ pattern would immediately increase the quality by +1 dB, so the gains from a $P \overset{4B}{\rightarrow} P$ pattern average quality loss of QCTVA measured as −8.16 dB in comparison with −8.92 loss for timely uniform distribution are not valid any more, when the full stream is coded with a +1 dB PSNR and "only" timely uniform distribution is applied (see Figure 4.10).

Note, that this above average quality loss will still be outperformed on some unevenly distributed motion scenes, so QCTVA (and the associated overhead) should not be neglected. The optimum video production would use QCTVA for uneven motion scenes, where significant quality gains can be expected, especially when coded with $P \overset{4B}{\rightarrow} P$. For all other scenes, timely uniform distribution with a pattern of $P \overset{1B}{\rightarrow} P$ is sufficient in terms of processing requirements. Using $P \overset{1B}{\rightarrow} P$ is further preferable because of better overall PSNR results at the same bandwidth. Still, analyzing and preparing a stream for optimum prioritization has to be done off-line and can be very time-consuming. A good middle course has to be found for each specific application field.

CHAPTER

5 Stream Switching

5.1 Overview

5.1.1 Limitations of In-Stream Adaptation

Various adaptation methods like frame dropping are applied to compensate fluctuations in the available network bandwidth. Still, all known adaptation methods have their limit. So eg. dropping more than 50% of all available frames will lead to very choppy presentation results. This even ignores the fact that enough B-frames ought to be available, otherwise I- or P-frames will be dropped, which also leads to massive decoding errors.

These limitations in adaptation range do exist for all available adaptation methods that work on a certain stream, either by reaching the base layer (so dropping all available enhancement data) or by reaching an intolerable qualitative result. If not quality is the limiting factor, then inherent coding overhead for adaptable content forces codecs to stay within reasonable adaptation ranges. Eg. MPEG-4 Fine Granular Scalability (FGS), when coded with a very small base layer, suffers of over 2 dB PSNR loss at high bitrates (when using very much of the available enhancement layer), in comparison to the non-scalable MPEG-4 codec [32].

Further, subjective tests [61] have shown that users prefer eg. a lower resolution video at better quantization in comparison to a higher resolution video with many quantization errors, assumed that both streams are coded with the same bandwidth requirements. Also for lower bandwidths, a reasonable frame rate of at least 15 fps has to be maintained, choosing a lower resolution or quality in exchange.

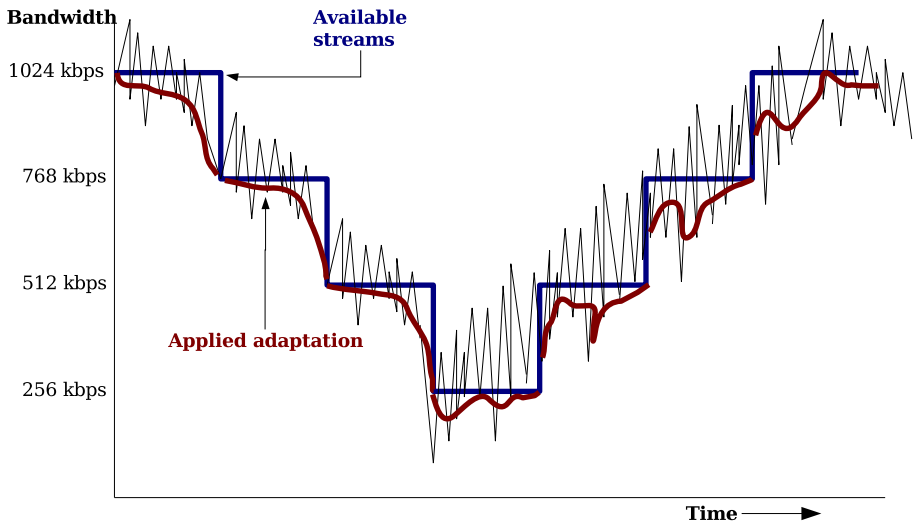


Figure 5.1: Combining adaptation and stream switching

To overcome those limitations and offering better user experience, when the available network bandwidth falls below or exceeds a certain threshold, the actual stream should be changed to another one, better fitting the given network situation. This is common use for modern commercial video streaming systems like RealMedia's RealPlayer or Microsoft's MediaPlayer in connection with the appropriate servers. [79] analyzes the usage and functionality of Real's stream switching support called *SureStream*, where the sender has multiple (statically pre-coded) versions of the same video content available. A more general view is given in [80], where stream switching is also discussed between media sources, so a streaming server might completely hand over the streaming to another (eg. less congested) streaming server, when this server breaks down or is overloaded.

Figure 5.1 shows a hypothetical changing network bandwidth curve, where streams are switched accordingly. Further, a more fine-grained in-stream adaptation is applied, to react further to the inherent bandwidth fluctuations. When bandwidth falls too much and in-stream adaptation cannot cope with it any more, a new stream has to be chosen. This approach will be discussed and evaluated in detail in the following subsections.

Before doing so, we have to outline some requirements for stream switching. Based on [81], the most important ones are:

- *Minimal receiver perturbation*, so the process of switching has to be seamless, introducing none or minimal disturbance from a player point of view. This also includes the avoidance of gaps in the media stream caused by missing packets, undecodable frames because of missing reference frames, or lack of synchronisation information between multiple video/audio object streams in the same scene. This requirement enforces special effort in stream switching environments and excludes a simple (but automated) session teardown followed by restarting another session, since this would take too much time (round trips) and would make the switch point very obvious to the user.

- *Minimal network perturbation*, by avoiding to send redundant data or unnecessary retransmissions or forward-error correction (FEC) information (for the already abandoned stream). When congestion occurs, down-switching has to be performed immediately and the network has to be disburdened as fast and as much as possible, to allow all intermediary routers to stabilize their buffers again. For up-switching, the previous restrictions can be loosened, since additional network bandwidth is available anyway.

5.1.2 Thin Streams versus Stream Switching

Many investigations have been done into splitting a scalable video into multiple layers and distributing them in a multicast fashion. Multicast is only good for real-time video broadcast, meaning that hundreds of clients are watching exactly the same video second at a time. Then the clients can choose, how many layers their network connection can cope with, and book the layers up to a certain level [82] [83] [84] [85].

Because of coding overhead and coarse-granularity of layers, stream switching does outperform the well studied academic approach of thin streams [86] [87]. Further, IP multicast, although embraced by many researchers, is not everywhere available in today's Internet (also because router companies do not offer support, or providers have turned multicast off in the devices' default settings). Moreover, multicast is not useful for supporting video on demand (VoD). Because of these reasons, eg. Real's or Microsoft's servers prefer to stream different versions of a video stream in parallel (simulcast), which then also might be requested at different points in time, hereby

offering VoD according to each client's personal needs.

5.2 Possibilities of Distribution of Functionality

5.2.1 Client vs. Server Initiated Switch

In general, when packet loss is measured, there are two ways of performing stream switching: either triggered directly by the client or by the server, where according to the received information, the server decides on the necessary amount of adaptation or initiates a stream switch.

Initiating (requesting) adaptation and switching at the client would introduce significant complexity to the client software, knowledge of the available streams, algorithms for detection of the ideal switching time and so on. In the following, we want to keep the client as simple as possible, so we let the server do all the decision taking and preparation.

Note that, to enable a client-server streaming environment with server-side stream switching (and adaptation in general), the available network bandwidth has to be accurately measured by a rate control system. This system has to hint the server about packet loss and actual (or even future) network behaviour (see Chapter 6).

5.2.2 Client Transparent vs. Non-Transparent Switching

Stream switching can be conducted using different streams encoded with the same codec (eg. MPEG-4) but at different bandwidth rates and/or resolutions and/or frame rates. Since most codecs have a certain bandwidth range where they perform best, different bandwidth scenarios can require also varying codecs for the different streams.

So the available streams used for stream switching (further called the *switch set*) might vary in different domains, where switching between those domains can happen without informing the client (*client transparent*), or the client has to be informed a priori, to be able to prepare for the new stream (*non-client transparent*).

- *Temporal domain.* A streaming video player, when set up with a certain decoder like MPEG-4, can cope with a dynamically changing frame rate (including

changing frame patterns but also frame loss), as long as frame dependencies are staying intact. For this reason, the server can send another stream of the same codec but at a different frame rate, without breaking the decodability at the client side. So this switching is *client transparent*, and no further signalling is necessary.

- *SNR domain.* Further, a video player based on MPEG-4 can cope with dynamically changing bitrates, so the server can switch within different quantization levels (which results in lower bandwidth), without breaking the decodability at the client side. Actually, when a stream was coded with constant bitrate, each frame is coded with a different quantization level to stay within the given bitrate bounds, anyway. Again, this switching is *client transparent*, and no further signalling is necessary.

- *Spatial domain.* It is becoming more complicated, when the stream resolution changes eg. from CIF to QCIF. Here the client decoder has to detect the new resolution. Either it changes the displaying window size or it upscales the lower resolution to the higher resolution. The latter is expected by the user to save him/her from unnecessary perturbation because of display resizes [88]. Depending on available processing power and hardware acceleration, various upscaling algorithms, deblocking filters and smoothing filters (see Subsection 3.2.3 on page 35) can further increase perceived quality.

- *Codec domain.* Obviously, whenever the codec has to be changed from eg. MPEG-4 to H.263, the client terminal has to be informed a priori, so it can initialize the new decoder and pass the new stream data to the correct decoding engine (either in hardware or software).

Critical Discussion The above mentioned switching domains are only partially available in actual video streaming systems like Real [79] since some domains pose severe implementation problems.

Whenever a system supports video streaming over unreliable RTP/UDP, *temporal switching* is available since it has to cope with packet loss, which will lead to lost frames (as long as no retransmission or forward error correction is implemented). Changes

in quantization are normally not problematic since they are inherently anchored in modern codecs like MPEG-4 or H.264, making *SNR switching* a widely used solution. Although our *ViTooKi* MuViPlayer is capable of performing an automatic upscaling using nearest neighbour interpolation after receiving a new codec header, other players like the commercial (but freely available) QuickTime fail to do so and crash. So switching in the *spatial domain* is available only in limited quality for today's client terminals.

Changing between different codecs is not very likely, not only because of missing support of different codec libraries within one player, but also because of copyright and marketing issues. *ViTooKi* [56], which is leveraging support of various codec libraries, would basically be capable of codec switching, but this feature was not implemented because of complexity reasons.

5.3 Switching in the Temporal, Spatial, and SNR Domains

In the following, we want to propose a new way of combining switching streams with dynamic *in-stream* adaptation. The switch set consists of different quantization and resolution streams and switching will be used as a coarse grained reaction to buffer and network problems. At the same time, fine-grained dynamic *in-stream* temporal adaptation is applied to compensate small bandwidth fluctuations. We will show that this novel approach substantially improves visual quality and keeps the client buffer in a more stable state than just coarse grained stream switching.

The main challenge for video streaming in lossy environments is to optimize user perception. The most important rule of thumb is to never let the client buffer run out of data. To ensure this, when the buffer level gets critical, the system switches to a lower bandwidth stream, so more video data can be sent within one streamout second, which fills up the buffer again.

If the buffer is within acceptable bounds but the available network bandwidth does not reach the needed streamout bandwidth, according to Chapter 2, we have to adapt the video data accordingly, to prevent a buffer underrun also in the future. Using today's available codecs, the cheapest and (at least in this work) well-discussed

in-stream adaptation method is temporal adaptation. Anyway, when temporal adaptation cannot compensate a severe bandwidth reduction (the remaining frame rate is too choppy at eg. below 15 fps, or no more B-frames are available), stream switching has to be performed.

On the other hand, when the buffer level is overly high, the system can switch to a higher bandwidth stream for actively draining the buffer but gaining better quality, which, under a normal buffer fill level, would not fit the given network bandwidth.

5.3.1 The Test Environment

The following evaluations were done with the *Big Show Both* video, encoded at various quantizations and resolutions using the Microsoft reference software for MPEG-4. The video has 13000 frames with one I-frame each 30 frames, using a static pattern of $P \overset{1B}{\rightarrow} P$, which means `IBPBPBP...PB` (see Table 4.1 on page 47). Playout of the video was done at 25 fps, which results in a video length of 520 seconds (8:40 minutes).

resolution	kbps	I-/P-quant	B-quant	overall PSNR [dB]	relative PSNR diff	absolute PSNR diff
CIF	704	12	12	30.19	0	0
CIF	496	16	16	28.75	-1.44	-1.44
CIF	352	21	23	27.45	-1.30	-2.74
QCIF	241	12	16	22.54	-4.91	-7.65

Table 5.1: Used switch set of streams with varying resolution and quantization

Table 5.1 lists the used variations, further called the *switch set*, representing the different variations available. The average bandwidth in kbps is always decreased by close to 30% for each consecutive variant. All streams are streamed with a 15% increased $streamBW$, to compensate for variable bit rate variation (see Chapter 2).

It was not possible to encode a CIF version to reach a bandwidth of 241 kbps (which is 30% below 352 kbps), so some other means of reduction had to be taken. Since this work is focussing on the MPEG-4 video codec, using the reference encoder, it was not an option to change the codec to eg. H.263. So spatial reduction is obviously the most useful way, when further SNR reduction fails (quantization is already at its lowest bounds). Further, having (at least one) QCIF version in the switch set will

enable the server to better fulfill special requirements from low resolution clients like cell phones or handhelds.

Production of the QCIF version was performed as follows: the original CIF video was downscaled to QCIF in the uncompressed (YUV) domain using the simple and fast *nearest neighbour search* algorithm as described in section 3.2.3. After decoding at the client side, this QCIF video was upscaled again to CIF (using *nearest neighbour search*) and then compared to the original CIF version. This results in an effective relative PSNR loss of -4.91 dB to the next higher quality in CIF resolution. Please note that better scaling methods will lead to significantly better results (see section 3.2.3), but those were out of scope and computational power of this work and test environment.

Similarly to the steppy bandwidth curve in Figure 5.1, the gradual degradation and increase of the available bandwidth was simulated with the Linux traffic shaping class *leaky bucket filter*, simulating a 50% bandwidth variation range from 700 kbps down to 343 kbps. Starting at 700 kbps, the bandwidth is reduced by 30% every thirty seconds, so the next step will be 490 kbps for 30 seconds, then the lowest step of 343 kbps is reached. After another 30 seconds, the bandwidth goes up again to 490 kbps, until it reaches the top level of 700 kbps. After that, the whole process is repeated infinitely. Because of steps based on a percentage value, this gives a steppy, but U-shaped bandwidth curve.

In Figure 5.2(a), the spiky curve shows the measured streamout bandwidth, which coarsely follows the traffic shaped bandwidth steps. Please note, that our streaming environment, which was implemented within ViTooKi, was capable of estimating the available bandwidth and the client buffer fill level based on knowledge on sent packets and RTP NACK messages as described in Chapter 7.3. Retransmissions were also enabled, so lost packets of important frames were sent again, which highly increases visual quality. Exact matching of the traffic shaped bandwidth steps is impossible because of packet size boundaries, so the used bandwidth spikes are always varying within one MTU (maximum transfer unit). Further, since TCP-friendly behaviour is envisioned, bandwidth is always re-measured and then the streamout bandwidth is adjusted in an AIMD fashion (additive increase, multiplicative decrease). This leads even more to the displayed spiky streamout bandwidth and shortly delayed reaction.

5.3.2 Coarse-Grained Stream Switching in the Spatial and SNR Domain

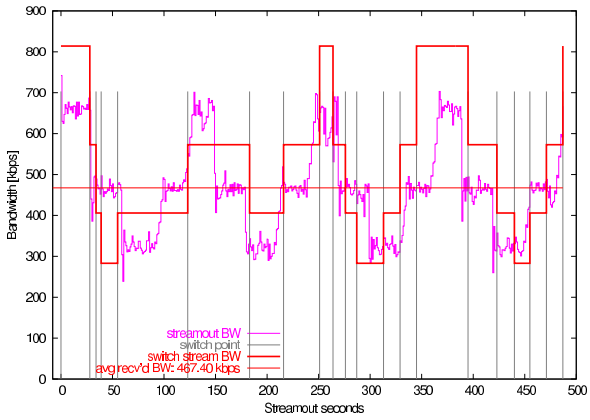
In Figure 5.2(a), the red line for stream switching and the currently chosen stream is coarsely following the network bandwidth. The decision on stream switching is based on the available bandwidth and the client buffer. If the available bandwidth is very far off the necessary stream bandwidth (below 30 %), or if the available video seconds ($Vsecs$) stored in the client buffer fall below 5 video seconds, the current stream is not acceptable any more and a lower bandwidth stream has to be chosen.

Despite normal down- and upstepping to select the est fitting stream in bandwidth from the available *switch set*, further switching behaviour is noted. At streamout seconds ($Ssec$) 28, 264, and 440, Figure 5.2(a) shows a down-stepping to a lower quality stream version, triggered by very low client buffers. This nicely correlates with the buffer fill level shown in Figure 5.2(b), which drops below the limit of 5 video seconds.

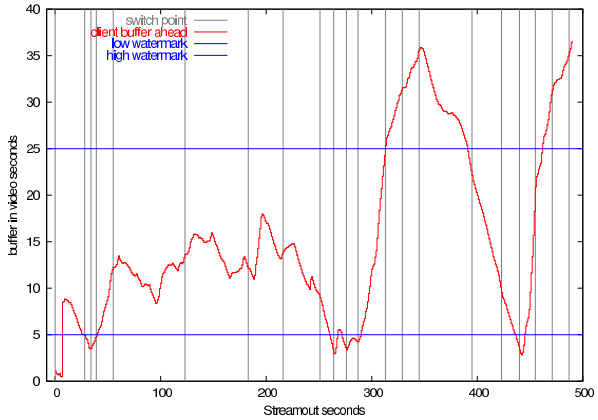
If the client buffer fill level is overly high, this can be used to enforce the system to use a better quality video stream, even if network bandwidth is low. Our system is configured to switch up to a higher quality stream whenever the client buffer exceeds 25 video seconds. This is shown at $Ssecs$ 313 and 471, triggered by high client buffers shown in Figure 5.2(b).

When buffers are very low, the lowest quality stream of our switch set has to be chosen. Unfortunately, this is a QCIF stream, which has to be scaled up again on the client side. This obviously gives very low PSNR results (see Figure 5.2(c)), but, on the other hand, might be a perfect fit for a low-resolution client in other cases.

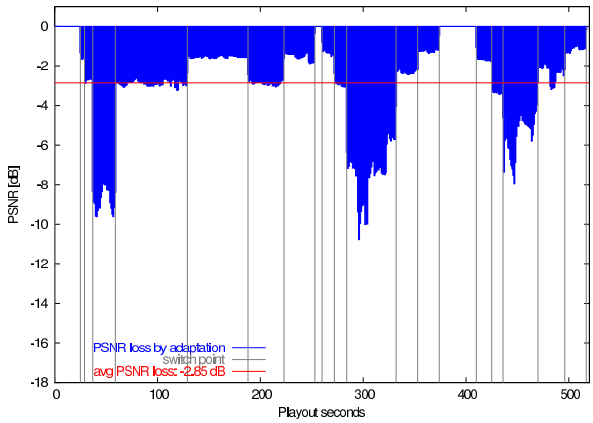
Anyway, the overall average PSNR loss for all 520 video seconds is -2.85 dB (in comparison to always streaming the best quality stream).



(a) Network bandwidth and switching behaviour



(b) Client buffer fill level of available video seconds



(c) Qualitative reduction caused by adaptation

Figure 5.2: Stream switching without any further in-stream adaptation

5.3.3 Adding Fine-Grained Temporal Adaptation

Instead of using a QCIF version, the chosen method for commercially available solutions like RealVideo [24] is to have low quality streams based on statically built coarse grained temporal adaptation (eg. from 25 fps down to a 12.5 fps version). To also cover this behaviour in our measurements, we have simulated this by using the lowest quality $P \overset{1B}{\rightarrow} P$ CIF stream from our *switch set* with I- and P-quantization 21 and a B-quantization of 23, which leads to an average bandwidth of 352 kbps. Then we removed all B-frames, which results in a framerate of 12.5 fps and an average bandwidth requirement of 232 kbps (which approx. leads to the wanted 30% reduction). With an average PSNR value of 24.73 dB, this stream suffers a relative PSNR reduction of 2.72 dB (instead of -4.91 dB with the QCIF version) as compared to the original (higher step) stream.

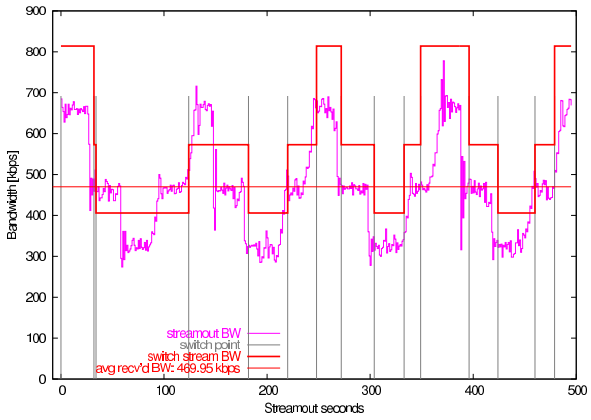
Although this low-framerate version gives better PSNR results than the QCIF upscaled version, it lacks flexibility on low-resolution clients. For this reason, this work will sustain from using coarse grained temporal versions, but will show the ability of improving switching strategies by including finer grained temporal adaptation within the actually chosen video stream. This will always give equal or better results than simple stream switching.

Further, the following examples will show that fine-grained frame dropping in some cases renders unnecessary to switch down to the very lowest quality, so it does not matter if it would have been QCIF or a statically frame-rate adapted version. Note, that our already defined *switch set* was chosen to offer $P \overset{1B}{\rightarrow} P$ streams, so there is always the possibility of finer grained, dynamic temporal adaptation within the range of 25 to 12.5 fps.

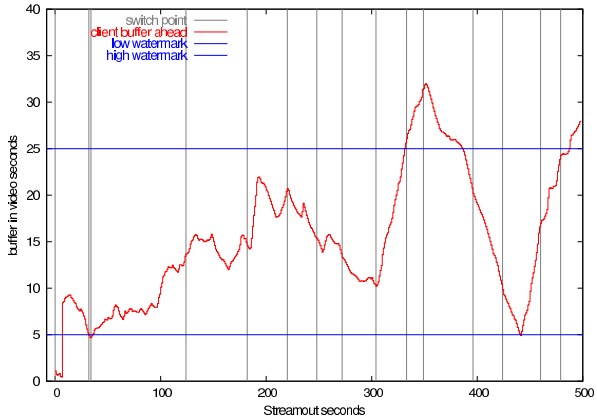
All streaming tests with temporal adaptation where done using timely uniform distribution, so we decided to only use $P \overset{1B}{\rightarrow} P$ streams, although knowing, that $P \overset{4B}{\rightarrow} P$ would offer more adaptation possibilities. Especially when leveraged with a quality based prioritization approach like QCTVA, also slightly better overall PSNR results would be possible. Still, according to our findings in the previous chapter, timely uniform distribution gives reasonable results with up to max. 50% frame rate reduction (eg. down to 12.5 fps), not to mention the faster prioritization process, when performed in the compressed domain.

Adding dynamic temporal adaptation, Figure 5.3(a) shows the better switching behaviour, when the streaming environment is exposed to the exact same bandwidth curve from the previous measurement in Figure 5.2. It was never necessary to fall back to the lowest bandwidth stream (may it be either QCIF or a 12.5fps CIF stream). This was achieved by intermediate and dynamic temporal adaptation, as shown in Figure 5.4. Note, that it was never necessary to drop the frame rate below 18 fps. Further, there were many situations, where no temporal adaptation was necessary at all, so we were keeping the original frame rate of 25 fps.

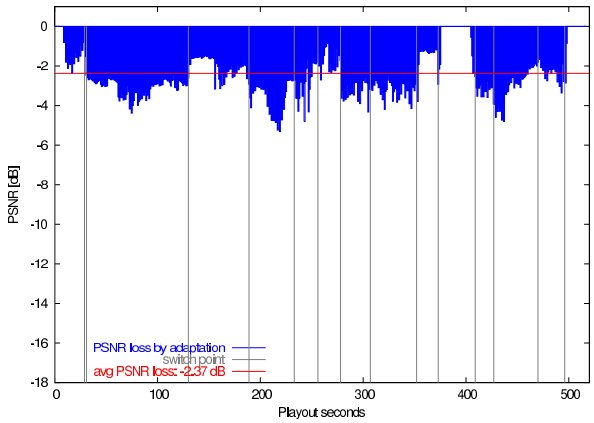
Figure 5.3(b) shows that, because of temporal adaptation, the client buffer is more stable and always within safe bounds. The average PSNR loss (see Figure 5.3(c)) is at -2.37 dB, so it is also better than simple stream switching shown in Figure 5.2(c).



(a) Network bandwidth and switching behavior



(b) Client buffer fill level of available video seconds



(c) Qualitative reduction caused by adaptation

Figure 5.3: Stream switching with temporal adaptation

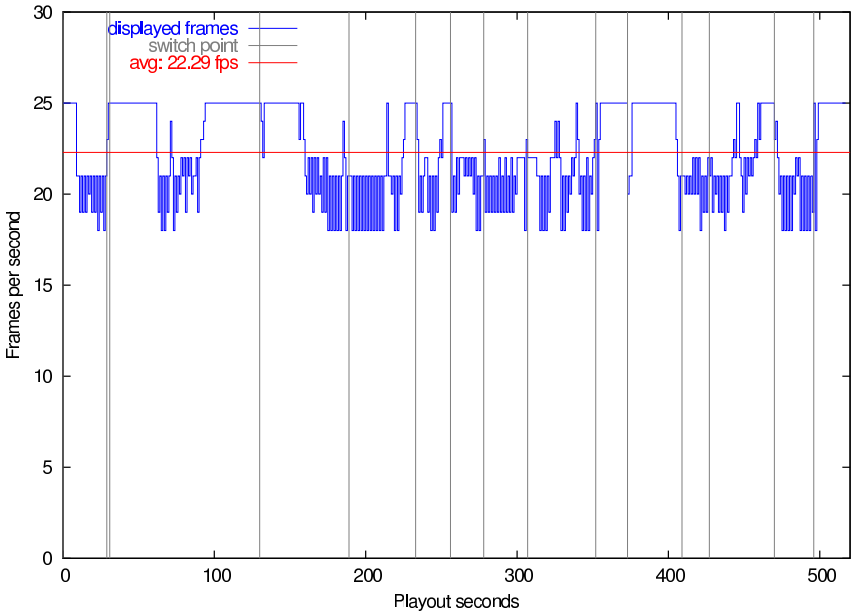


Figure 5.4: Visual frame rate per playout second after temporal adaptation

5.3.4 The Optimal Switching Point

As stated before, to perform switching, we have to have a well-selected *switch set* of available videos. Further we need correct and immediate feedback on the actually available network bandwidth, which also allows us to estimate the client buffer fill state.

Still, there are some other issues how to find the optimal point in time to decide on switching. It is especially important to synchronize to the next possible I-Frame in the *new* video stream, because future decoding has to be permitted without any missing reference frame information, which would result in decoding errors.

For other codecs like Real's *SureStream*, specially encoded S-frames (switching frames) enrich the frame patterns to allow earlier switching without wasting too much bandwidth in comparison to I-frames [24]. Those S-frames are also introduced into new codec development efforts like the fine-grained MPEG-4 scalable codec (FGS) [89] or MPEG-4 AVC [90]. Please note that MPEG-4 in the Advanced Simple Profile [29] (as it was used by our measurements) is only capable of I-, P- and B-frame support, so there is room for further optimization of our experiments when changing to another codec. So when we speak of I-frames in the following, this is meant interchangeably with any possible synchronization frame, either I-frame or S-frame.

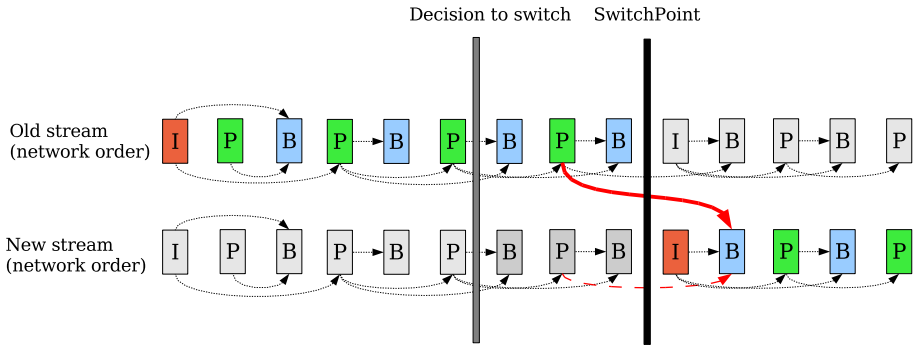


Figure 5.5: Streams are switched at the next available I-frame

When the system has made the decision to switch, it has to find the next possible *switch point*, where it has to take special care for decodability of referencing frames and resynchronization to the next available I-frame. Further, for any given timestamp, no frame should be sent doubly, so each frame is sent out only in one single quality.

For that, the actual streamout position (the frame number) in the *old* stream has to be used as the start point in the *new* video stream. Starting at this point in the new stream, the system has to step forward until the next available I-frame is found. All frames before this I-frame are continued to be streamed out by the *old* stream stopping closely before this I-frame, and then, finally, the switch-over to the *new* stream is performed (see Figure 8.8).

Unfortunately, when switching at the next I-frame, there is a problem for the following B-frames in the new stream: Each B-frame needs two reference frames, but the new stream only supplies the decoder with one, namely this first I-frame. So the old P-frame from the old video is still in the decoder buffer and will be used as the second reference frame. In the down-switching case, this even results in better quality, since the B-frames are able to use a higher quality reference frame for their motion vectors. In the up-scaling direction, those first B-frames might suffer from a little loss in quality, but since both videos at this point in time are holding absolutely the same video content, it will not be visible as any decoding error.

An extension to this naive search of the next switch point could be taking also into account the available client buffer fill level. Especially when the client buffer is overly full (eg. it holds more than 25 video seconds of data in advance), we will switch up to a better quality, although this is not reflected by the available network bandwidth. Keep in mind that the buffer is overly full with low-quality frames, so it also makes sense to drop some later video seconds and to switch streams at an earlier point in time, so eg. flush the buffer so that it only holds 10 video seconds of low quality but is receiving new (high quality) video data after switching is performed. This gives faster user satisfaction by increased quality but obviously means to accept redundant quasi-retransmission, since the same frame was sent in different qualities and the system decides on the latter and higher quality one. This behaviour might be unwanted if the client has to pay for network usage and adds some extra algorithmical overhead for client and server.

For more transparent behavior, our ViTooKi streaming environment refrained from flushing the buffer and forcing unnecessary quasi-retransmission. But as described in the last section and as shown in the figures, we took into account the actual buffer fill level for making the switching decisions. If the buffer fell below the

low water mark of 5 seconds, we enforced down-switching, but on the other hand, if we exceeded a 25 seconds high water mark, we switched up to a higher quality.

5.4 Conclusion and Future Work

Although the combination of stream switching and B-frame dropping offers better buffering behavior, the user is subjected to an ongoing fluctuation in the visual frame rate. First subjective evaluations have shown that this loss of frame rate is not critical, as long as variations always stay in the upper frame rate ranges (somewhere between 25 and 18 fps). Although PSNR loss in our experiments is severely impacted by missing frames (which does not reflect the real visual experience), the proposed combination of stream switching and the temporal adaptation only result in an overall average quality loss of 2.37 dB PSNR, whereas the simple switching only scenario suffers from -2.85 dB PSNR.

Still, this raises the question, if PSNR is a suitable measure for temporal differences in a video. Until now we take it as the only really widely accepted method, and ignore others like *Just Noticable Difference (JNDmetrix)* or *Visual Quality Metric (VQM)* [91, 92] because of their complexity and unthoroughly tested state. Further, they are not easily and freely available to the open public, so it is hard to deliver comparable results for other studies.

The used *switch set* of 4 streams, so the pool of available stream variations was hand-chosen from hundreds of pre-encoded streams, using the Microsoft reference software for MPEG-4 with a static $P \overset{1B}{\rightarrow} P$ frame pattern. To provide a streaming server with a good *switch set*, it is important to use variable frame patterns and variable quantization steps, so an encoder can be set to produce a certain (fixed) average bandwidth stream for each -30% step. It is not feasible for real production servers to produce dozens of streams and choose the best ones.

For production servers, most of the actually available commercial or open-source codecs like OpenDivX [93] or FFmpeg [65], are offering better qualitative results at lower bandwidths, even using double-pass coding to optimize their results, choosing of variable frame patterns and so on.

On the other hand, variable frame patterns also vary in the amount of scalability

(available B-frames within one video second) and will lead to more irregularily arriving I-frames which can be used as *switch points*. This makes the reaction time between detecting and really switching highly dependent on the actual position in the video stream. Therefore, for these measurements, a static pattern over the whole video was chosen.

Using all of the above mentioned optimizations and adding special S-frames (see above) will further improve the performance and positive user perception of stream switching, where in-stream adaptation always will help on stabilizing buffers and/or preventing unnecessary stream switches.

This work used temporal adaptation as a proof of concept, but all other (not yet widely) available in-stream adaptation methods of various scalable video codecs (eg. MPEG-4 FGS or wavelet coding) are even more capable of stabilizing buffers and/or preventing unnecessary stream switches, because they work on even more fine grained steps.

Finally we conclude that in-stream adaptation using any scalable codec in connection with stream switching is a perfect combination, when there are enough streams available in the *switch set*, so the in-stream adaptation can be performed in its optimal range (eg. stay between 25 and 15 fps). Future work will have to investigate the impact of such a multi-dimension adaptation system with respect to server load and the number of possible parallel users. Further, evaluations with actually available codecs and with variable frame patterns have to be performed, to find reasonable combinations of stream switching and in-stream adaptation under those conditions.

CHAPTER

6 Multimedia Negotiation and Streaming

6.1 Overview

In this chapter, we will discuss all protocols which are needed to enable a standard conformant client/server multimedia streaming environment based on an unreliable best effort network. This includes sending the consecutive packets over the network with RTP and receiving RTCP statistics on the overall packet loss. Further, ways to communicate and setup a multimedia presentation using RTSP with an included SDP description of the included video and audio streams are explained.

6.2 Multimedia Data Transport with RTP/RTCP

The Real-Time Transmission Protocol (RTP) is defined in RFC 1889 [94]. It provides end-to-end delivery services for all kinds of data with real-time characteristics. To make RTP easy to include in actual networks, it might be based on different underlying standard protocol layers, like ST-II, UDP/IP, IPX or ATM AAL5. The widest spread protocol is UDP, a packet-oriented protocol based on IPv4, which offers checksum services but does not guarantee that packets arrive in order or arrive at all. RTP itself also does not guarantee delivery or prevent out-of-order delivery, but includes packet sequence numbers to reorder incoming packets and other useful information for real-time multimedia data like presentation time stamps. These have to be handled by the application.

IP, and hereby UDP, also offers the interesting feature of multicasting live content,

which means having multiple clients receiving one and the same packet sent by a server. This decreases network load and also guarantees smaller server-side connection maintaining costs (eg. a server just streams its live stream, but it does not care about who is listening). Please note, that this solution is not applicable on *video on demand* systems, where each client chooses its own start time. Since this work is focussing on exactly this *video on demand* scenario, we are not using multicast and maintain multiple connections for each attached client.

6.2.1 Mixers and Translators

Since RTP connections allow multiple clients with different capabilities which are connected over various networks, RTP defines ways of intermediate data adaptation and network transition.

Mixers are positioned somewhere on the route between the server and the clients, and change the incoming data in some means, eg. they change high-quality video to a lower resolution and stream this low-quality video to a client connected via a low bandwidth network. All other clients connected via high bandwidth networks still receive the high-quality video. In the context of this work, this will be called a "multimedia gateway", since it performs simple forwarding but also transcoding on demand.

Since every distinct RTP stream has a distinct RTP synchronization source identifier (SSRC), every re-mixed stream gets a new identifier, but keeps track of the originating sources as a "contributing source" (CSRC), so the client still knows the original source and also possibly negotiates directly with that original source.

Translators forward RTP packets with their source identifiers intact, since they do not change the payload itself. A translator may be used on firewalls which do not allow multicast streams, so the multicast is translated into unicast and directly sent to the clients behind the firewall (enabling all possibilities of authentication and other means of access control). Another typical usage for translators would be as a gateway between underlying networks like eg. UDP/IP and ATM AAL5.

6.2.2 Data Packet Format

The following section describes the RTP packet format, which defines the general fields for multimedia data transmission.

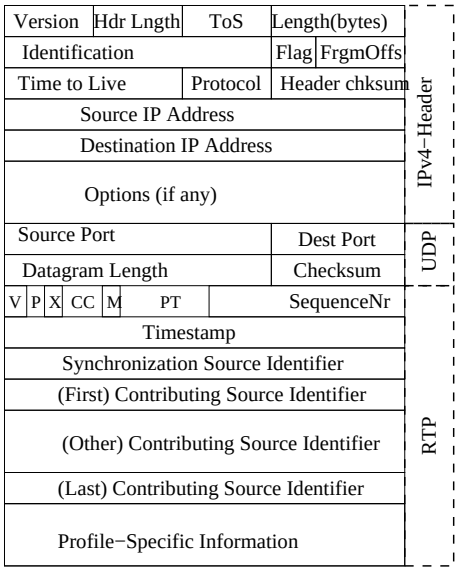


Figure 6.1: RTP header on top of UDP/IPv4

Figure 6.1 shows the RTP header on top of UDP/IPv4. Only the most important fields of the RTP header itself will be discussed here, for more detailed information please refer to RFC 1889 [94].

Payload Type (7 bits) The *payload type* helps the application understand and interpret the RTP data correctly. There are some predefined *payload types* depending on the used RTP profile, eg. for video and audio [5] like GSM, H.261 or MPEG-1 (see section 6.2.4). Eg. for MPEG-4, there is no fixed predefined payload type available, so dynamic ones are generated for each stream.

Sequence Number (16 bits) The *sequence number* is incremented for each RTP packet sent, and may be used by the receiver to detect packet loss and to restore the sequentially correct packet order. The initial value should be a random number, to make known-plaintext attacks on encrypted packets more difficult. Even if the server

itself does not provide packet encryption, the *sequence number* should be randomly chosen, because there might be translators on a packets' way which provide secure tunneling.

Timestamp (32 bits) The *timestamp* has to be generated by a monotonically and linearly increasing clock. To allow sufficient jitter calculations and synchronization accuracy, the clock frequency should be more fine-grained than eg. a given video frame rate or audio sample rate. For security reasons, the *timestamp* should also be initiated with a random number [95]. Over multiple RTP packets, the *timestamp* might also be equal whenever multiple packets were (logically) generated at the same time. Consecutive timestamps might also be out of order, even if their *sequence numbers* are monotonically increasing: eg. MPEG encoded video streams store frames in a different order than they have to be displayed later on.

Synchronization Source Identifier (SSRC) (32 bits) This is a randomly chosen ID for an RTP stream. The *SSRC* has to be unique so a client, mixer or translator can distinguish between different RTP streams. Even though the chance for multiple but equal *SSRCs* is low, RTP implementations must be prepared to detect and resolve collisions.

Contributing Source Identifiers (CSRC) (32 bits) As discussed before, mixers include all originating sources as "contributing" ones for the newly generated RTP stream. The *CSRC Count (CC)* field (4 bits) holds the number of stored *CSRCs*. If there are more than 15 sources, all above those fifteen are lost.

6.2.3 Control Packet Format

To monitor and adjust quality of service of RTP streams, there is the RTP control protocol (RTCP), which is also defined within the RFC for RTP. Such an RTCP packet carries information about packet loss, jitter, and so on. RTCP is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets. Data and control packets have to be two different streams, so when we are using UDP/IP, we use two consecutive ports

(n for data, $n+1$ for control, where n has to be an even number). This also allows monitoring stations, only to receive RTCP packets and gather statistics.

RTCP performs four functions:

1. The main feature is to send information about the behavior of the underlying network where the affected stream is sent over. This feedback can be immediately used for adaptive encodings, to decrease or increase the quality of video or audio or to switch to a different encoding mechanism.

2. RTCP also has information packets where clear-text messages are sent. The most important is the CNAME, which is the unique canonical name for an RTP stream (eg. `jdoe@home:mystream`). So if the SSRC changes because of collision, all participants keep track of the stream by its CNAME.

3. In a multicast setup with multiple clients, each participant sends its control packets to all the others. This is needed to calculate the rate at which the RTCP packets are sent, which assures a reasonable proportion of the amount of RTCP packets versus the amount of RTP packets sent out over a certain time. For unicast, this reduces to packets sent between the client and the server.

4. Functions 1-3 should be used in all application scenarios, but there is a further possible usage. RTCP could be used to keep track of all joined participants in a loosely controlled multicast session, where virtually everyone may connect and disconnect without membership control or parameter negotiation. This becomes possible since whenever a client joins a session and receives data, it will send unrequested RTCP packets to the server.

To keep control traffic low even in a multi-user scenario, it is recommended to dynamically adjust the RTCP traffic to only 5 % of the overall data traffic. Also, the minimum interval between two RTCP packets is recommended to be 5 seconds. To overcome those limitations of too seldom RTCP packets, this work will discuss a new extension on more immediate feedback (see Section 7.3).

RTCP defines not only sender and receiver report packets, but also textual source description packets, a goodbye packet and a flexible application-defined packet. All of those RTCP packet types use the same header specification for the first 16 bytes, but

with different packet types (PT) starting from 200 upwards, so it is easy to identify the different packets.

Sender and Receiver Reports To inform all partners (which, for unicast, is just the server and one client) in a session about packet loss and the actual network behavior like jitter, both client and server are sending RTCP reports to each other. The server sends an RTCP sender report (packet type = 200), whereas all clients send RTCP receiver reports (packet type = 201). The only difference between the RTCP sender report (see Figure 6.2) and the RTCP receiver report (see Figure 6.3) is, besides the packet type code, the 20-byte sender information section with the according timestamps and a packet and byte counter. The *NTP timestamp* stores the globally defined time when the report was sent, whereas the *RTP timestamp* corresponds to the timestamp format used in the RTP packets, which can also be offset by a random value, like it is used in the according RTP packets.

Then a variable number of report blocks (according to the *report count* field (RC)) are attached. They store the *SSRC*, some statistics about *packet loss* and *jitter*, and finally the time of the last sent RTCP packet related to this *SSRC* and the time since that last sent RTCP packet. This is very useful for the receiver of this packet to calculate the roundtrip time.

SDES: Source Description RTCP Packets Since every RTP stream has some cleartext properties like its canonical name, RTCP offers a way to distribute those to all participants. An RTCP packet with SDES information (packet type = 202) is shown in Figure 6.4. Those properties are

- CNAME: the canonical identifier for an RTP stream (eg. `jdoe@home:mystream`)
- NAME: username, who set up this RTP stream
- EMAIL: electronic mail address of the above user
- PHONE: phone number in the international format (leading plus sign (+) and country code)
- LOC: geographic user location

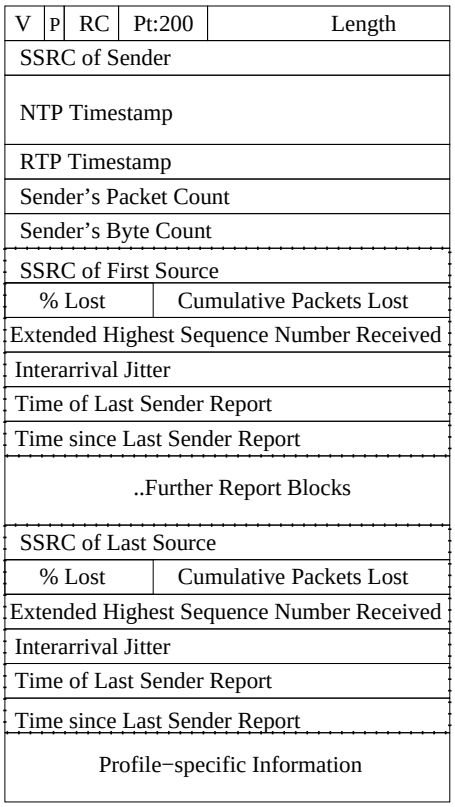


Figure 6.2: RTCP sender report (packet type = 200)

- TOOL: application name and version
- NOTE: notice or status field about the source
- PRIV: private application-specific extensions
- END: indicates the last *SDES item* for this *SSRC/CSRC*.

To keep network load low, there should be an algorithm for sending different SDES packets, eg. always send *CNAME*, but send *EMAIL* every fifth packet, do not send all the others.

BYE: Goodbye RTCP Packets There is a special BYE packet with the *packet type* set to 203. It might also include a text like "camera malfunction". Mixers should forward those BYE packets unchanged and if necessary, send their own BYE packets for their generated *SSRCs*.

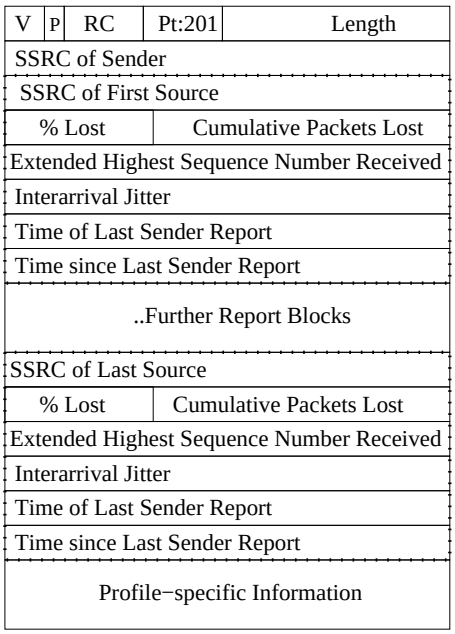


Figure 6.3: RTCP receiver report (packet type = 201)

APP: Application-defined RTCP Packets Finally, there is an APP packet, which is open for experimental use and for application-specific functions. Herefore the *packet type* is 204.

6.2.4 RTP Profile for Audio and Video (RTP/AVP)

RTP itself does not define any behavior and handling of data types. Special profiles describe how to treat different data types, define the exact data packet formats and their packetizing issues. For audio and video, there is already a predefined profile defined in the RFC 1890 [5]. It supports all different audio and video formats and defines how they have to be stored in the RTP payload data area.

Eg. for audio, RFC 1890 defines a default 20 ms packetization rate (which might go up to 200 ms to reflect some formats' bulk frame sizes). The sampling frequency can be chosen out of 8000, 11025, 16000, 22050, 24000, 32000, 44100, and 48000 Hz.

Only some audio and video encodings are shown here, for a complete list refer to RFC 1890 [5]. Newer encodings like for MPEG-4 audio/visual streams are discribed in RFC 3016 [96]. Other packetization formats like MPEG-4 FlexMux [97] are discussed

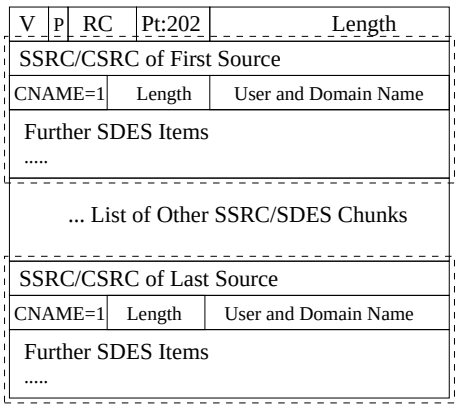


Figure 6.4: SDES: source description RTCP packet (packet type = 202)

and compared in [98].

6.2.4.1 Some Audio Encodings

RFC 1890 exactly describes the packet formats of various audio encodings bit by bit.

G.722, G.723, G.726, G.728, G.729 According to ITU-T Recommendations, those encoding standards define audio encodings with rates between 6.4 kbps up to 64 kbps. Those encodings are used for videophone terminal applications up to Internet telephony in high quality.

GSM, GSM-HR, GSM-EFR The European GSM standard for full-rate speech at 13 kbps (33 bytes for 160 samples), half-rate speech (14 bytes) and extended full-rate speech (31 bytes).

MPA MPA defines the payload for MPEG-1 and MPEG-2 audio in the differently complex layers I, II, and III. The exact payload format for MPEG-1/MPEG-2 audio and video can be found in RFC 2250 [99].

RED The redundant audio payload format "RED" (defined in RFC 2198 [100]) holds not only compressed audio data for the current interval, but also a highly compressed version of the last interval. This allows to reconstruct lost packets in better quality than silence substitution or amplitude/frequency interpolation.

6.2.4.2 Some Video Encodings

The RTP *timestamp* frequency is defined as 90,000 Hz for all the following video encodings. This suffices enough for jitter estimation and calculations with RTCP *timestamps*.

Motion JPEG MJPEG defines a way of storing consecutive JPEG-encoded still-images. Some initialization tables are only sent once and then each raw image data set is sent with only a minimum of necessary header information. Find out more about RTP/MJPEG in RFC 2435 [101].

H.261, H.263 According to ITU-T Recommendations, those encoding standards define video encoding with low bitrates up to 64 kbps. These encodings are used for videophone terminal applications and video telephony.

MPV MPV defines the payload for MPEG-1 and MPEG-2 video and, like MPA for audio, is also described in RFC 2250 [99].

6.3 Media Control and Announcement with RTSP/SDP

Before a client can view any video/audio content, it somehow has to contact the server and find out the possible versions of the video. This handshake also includes passing information about terminal capabilities and channel restrictions, or supported protocol features (eg. RTP with extensions).

To maintain interoperability between various client/server streaming solutions, they all should behave according to the *Internet Streaming Media Alliance* (ISMA) implementation specification [102]. This document standardizes the general behavior of a media server and media clients, taking full advantage of available open standards like RTP/RTCP over UDP/IP, RTSP, SDP and the MP4 file format.

6.3.1 Session Description Protocol (SDP)

The purpose of SDP [103] is to supply information about available media streams and session settings, so receivers of this information can eg. adjust their decoding and displaying environment according to the stream they are going to receive.

6.3.1.1 General Session Description Block Layout

An SDP block includes

- a session name and purpose,
- a time, when and how long a session is active and valid,
- the media streams contained in this session,
- necessary information to receive those media (eg. IP addresses, ports, formats).

An SDP session description consists of a number of lines of text. Each line has the form `<type>=<value>`, where `<type>` is always one single character. There are no whitespaces before or after the `=` sign and every line is concluded with carriage return/line feed (CR/LF).

Table 6.1 shows the generic setup of an SDP block. Optional items are marked with an asterisk (`*`). The order sequence of the items is fixed, so, first of all, there has to be the version (`v=`) item followed by the owner item (`o=`).

6.3.1.2 Important Fields

Please use Table 6.2 as an example of a full SDP session description to find the described fields here.

Attributes (a=) Attributes are either in the form of a property (`a=<attribute>`) or for passing values (`a=<attribute>:<value>`). They might be used on a "global session level" or on a "per media level".

Media Announcements (m=) A session description may contain a number of media description blocks, starting with an `m=` item followed by some optional items, mostly at least some attributes and a bandwidth hint.

Media announcements use the following parameters:

```
m=<media> <port> <transport> <fmt list>
```

- `<media>` might be either `audio, video, application, data, control`
- `<port>` is the transport port, so in the RTP case, this should be an even number in the range of 1024 to 65535. Still, it could be zero if eg. the port is negotiated by some other means than SDP.
- `<transport>` sets the used transport protocol, which is most likely `RTP/AVP` or `UDP`.
- `<fmt list>` stands for one or more subsequent fields with media-specific format information. If set to `RTP/AVP`, it will be the media payload type defined in RFC1890 [5].

Since also dynamic RTP payload types can be used for various media formats, we need some other means to send additional information about the exact codec and sampling rate. This can be done by the following attribute

```
a=rtpmap:<payload type> <encoding name>/<clock rate>[/<enc params>]
```

MPEG-4 is not defined by RFC1890 [5], so MPEG-4 has to use one of the dynamic RTP payload types and has to somehow inform the receiver about the real type of this media stream. To further add specific format information, SDP defines the following attribute:

```
a=fmtp:<format> <format specific parameters>
```

Table 6.2 describes an MPEG-4 video stream with the according `rtpmap` and `fmtp` attributes. First, in the global session level, the creator and the video length (13 seconds) is defined. The first (and only) media block lists information about the needed bandwidth, the codec `MP4V-ES` and spatial resolution of CIF (352x288).

Session description (main) block
v= (protocol version)
o= (owner/creator and session identifier)
s= (session name)
i=* (session information)
u=* (URI of description)
e=* (email address)
p=* (phone number)
c=* (connection information - not required if included in all media)
b=* (bandwidth information)
one or more **time description blocks** *(see below)*
z=* (time zone adjustments)
k=* (encryption key)
a=* (*zero or more* session attribute lines)
zero or more **media description blocks** *(see below)*

Time description block
t= (time the session is active)
r=* (*zero or more* repeat times)

Media description block
m= (media name and transport address)
i=* (media title)
c=* (connection information - optional if included at session-level)
b=* (bandwidth information)
k=* (encryption key)
a=* (*zero or more* media attribute lines)

Table 6.1: General setup of a session description data block

```
v=0
o=StreamingServer 1043923944 926516 IN IP4 143.205.122.54
s=lotr.mp4
e=admin@video.itec.uni-klu.ac.at
c=IN IP4 143.205.122.54
t=0 0
a=range:npt=0-13.23300
m=video 0 RTP/AVP 96
b=AS:1514
a=rtpmap:96 MP4V-ES/90000
a=fmtp:96 profile-level-id=1
a=cliprect:0,0,352,288
```

Table 6.2: Example of an SDP session description data block

6.3.2 Real-Time Streaming Protocol (RTSP)

The Real-Time Streaming Protocol (RTSP) [104] establishes and controls either a single or several time-synchronized streams of continuous media such as audio and video. The stream itself is typically not included, but might be also be interleaved into the control stream. The stream may also be sent via RTP, or any other real-time protocol like RealNetworks RDP.

RTSP provides "VCR-style" remote control functionality, like play, pause, fast forward, reverse, and absolute positioning.

The protocol has a similar syntax and operation to HTTP/1.1 [105], and the default port is 554. Each presentation and media stream is identified by an RTSP URL like `rtsp://video.bigserver.com/new-smash-hit.mp4`

For error codes, HTTP error codes like "`404 File not found`" are re-used, but there are also new ones like "`453 Not enough bandwidth`".

6.3.2.1 Important Methods

In the following, we will list the most important methods to get information about existing streams, setting up streaming sessions and then play/pause/stop them. All examples are based on the ISMA standard [102].

DESCRIBE The DESCRIBE method retrieves the description of a presentation or media object identified by the request URL from a server.

```
C->S: DESCRIBE rtsp://kermit:3128/lotr.mp4 RTSP/1.0
      CSeq: 1
      Accept: application/sdp
      User-Agent: ItecMp4Player

S->C: RTSP/1.0 200 OK
      Server: ItecMp4Server
      CSeq: 1
      Content-Type: application/sdp
      Content-Length: 739
```

```
Date: 30 Jan 2003 11:52:24 GMT
Content-Base: rtsp://143.205.122.54/lotr.mp4/

v=0
o=ItecMp4Server 1043923944 926516 IN IP4 143.205.122.54
s=lotr.mp4
e=admin@
c=IN IP4 143.205.122.54
b=AS:1514
t=0 0
a=control:*
a=mpeg4-iod:"data:application/mpeg4-iod;base64,AoCAgG0AT
    ////wH/A4CAgCoAAgAEgICADQEFAAATAAAAmAAAAAoGgICAEAAkAAAD6
    AABX5AgAAAAABcDgICAMgADAASAgIAVAg0AABMAAACYAAAACgWAgIADA
    ABgBoCAgBAAJAAAA+gAAV+QIAAAAAAX"
a=isma-compliance:1,1.0,1
a=range:npt=0-13.23300
m=video 0 RTP/AVP 96
b=AS:1514
a=rtpmap:96 MP4V-ES/90000
a=control:trackID=1
a=cliprect:0,0,352,288
a=fmtp:96 profile-level-id=1
a=mpeg4-esid:1
```

So the client asks the server for a media object *lotr.mp4* and expects an SDP description (*application/sdp*) [103]. The server answers with a unique sequence number *CSeq* and an *SDP* block with information about the title, author, bandwidth, and time length. Since this MP4 file might contain multiple MPEG-4 elementary streams, SDP also transmits the initial object descriptor (IOD). Then all elementary streams are listed with their type, accessible network protocol and trackIDs. Here, only a video stream is included. Find more information about SDP in Section 6.3.1.

SETUP Now the client knows the details about the RTP/AVP stream and will prepare everything by using the SETUP command. The URL is extended by the trackID, to directly set up the first (and only) video elementary stream.

```
C->S: SETUP rtsp://kermit:3128/lotr.mp4/trackID=1 RTSP/1.0
      CSeq: 2
      Transport: RTP/UDP;unicast;client_port=40002-40003

S->C: RTSP/1.0 200 OK
      CSeq: 2
      Server: ItecMp4Server
      Session: 71987123
      Date: 30 Jan 2003 11:52:24 GMT
      Transport: RTP/UDP;unicast;client_port=40002-40003;
          source=143.205.122.54;server_port=20000-20001
```

The client installs the stream from the MP4 container and the server sets it up, using the session number. If there where multiple streams (eg. an additional audio stream), there had to be two SETUP commands, but both streams would be accessible with the same session ID.

PLAY After that, the full session with all active (already set up) streams is started and synchronized. The `Range:` field depicts the start and end time within a media stream, so this allows random access into the stream.

```
C->S: PLAY rtsp://kermit:3128/lotr.mp4 RTSP/1.0
      CSeq: 3
      Session: 71987123
      Range: 0.0 - 13.233000

S->C: RTSP/1.0 200 OK
      CSeq: 3
```

PAUSE The PAUSE communication scheme looks like that for PLAY. When a stream is paused, all reservations and already set up paths and streams stay active but no data packets are transmitted.

TEARDOWN The TEARDOWN communication scheme looks like the one for PLAY. But here, all resources associated with the session are freed. The session identifier is not valid anymore and to restart a stream, a new SETUP request has to be sent.

CHAPTER

7 Extensions to RTSP, RTP and RTCP

7.1 Overview

To enable stream switching within more than the temporal and SNR domain, which can be handled client-transparently, but also for the client non-transparent domains like the spatial and codec domain, extensions for RTSP were proposed in an Internet draft [106]. They are discussed in the following.

To allow the proposed two-dimensional dynamic adaptation (the combination of stream switching and in-stream temporal adaptation) within a video streaming session, the basic standard of RTP/RTCP was extended with more immediate RTCP feedback mechanisms which also allow statistics on exact frame loss. This is further used to implement RTP retransmission which severely increases the visual quality. In this chapter we evaluate the improvements and hereby show the importance of these extensions for a viable multimedia streaming environment.

7.2 Extensions for RTSP Stream Switching

To make life easier for client terminals, the server should inform the client about possible *switch sets* already at the beginning of a streaming session, so all available codec/bandwidth/domain combinations, that might become active, can be correctly set up and prepared to function. The Internet draft on stream switching [106] presents possibilities for preliminary listing and on-demand signalling of switchable streams based on extensions to RTSP and SDP. Please find an overview on RTSP and SDP

in Section 6.

Note, that these extensions only become really useful if the server *has to inform* the client about its decision on switching the stream, so whenever *non-transparent* stream switching is used. This work was evaluating only client transparent switching, so none of the here listed RTSP extensions where actually implemented. Nonetheless, this extension is an important contribution for future stream switching environments to leverage all possible switching domains, so it will be discussed in the following.

Before coming up with new and self-standing commands, namely `SWITCHSETUP` and `SWITCHSIGNAL`, this Internet draft also evaluated the (re-)use of already existing RTSP commands like `PAUSE/PLAY` or `MUTE/UNMUTE`. Unfortunately a re-use of the old commands would cause incompatibility problems or sub-optimal behavior.

The following example RTSP communication shows the usage of the new extensions. It was taken from [106], but is further simplified. A client's `DESCRIBE` request is answered by an SDP description containing all available (switchable) streams, with bandwidths ranging from 200 kbps, over 100 kbps, down to 50 kbps.

```
C->S: DESCRIBE rtsp://foo/twister RTSP/1.0
        CSeq: 1

S->C: RTSP/1.0 200 OK
        CSeq: 1
        Content-Type: application/sdp
        Content-Length: xxx

        v=0
        o=- 2890844256 2890842807 IN IP4 172.16.2.93
        s=RTSP Session
        i=An Example of RTSP Session Usage for Stream Switching

        a=control:rtsp://foo/twister
        t=0 0

        m=video 7722 RTP/AVP 96
        a=rtpmap:96 MP4V-ES/1000
```

```
a=control:rtsp://foo/twister/video1
b=AS:200

m=video 7726 RTP/AVP 98
a=rtpmap:98 MP4V-ES/1000
a=control:rtsp://foo/twister/video2
b=AS:100

m=video 7726 RTP/AVP 100
a=rtpmap:100 MP4V-ES/1000
a=control:rtsp://foo/twister/video3
b=AS:50
```

After the client has received all possible choices, it answers with a `SETUP` for the main stream, but signals the server that it will be accepting all other variants for switching. To do so, the newly introduced command `SWITCHSETUP` is used in conjunction with the SessionID, which was received from the standard `SETUP` command.

An optional header field of the `SWITCHSETUP` command is `Switch-control`, which, in this example, is set to

`Switch-control=non-transparent-forewarning: 2000`

So the server might switch on its own at any time to every stream, but when choosing this non-transparent stream, a warning has to be sent out at least 2000 milliseconds before the real switching is performed.

Further note that client and server negotiate different UDP ports for each stream. This is not obligatory, since this differentiation can also be performed using the dynamically assigned RTP Stream-Ids received in the Session Description (SDP) block within the `DESCRIBE` answer.

```
C->S: SETUP rtsp://foo/twister/video1 RTSP/1.0
      CSeq: 2
      Transport: RTP/AVP;unicast;client_port=8002-8003

S->C: RTSP/1.0 200 OK
```

```
      CSeq: 2
      Transport: RTP/AVP;unicast;client_port=8002-8003;
                 server_port=9004-9005
      Session: 12345678


C->S: SWITCHSETUP rtsp://foo/twister/video2 RTSP/1.0
      CSeq: 3
      Transport: RTP/AVP;unicast;client_port=8006-8007
      Session: 12345678
      Switch-control=non-transparent-forewarning: 2000

C->S: SWITCHSETUP rtsp://foo/twister/video3 RTSP/1.0
      CSeq: 4
      Transport: RTP/AVP;unicast;client_port=8010-8011
      Session: 12345678
      Switch-control=non-transparent-forewarning: 2000


S->C: RTSP/1.0 200 OK
      CSeq: 3
      Transport: RTP/AVP;unicast;client_port=8006-8007;
                 server_port=9008-9009
      Session: 12345678

S->C: RTSP/1.0 200 OK
      CSeq: 4
      Transport: RTP/AVP;unicast;client_port=8010-8011;
                 server_port=9012-9013
      Session: 12345678
```

After that, the client starts to play the main video stream, which can be switched at any time by the server. If it is a non-transparent switch, the `SWITCHSIGNAL` will be sent at least 2000 milliseconds before switching.

```
C->S: PLAY rtsp://foo/twister RTSP/1.0
```

```
           CSeq: 5
           Range: npt=0-
           Session: 12345678

     S->C: RTSP/1.0 200 OK
           CSeq: 5
           Session: 12345678

[...]

    S->C:  SWITCHSIGNAL rtsp://foo/twister/video1 RTSP/1.0
           CSeq: 33
           Range: npt=47.234
           RTP-Info: url=rtsp://foo/twister/video2

    C->S:  RTSP/1.0 200 OK
           CSeq: 33
```

Critical Discussion Especially the domains of switching between different codecs and spatial resolutions will offer a tremendous increase of the coverable bandwidth range. Those domains can only be adressed by client non-transparent switching, which demands some ways of communicating the server's switching decision to the client. This task can be solved by extending the widely used RTSP protocol.

RTP stream multiplexing or the setup of multiple UDP ports can be handled by explicit `SETUP` of each promoted switch stream from the SDP description block. Unfortunately, when thinking of multi-stream MPEG-4 like a background and a foreground stream, there are no means to mark the streams connected to the same scene, and various bandwidth variations thereof. The actual Internet draft only offers support for a single stream encoded in multiple versions. This description of switch sets and multi-object scenes can be tackled by the notion of *Digital Items* covered in the upcoming MPEG-21 standard [107, 108]. Although some first support for *switch sets* is already given by a new generation of SDP which allows the addition of MPEG-21 Digital Items [109], further evaluation and analysis for completeness has to be done.

Not only possible switch sets have to be sent, it is also important to communicate which basic switching capabilities the client understands in general (temporal, SNR, spatial or codec).

Still, for the simple use case with completely transparent switching, there is no need to inform the client about the possible switch set, since this information is completely useless for the client.

7.3 Extensions for RTCP-based Feedback

Standard RTCP does not offer any information about *which* data packets were lost, just the measured packet loss ratio. Further, RTCP packets are only allowed to use a maximum of five percent of the bandwidth and the feedback interval is set to a minimum of five seconds.

To enable more accurate and immediate action on network problems, an extension to RTCP for feedback was proposed by Ott et al [7]. In the best case, this allows information on loss (NACK) or receipt (ACK) of RTP packets in one network round-trip time. Important data can be retransmitted and/or the original stream can be adapted to a lower bandwidth.

This Internet draft proposes three modes of operation depending on the group size of participating hosts in an RTP setup:

1. The *immediate feedback mode* is used when the group size is small enough, so that every receiving party has enough bandwidth to immediately send all RTCP feedback packets.

   The maximum group size for the *immediate feedback mode* is determined by a number of parameters like the type of feedback used (ACK vs. NACK), bandwidth, packet rate, packet loss probability and distribution, media type and the codec.

   ACKs (positive acknowledgements of received packets) are restricted to point-to-point communications only, which means that a maximum of one client is connected to the same server session.

2. Within the *early RTCP mode*, the group size or other parameters do not allow to react on each event that would be worth (or needed) to report. Still, the receiver is allowed to send an RTCP packet before its regularly scheduled RTCP interval. But to send further early packets, it has to wait until the next interval starts. If the next interval is too distant in time, so that it renders unnecessary to still send the back-held RTCP packet, this packet has to be discarded.

3. With a very large group size, it is no longer useful to provide feedback from individual receivers at all. Here, normal rules for RTCP intervals and packaging apply.

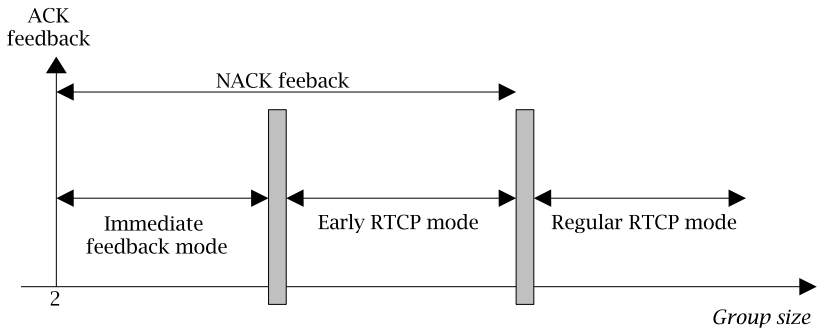


Figure 7.1: RTCP feedback modes of operation with growing group size

Figure 7.1 shows the three modes, where the transition is not fixed but dependent of multiple parameters and the application requirements.

7.3.1 Feedback Types

The Internet draft not only introduces different possible sending modes of RTCP packets, but also an additional RTCP packet extension to cover more detailed feedback on single entities like packets or video frames.

Feedback messages are classified into the following categories:

- *Transport layer feedback messages* for general purpose feedback information. These messages are based on packets and RTP sequence numbers, so they are independent from the particular codec or application.

- *Payload-specific feedback messages* are highly dependent on the used payload type, so they are codec specific. Possible feedback might be sent for *Picture Loss Indication (PLI)*, *Slice Lost Indication (SLI)* or *Reference Picture Selection Indication (RPSI).*

  RPSI is used to indicate the last valid reference frame available in the decoder. With this knowledge, the server-side encoder can code and send the following *referencing* frames based on an older reference frame, which will allow errorless decoding at the client side. This will increase frame sizes because of reference distance, but might still be more efficient than retransmitting the intermediate reference frame. Further, this missing reference frame might not arrive in time anyway.

- *Application layer feedback messages* are totally handled by the application and are not further specified in the Internet draft nor in this thesis.

7.3.2 Packet Formats

7.3.2.1 Common Packet Format

All feedback messages are based on the generic RTCP packet format and extend it as shown in Figure 7.2.

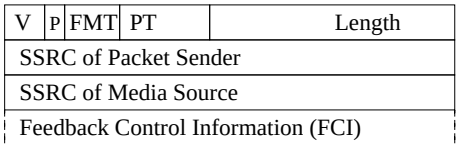


Figure 7.2: Common packet format for feedback messages

Payload Type (PT) (8 bits) The *Payload Type* is either 205 for transport layer feedback messages (RTPFB) or 206 for payload-specific feedback messages (PSFB).

Feedback Message Type (FMT) (5 bits) With respect to the *payload type*, the *FMT* is set according to Table 7.1. Application layer feedback messages are just payload-specific messages with their *FMT* set to 15.

Value	Transport Layer Feedback	Payload-Specific Feedback
1	generic NACK	Picture Loss Indication (PLI)
2	generic ACK	Slice Lost Indication (SLI)
3	unassigned	Reference Picture Selection Indication (RPSI)
15	unassigned	application layer feedback message
31	reserved	reserved
rest	unassigned	unassigned

Table 7.1: RTCP feedback message types correlated with their payload types

7.3.2.2 Transport Layer Feedback Messages

Within the feedback for *transport layer*, only *ACK* and *NACK* messages are specified.

NACK Missing RTP sequence numbers are encoded in the feedback control information block as shown in Figure 7.3. The *PID* is the *Packet ID* of the first lost packet, where the *bitmask of following lost packets (BLP)* is a bitmask of the following 16 packets. Every lost packet is depicted by a set bit (1). The least significant bit stands for $packet_{PID+1}$, the most significant bit for $packet_{PID+16}$.

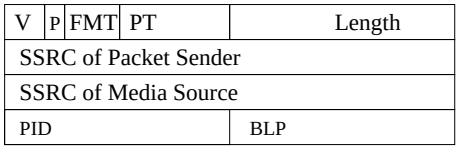


Figure 7.3: NACK packet format for lost RTP sequences

ACK RTP sequence number acknowledgements are encoded as shown in Figure 7.4. The first received packet ID is stored in the *PID* field. Then the *Range of ACKs (R)* flag switches the interpretation of the *BLP* field. If $R = 1$ then *BLP* stores the number of consecutive packets to be ACKed, not including the first packet in *PID*. If $R = 0$ then the *BLP* is used as a bitmap like in NACK messages, where a set bit (1) means that the corresponding packet is ACK'd. Here, because the R flag needs one bit, only 15 bits are available for consecutive packets $packet_{PID+1} \ldots packet_{PID+15}$.

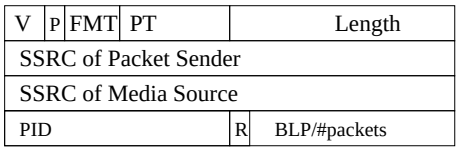


Figure 7.4: ACK packet format for received RTP sequences

7.3.3 SDP Extensions

The RTCP feedback capability is expressed by an introduced attribute to SDP, which is used within RTSP. According to our previous examples on SDP (see Section 6.3.1), the media block of the video has the feedback attribute set.

```
m=video 0 RTP/AVP 96
b=AS:1514
a=rtpmap:96 MP4V-ES/90000
a=rtcp-fb:96 nack
```

7.4 Extensions for RTP Retransmission

The previously introduced RTCP feedback informs only about lost packets, but it does not specify how a server should react on this packet loss.

The server could introduce forward error correction (FEC) and redundancy to packets so that lost packets could be (partially) regenerated [110, 111]. This scheme might be used for realtime video conferencing, where virtually no latency is tolerated, but it will increase the packet size and hereby the needed overall bandwidth even in an error free (or low-error) environment.

In the case of multimedia streaming, an initial delay of a few seconds is tolerated, so this introduces some time gap where the server could intelligently resend the lost packets. This only makes sense on very important frames like *I-Frames* and only, if it is very likely that the resent packet arrives in time.

Some work is done in adding retransmission support to UDP on the application layer [112], whereas the retransmissions are handled on an upper layer above RTP. Still, this would only introduce a new layer with latencies and would define its own protocol for packet handling.

It is better if RTP is extended to directly support retransmission requests in RTCP packets. These retransmissions are defined in another RTP extension called the *RTP retransmission payload format* [6] proposed by Leon and Varsa.

This retransmission scheme fulfills the following requirements:

- It does not break general RTP and RTCP mechanisms.
- It is suitable for unicast and small multicast groups.
- It works with mixers and translators (see Section 6.2.1).
- It works with all known payload types.
- It allows the use of multiple payload types within a session.
- Sequence number preservation is guaranteed.

If we would just resend a packet with its original RTP sequence number, we would break RTCP statistics. But every retransmitted packet has to store its old RTP sequence number, so it can easily be re-inserted into the right place in the received data stream. The optimal solution is to send the original and retransmission packets in two separate streams. Hereby the retransmitted packets are not in the same sequence number space as the normal data packets, so all original and retransmitted packets can be distinguished and RTCP statistics are working properly.

7.4.1 Options for the Multiplexing Scheme

Still, these two streams may be sent either in two different sessions i.e. *session-multiplexing* or multiplexed in the same session using different SSRCs, i.e. *SSRC-multiplexing*.

7.4.1.1 Session Multiplexing

In session multiplexing, the original and retransmission streams are sent to different network addresses and/or port numbers. This approach introduces more flexibility, especially in a multicast scenario. Here every client can decide if it wants to subscribe to the retransmission multicast group. It would also be possible for the server to

make availabe a multicast channel serving the original stream and multiple unicast retransmission channels. Every client subscribes to its own retransmission channel and is hereby not bothered by unneeded data.

Using separate sessions also simplifies the handling in the network, especially for mixers, translators and also caching proxies. Either stream is using the same SSRC value but different (dynamic) payload types.

When *session multiplexing* is used, every stream will receive separate RTCP receiver reports. This also allows to independently choose the RTCP bandwidth for every stream.

7.4.1.2 SSRC Multiplexing

In SSRC multiplexing, both the original stream and the retransmissions are sent together within one network connection. This saves network ports, which is especially favourable for streaming servers and multimedia middleware which are involved in a high number of concurrent sessions.

When these two streams are packed into one connection, they have to use different SSRC values and different (dynamic) payload types.

7.4.2 Payload Format

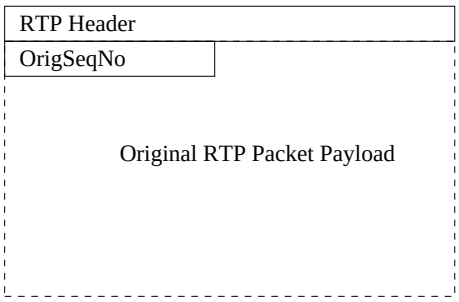


Figure 7.5: RTP retransmission payload format

The retransmission packet has to have the same timestamp as the original packet, but its own RTP sequence number. Further, the old RTP sequence number of the original packet is stored in the newly generated retransmission packet before the data is attached (see Figure 7.5). Hereby every retransmission packet can be re-integrated

into the original packet stream, but is also uniquely identifiable because of its own RTP sequence number.

7.4.3 SDP Extensions

To establish the retransmission stream for an original video stream, we have to communicate all details within the first *RTSP describe*. Therefore, the Session Description Protocol (SDP, see Section 6.3.1) is enhanced by a new attribute :

```
a=fmtp <number>: apt=<apt-value>;rtx-time=<rtx-time-val>
```

apt= is used to refer to the original stream payload type. **rtx-time=** is optionally added to inform the receiver of how long in milliseconds the server will try to resend a packet, beginning with the first sending of the original packet. If this is omitted, the maximum retransmission time is not defined or might be negotiated by other means.

7.4.3.1 SDP for Session Multiplexing

The following example shows one video with a retransmission stream. The original video is described in its media block (**m=**) with the dynamic payload type 96. RTCP Feedback is set to NACK (according to the above described SDP extensions). Then a new media block for the retransmission stream with the dynamic payload type 97 follows. Note that the RTP clock rate has to be the same as for the original video. MPEG-4 (*MP4V-ES*) uses 90000 ticks for one frame.

Finally, this retransmission stream is mapped to the original stream with:

```
a=fmtp:97 apt=96;rtx-time=3000
```

See the extended SDP example:

```
v=0
o=StreamingServer 1043923944 926516 IN IP4 143.205.122.54
c=IN IP4 143.205.122.54
m=video 0 RTP/AVP 96
a=rtpmap:96 MP4V-ES/90000
```

```
a=fmtp:96 profile-level-id=1
a=rtcp-fb:96 nack
m=video 0 RTP/AVPF 97
a=rtpmap:97 rtx/90000
a=fmtp:97 apt=96;rtx-time=3000
```

When multiple original video streams are described in one SDP, there *has* to be a flow identification (FID) grouping. This is also allowed for the single video case as shown in the following example. It establishes an FID group for both media blocks with the media IDs 1 and 2 (eg. `a=mid:1`).

```
v=0
o=StreamingServer 1043923944 926516 IN IP4 143.205.122.54
c=IN IP4 143.205.122.54
a=group:FID 1 2
m=video 0 RTP/AVP 96
a=rtpmap:96 MP4V-ES/90000
a=fmtp:96 profile-level-id=1
a=rtcp-fb:96 nack
a=mid:1
m=video 0 RTP/AVPF 97
a=rtpmap:97 rtx/90000
a=fmtp:97 apt=96;rtx-time=3000
a=mid:2
```

7.4.3.2 SDP for SSRC Multiplexing

When both streams should be sent over the same RTP connection, there is only one media block, which embodies two dynamic payload types.

`m=video 0 RTP/AVP 96 97`

All other parameters are the same as in the *session multiplexing* case.

```
v=0
o=StreamingServer 1043923944 926516 IN IP4 143.205.122.54
```

```
c=IN IP4 143.205.122.54
m=video 0 RTP/AVP 96 97
a=rtpmap:96 MP4V-ES/90000
a=fmtp:96 profile-level-id=1
a=rtcp-fb:96 nack
a=rtpmap:97 rtx/90000
a=fmtp:97 apt=96;rtx-time=3000
```

7.5 Evaluation of RTP/RTCP Extensions

7.5.1 Test Setup

The following evaluations are based on an implementation of the Internet draft for RTP extensions for RTCP-based feedback, which was added to the *UCL Common Multimedia Library* [113] of the University College London. Please find all implementation details in Appendix A.2.

The video stream used for the evaluation is the MPEG-4 reference stream "Big Show Both" with 13,000 frames and a frame rate of 25 fps in CIF resolution. The average bitrate is 400 kbps, with quantization levels of 28 for B-frames and 16 for I- and P-frames. This leads to an average PSNR value of 27.8 dB. See Figure 7.6 for the unadapted PSNR chart for the encoded stream. To gain temporal scalability, the stream was encoded with four B-frames between I- and P-frames in each *one-second GOP*, yielding a fixed frame pattern of `IBBBBPBBBBPBBBBPBBBBPBBBB`.

All of the following measurements were performed on a traffic-shaped network, which changes the available bandwidth every 30 seconds within a 20% range. Those 20% are reasonable eg. on the last mile caused by some cross traffic like background email checking. Starting with 410 kbps, bandwidth is reduced to 370 kbps after 30 seconds, then degraded to a minimum of 320 kbps, which enforces 20% quality reduction compared to the original stream; subsequently, bandwidth is increased to 370 kbps and, finally, back to 410 kbps. This bandwidth fluctuation pattern is repeated within an infinite loop, yielding an average available bandwidth of about 350 kbps.

Note that, without retransmission, arbitrary frames are lost, which may also include important I- and P-frames. This might make it impossible to decode even

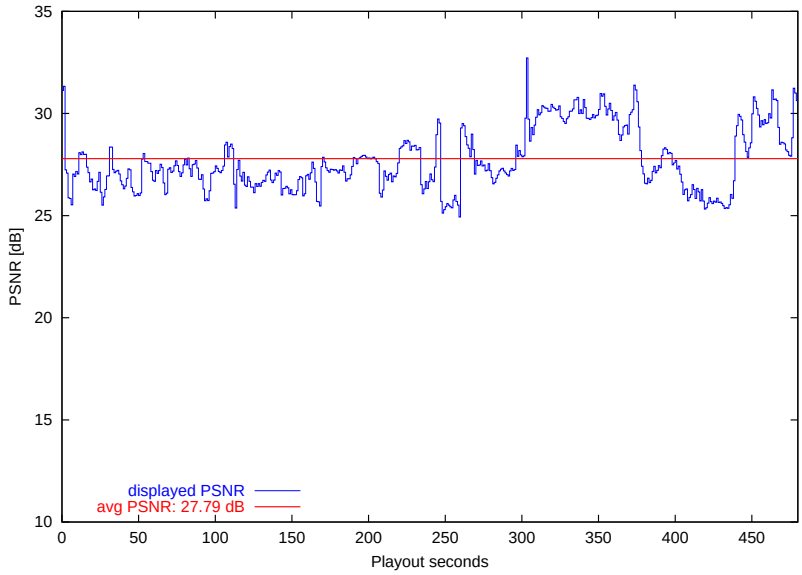


Figure 7.6: PSNR values for the unadapted video with 400 kbps

correctly received frames. This fact is displayed in the following graphs where we compare the received number of frames versus the number of decodable frames per second, which might largely diverge, if a reference frame was missing. Thus, our streaming system can retransmit I- and P-frames to allow for optimum decoding results at the receiver.

Adaptation and quality reduction is shown by displaying the PSNR loss in dB with respect to the unadapted stream shown in Figure 7.6. So the less adaptation is done to the video, the more we converge to the zero-difference line.

7.5.2 Basic RTCP Feedback

If the sender has to react to changing network conditions, *feedback* from the receiver is indispensable. The standard RTCP receiver report (see Section 6.2.3) includes the percentage of lost packets since the last RTCP receiver report and the number of totally lost and sent packets since the beginning of the session.

Those reports are sent minimally every 5 seconds and are not allowed to exceed 5% of the overall corresponding RTP session traffic. With increasing network utilization and the amount of participating senders/receivers in an RTP session, the reporting

interval will increase or RTCP receiver reports may even be turned off. Still, applications can use this rarely sent information to adjust their streaming bandwidth.

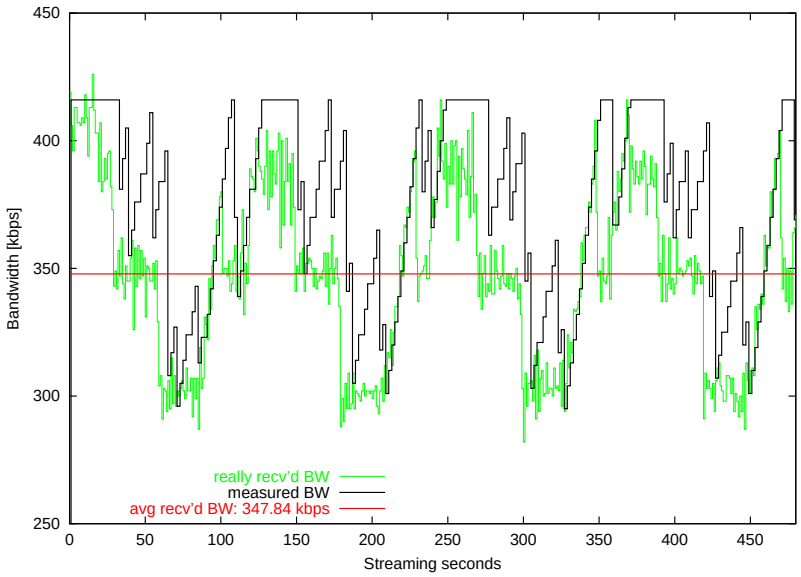


Figure 7.7: Bandwidth measurements with standard RTCP feedback

Fig. 7.7 nicely shows the steps of the traffic shaper and the late reaction on bandwidth changes due to long intervals between RTCP reports. Furthermore, all estimations on available bandwidth are too high, further aggravating packet loss. This fact and the unavailability of retransmission results in adaptation rates of up to 30%. Fig. 7.8 shows the received frames per second, but since some referencing frames like I- and P-frames are also missing, the really displayable number of frames per second decreases substatially. The average displayed frame rate is 15.7 fps. According to Fig. 7.9, we lose up to 16 dB PSNR quality because of lost I- and P-frames. The average quality reduction is about 5.6 dB.

7.5.3 RTCP-based Feedback Extension

For this evaluation, we only use the *immediate feedback mode* in a unicast scenario with one client and one server. The feedback type employed is the simple *transport layer feedback*.

Fig. 7.10 again shows the steps of the traffic shaper and the better reaction on

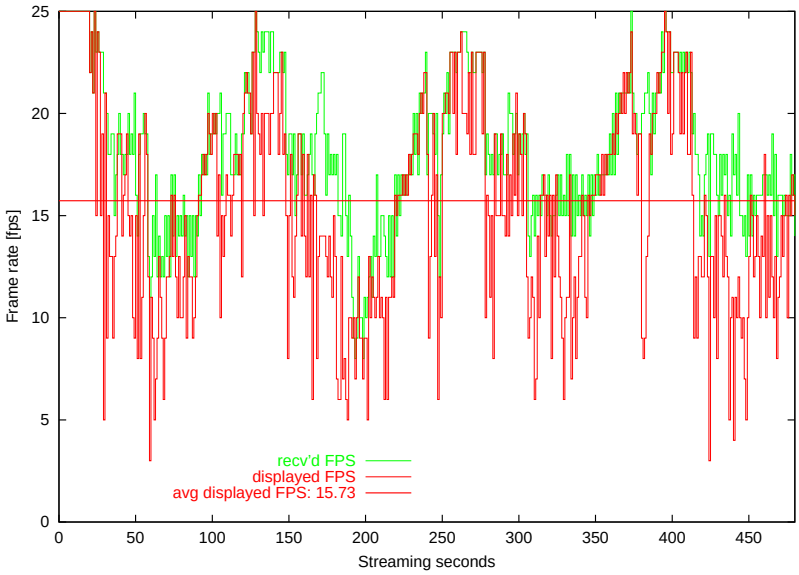


Figure 7.8: Frame rate adjustments with standard RTCP feedback

bandwidth changes because of the shortened RTCP reporting intervals. Also, the bandwidth fluctuations are encountered better. Still, the unavailability of retransmission leads to an average frame rate of 16.6 fps (see Fig. 7.11). According to Fig. 7.12, we lose up to 14 dB PSNR quality because of lost I- and P-frames. The average quality reduction is about 3.6 dB. Hence, the extended RTCP feedback allows a quality increase of 2 dB under the same network conditions.

7.5.4 RTP Retransmission

With retransmission turned on, the measured bandwidth steps are identical to those measured with immediate feedback in Fig. 7.10, but since we use retransmission on all packets (as long as they arrive in time), we obtain a higher frame rate of 19.1 fps (see Fig. 7.13). According to Fig. 7.14, we only lose up to 5 dB PSNR quality since we retransmit all lost I- and P-frames. The average quality reduction is less than 1.2 dB. Eventually, under the same network conditions, we achieve an average quality increase of 2.4 dB just by retransmission and an average quality increase of 4.4 dB as compared to the standard RTCP and RTP, without extensions.

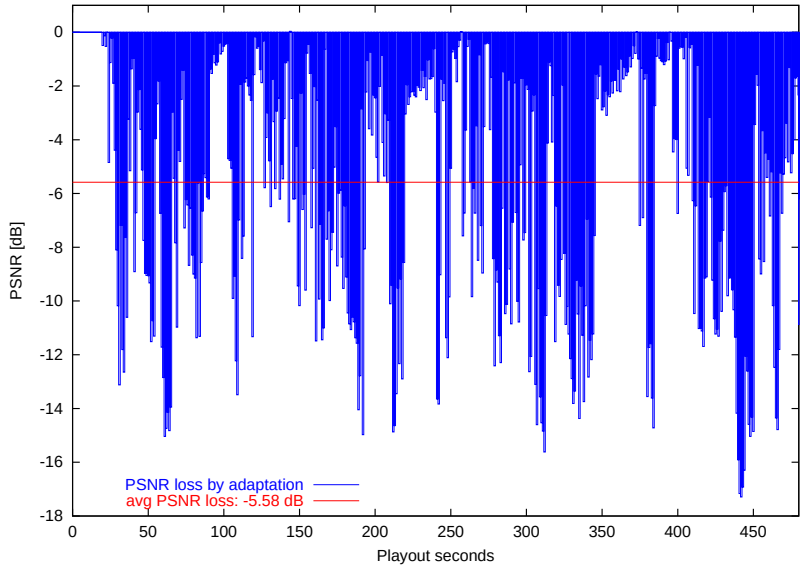


Figure 7.9: Quality loss with standard RTCP feedback

7.6 Conclusion and Future Work

The two proposed IETF drafts on extended RTP feedback and retransmission have proven to offer substantial benefit for unicast streaming environments over best effort networks employing IP. Further research and evaluation has to be done in the area of multicast scenarios and with multiple concurrent streams.

Mainly, we have to evaluate the efficacy of feedback and retransmission, as compared to other quality-ensuring measures like forward error correction (FEC) or adding redundancy to packets, so that lost packets can be (partially) regenerated [110, 111]. Still, advantages of packet retransmission will always be the low complexity on the receiver side and the bandwidth efficiency in a nearly error-free network.

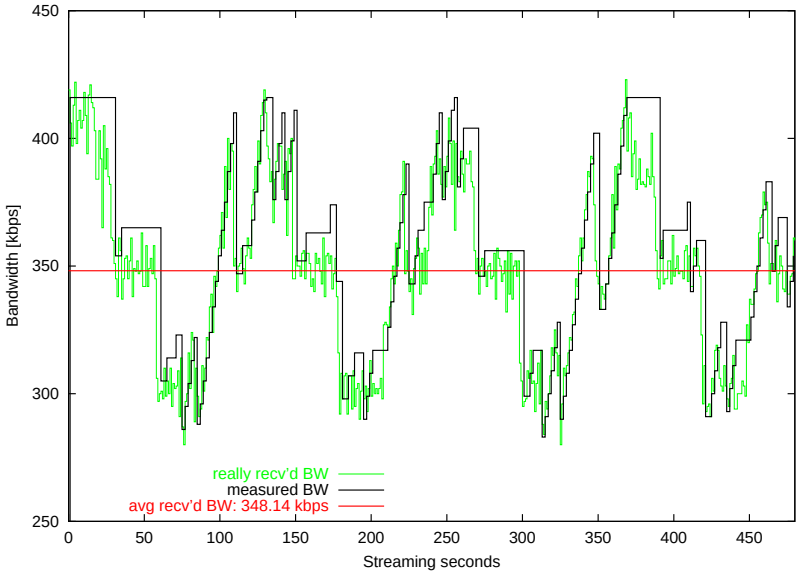


Figure 7.10: Bandwidth measurements with extended RTCP feedback

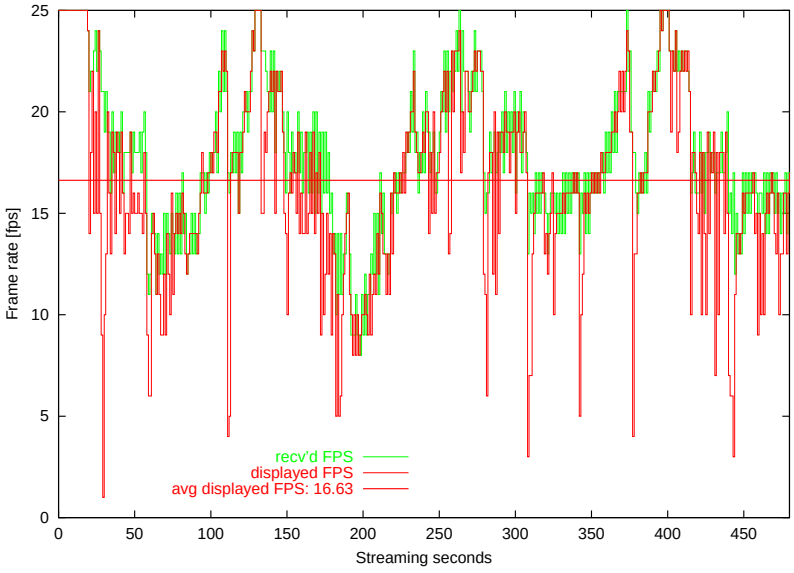


Figure 7.11: Frame rate adjustments with extended RTCP feedback

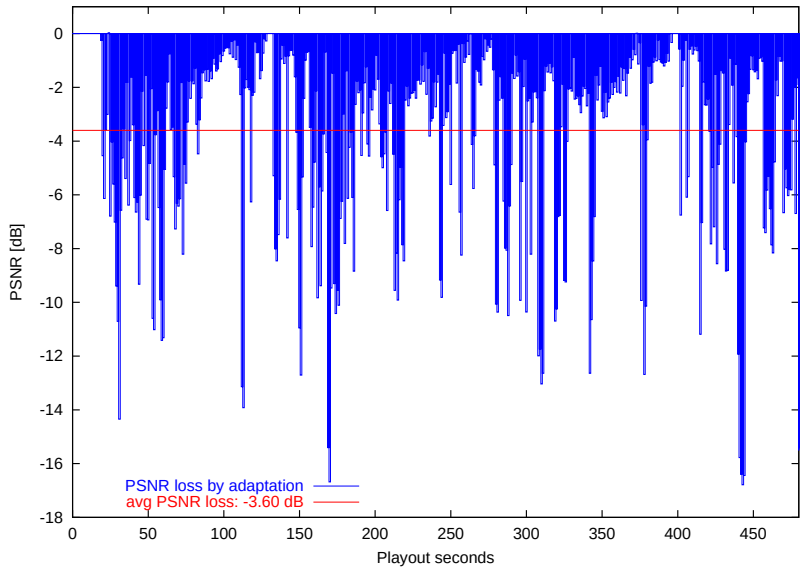


Figure 7.12: Quality loss with extended RTCP feedback

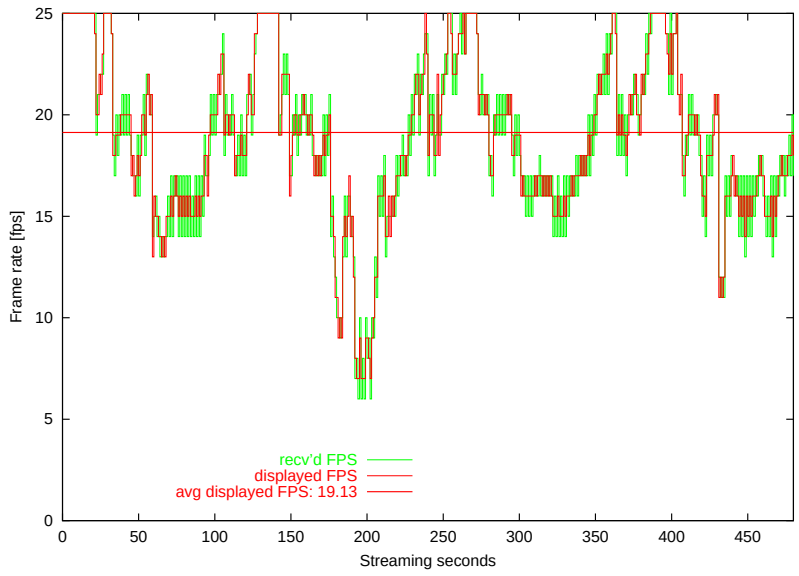


Figure 7.13: Frame rate adjustments with retransmissions

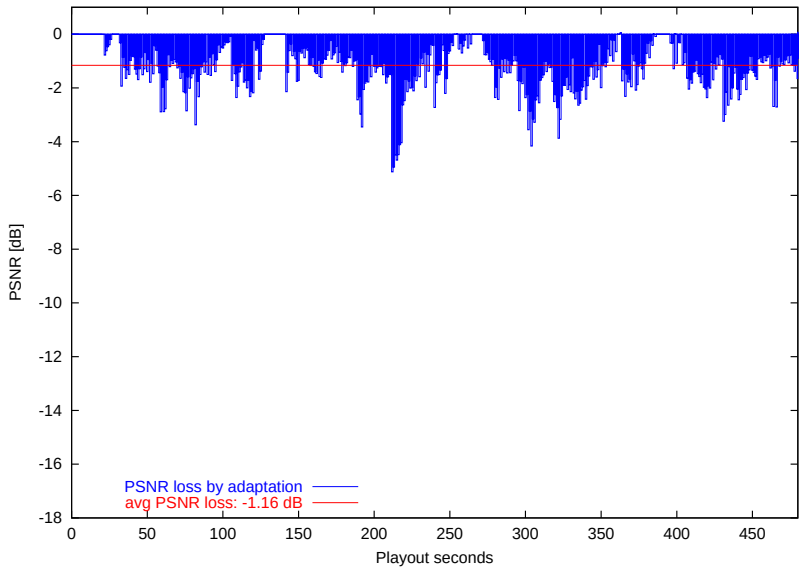


Figure 7.14: Quality loss after decoded retransmissions

CHAPTER 8

ViTooKi - The Video ToolKit

ViTooKi – the **Vi**deo-**ToolKi**t [56] is an open source client/server framework which was developed at our department. It is capable of sending, adapting and displaying MPEG-1/2/4 video streams, streaming and playing MP3 audio streams, and opens container formats like `.mp4` or `.avi`. It comes with full support for adaptive standard compliant multimedia streaming and proxy caching. It consists of a multi-platform core library and various applications using this ViTooKi core library. The core library is using FFmpeg [65] and/or XviD [66] , therefore it is very open for extensions for new video and audio formats and container formats.

Other toolkit projects on video streaming are MPEG4IP [114], Helix [115] and VideoLAN [116], none of them offers the needed capabilities nor the necessary combinations of dynamic adaptation and stream switching and retransmission. They also ignore the state of the client-side buffer. Such a buffer-aware architecture was published in [117]. It adds buffers for the displaying engine to compensate decoding jitter and implements frame dropping, but does not support stream switching or packet retransmission. ViTooKi implements all of these extensions and features, which were also discussed in detail prior in this thesis.

One of the use cases for ViTooKi, and the topic of Peter Schojer's Ph.D. thesis [118], is a caching proxy which, instead of deleting outdated content to reduce the needed cache size, first tries to reduce the media size by quality reduction [64]. Other available applications are an adaptive streaming server, an MPEG-21 compliant multivideo player with configurable terminal capabilities, a multivideo transcoder, a DVD ripper and an MPEG-7 video annotation tool. Many active student projects are

constantly increasing the amount of available tools, all leveraging the simple-to-use ViTooKi library. Appendix B lists some convenience functions which can be leveraged for statistical analysis, frame prioritizations and quality measurements.

This chapter will explain the ViTooKi design in detail, including the integration of all the previously introduced technologies like smooth buffer streaming, retransmissions, in-stream adaptation and stream switching. Adaptation methods are implemented as exchangeable and chainable Adaptators as described in Section 3.3.2. In ViTooKi, those adaptor chains are used eg. on the server/proxy side to transcode videos in real-time to fit the terminal properties specified by a client (eg. the display resolution) or to adjust to changing network bandwidths. Transcoding implementations exist for the temporal (B- and P-frame dropping), spatial (resolution), quality domain (quantization) and color reduction.

8.1 Generic Streaming Environment

Figure 8.1 shows the basic idea behind a client-server streaming architecture with all directly involved classes from the ViTooKi library. First, we will describe the server side. There is an *IO Input* Class which reads frame by frame from an MPEG-4 elementary stream by using the `getFrame()` method. Those frames are passed to the *DataChannel*, which is connected to an arbitrary number of attached *DataSinks*, which represent the open client connections.

The DataChannel passes each frame to its global adaptors, if there are any. Those adaptors might do grayscaling or could generate various statistics on by-passing frames (see Figure 8.2 for all actually available adaptors within their class hierarchies).

A somehow adapted (but maybe also unchanged) frame is then passed to each DataSinks' local adaptors. This can be used to post-processthe frames especially for this connected client. After this "private" adaptation, the frame is passed to the *IO Output* class via the `IO::writeFrame()` method. The output can be directed to a file storage output, but in the streaming case, a network output class sends data via the network to reach a remote client.

On the client side, everything is more or less the same but inversed. The frame

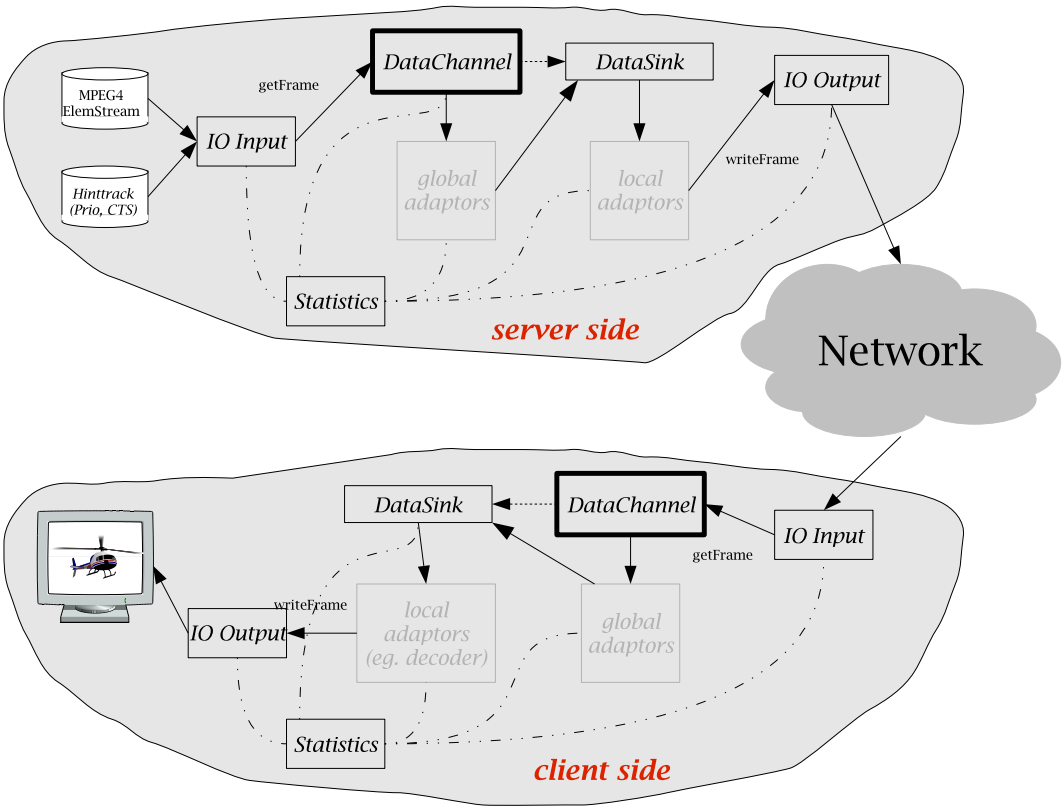


Figure 8.1: Class overview for client-server architecture in ViTooKi

is received by the *IO Input* and then passed to the DataChannel via the `getFrame` method. After applying some global adaptors, the frame is handed over to the local adaptors. For a video player, this might first be a synchronization adaptor, delaying further processing according to the frame's presentation timestamp, and after that, a decoding adaptor, to transform the frame from its compressed domain into raw YUV.

This frame is processed by the adaptors and then passed to the *IO Output* class via the `IO::writeFrame()` method. *IO Output* finally displays the frame on the screen.

8.1.1 Server-Side RTP Class

The following subsections describe a specialized *IO* class which integrates RTP support according to RFC 1889 [94] into ViTooKi for both sending and receiving. Low-level RTP communication is based on the UCL Common Multimedia Library [113],

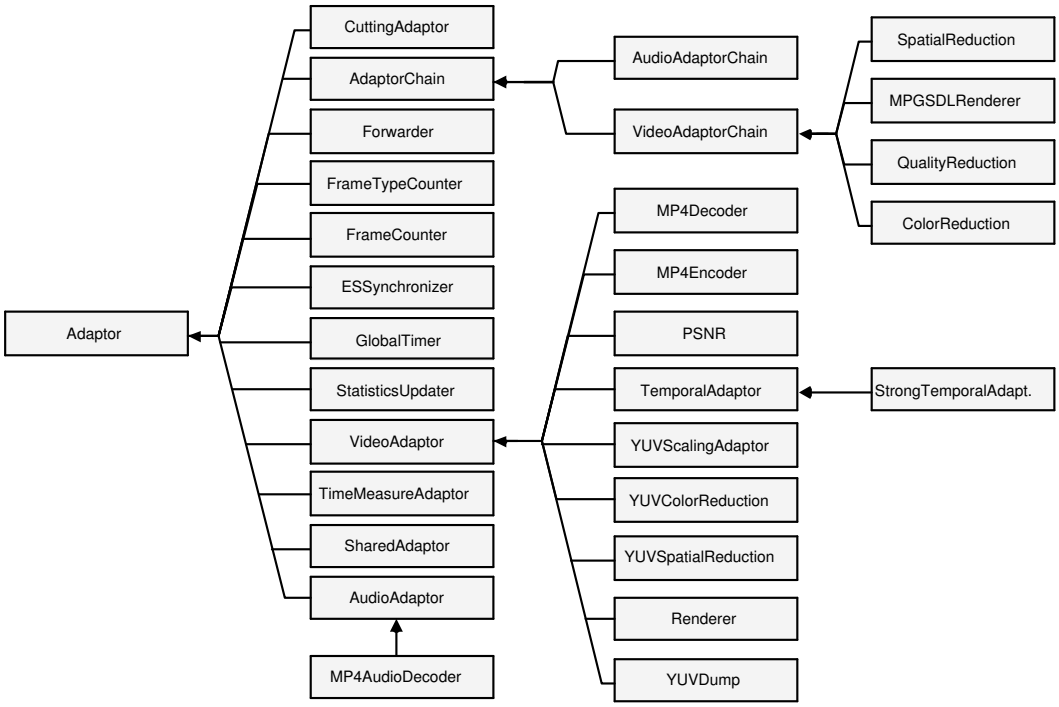


Figure 8.2: ViTooKi adaptors within their class hierarchies

which was extended by extra features like NACK/ACK feedback [7] and retransmission [6]. Those extensions are described in Section 7.3 and 7.4 and their performance was analyzed in Section 7.5.

As described in the general overview of ViTooKi, every stream is handled by a *DataChannel* which runs as a seperate posix thread. On the server-side, after reading an encoded MPEG-4 frame from a file, this frame is passed to a *DataSink* with its attached Output, which is our `Rtp` class. `Rtp::writeFrame()` is then called with the encoded MPEG-4 frame. Figure 8.3 shows that this frame is packetized in the *Packetization Engine* and is, accoring to RFC 3016 [96], fragmented into seperate access units of *Maximum Transfer Unit (MTU)* size of the underlying network (eg. ethernet uses 1500 bytes). Those network packets do not only store the payload, but also such information like the frame type, presentation timestamp and frame priority for dropping behavior (see Chapter 4). They are inserted in a list, called *preQ*, sorted by their timestamps.

With the initialization of the `Rtp` class, a seperate *sendThread* is started. This

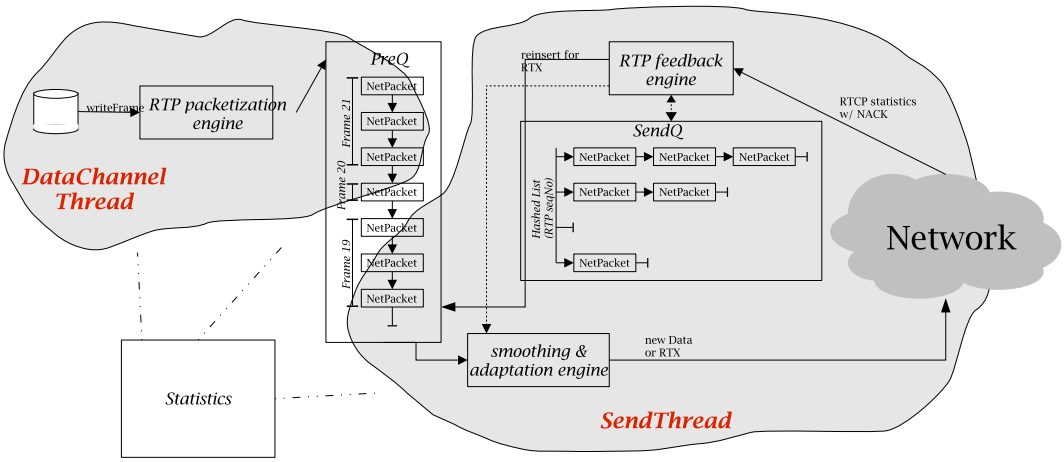


Figure 8.3: Concurrent threads at the server-side of the Rtp class

thread always tries to pop packets out of the *preQ*. According to the packets' priority and the measured available network bandwidth (for more details see section 8.2.1), this popped packet is dropped by the *smoothing & adaptation engine* or sent out on the network. If a packet is sent out, it will be stored into the *sendQ*, which is used for possible retransmissions and statistics. Before the *sendThread* tries to pop the next packet, it sleeps a calculated amount of time. Obeying these time slots, only a certain amount of packets is sent out in one streamout second ($Ssec$). This amount will then roughly match the reqested streamout rate for this second. The exact streamout rate will not be met all times because of packet size boundaries, so the used bandwidth is always varying within one MTU (maximum transfer unit). Further, since TCP-friendly behaviour[1] is envisioned, the available bandwidth is always re-measured and the streamout bandwidth is adjusted in a AIMD fashion (additive increase, multiplicative decrease). This leads even more to a spiky streamout bandwidth and shortly delayed reaction.

The *sendThread* also checks the network for incoming RTP feedback messages. If retransmission is requested for lost packets, they are picked out of the *sendQ* and are reinserted into the *preQ*, again sorted by their timestamps, which are always lower than any other packet already stored in the *preQ*. By using the timestamps,

[1]TCP-friendly means to be cooperative to other long-term high bandwidth connections like eg. `ftp`

retransmission packets are sorted among each other as well. This prioritization allows the immediate and in-time send-out of urgently needed data. Since we assume only a low need of retransmissions, we always resend requested packets if they could arrive in-time, more or less ignoring the increasing bandwidth hereof. They are not adapted within the *smoothing & adaptation engine*, because we do not want to break dependencies of partially received full frames, where eg. only one single packet is missing. This works well, since the packet loss is detected anyway and leads to future adaptation and hereby reaches network stabilization very soon.

8.1.2 Client-Side RTP Class

At the client side, there is a *recvThread* which checks for incoming RTP packets, which might be either new data or retransmitted packets (see Figure 8.4).

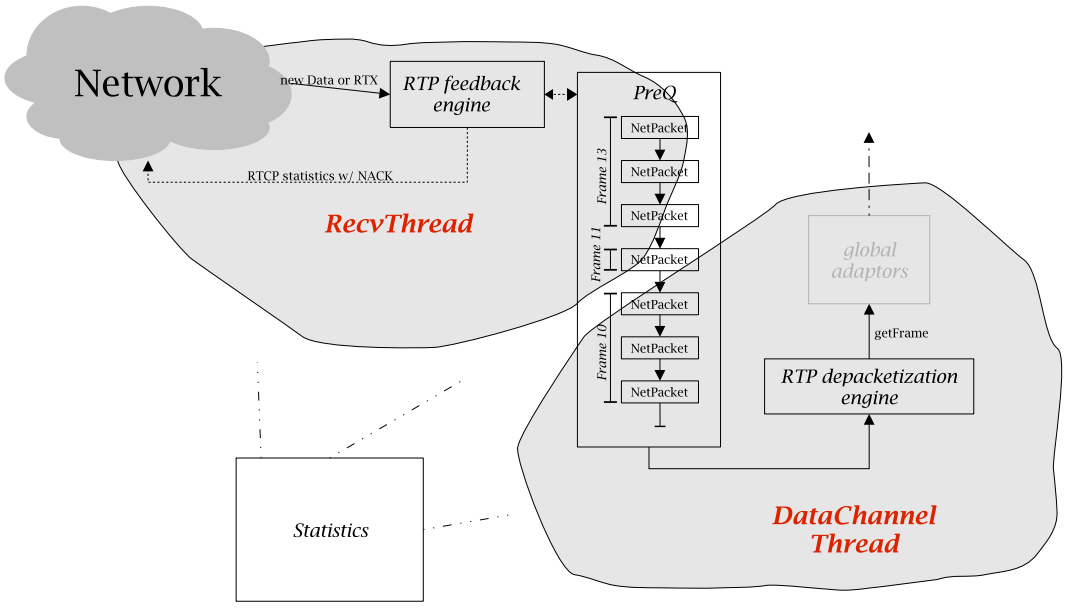


Figure 8.4: Concurrent threads at the client side of the Rtp class

Every received packet is inserted into the *preQ*, according to its timestamp. The *recvThread* is keeping track of the last incoming RTP sequence number $seqNo_n$, so if a new packet arrives and it's RTP sequence number $seqNo_m, m > n+1$ is not the direct successor ($seqNo_{n+1}$), an *immediate feedback RTCP* packet is generated and the server is informed of the missing packet(s) $seqNo_{n+1} \ldots seqNo_{m-1}$. This silently

ignores the fact that UDP does not guarantee in-order arrival of packets and can be solved by an introduced timer event, which waits a certain time before sending the RTCP feedback. Maybe the missing packet will arrive during this configurable period of time. Still, according to [112], receiving out of order messages is very rare in practice, so receiving a message with a too high sequence number is a good sign that the expected message was unrecoverably lost.

At the same time when the *recvThread* is started, the concurrent *DataChannel* thread tries to extract the very first frame with the `Rtp::getFrame()` method. Obviously, in the beginning, no packets have arrived yet, so the *DataChannel* thread goes into *prebuffering state* and waits for a definable amount of time (normally 8 seconds). After that, it assumes, that the *recvThread* has stored a significant number of packets in the *preQ*. Then the *DataChannel* thread continues and starts to pop packets from the *preQ*. Whenever the *DataChannel* thread runs out of data again, a new *prebuffering* period is introduced, but this also means a complete blocking of the client side player and is hereby a highly unwanted scenario. To avoid this buffer underflow, server-side adaptation and switching is applied.

Since `getFrame()` has to return a full and complete (decodable) frame, we have to combine the network packets into one single frame again. This is done by reading the packets' timestamps and their final marker bit. If any of the consecutive packets is missing, the full frame is discarded and the `getFrame()` method is called again to extract the next full frame. If an error occurs at this point, it is definitely too late for retransmission, since the current frame has to be decoded and displayed next. Still, the code could be extended by some kind of forward error correction like Priority Encoding Transmission (PET) [110, 111] or even incomplete frames could be passed to the decoder in favor of independently decodable video packets based on MPEG-4 video slice bounds [29] (though this feature is neither implemented in the FFmpeg nor XviD MPEG-4 codec implementation).

8.2 Bandwidth Smoothing and Adaptation

8.2.1 Bandwidth Consumption

To avoid client buffer overrun or underrun, the server side RTP class tries to send out packets for each streamout second $Ssec_n$ with a certain streamout bandwidth $streamBW$, which was calculated according to the proposed algorithm in Section 2.4.

Since this $streamBW$ is not guaranteed on best-effort networks, we first have to measure the really available $real\_netBW$. By receiving NACKs from the client side and looking up the according network packets from the *sendQ*, we can subtract the not received packet sizes from the sent packet sizes. Further, we have to make sure, that we only subtract NACKs from the according streamout second $Ssec_n$. So SSP_n denotes the set of all packets p_i in a certain $Ssec_n$ and $SSLP_n$ the set of all lost packets q_j in $Ssec_n$, where obviously $SSLP_n \subseteq SSP_n$.

$$real\_netBW_{Ssec_n} = \sum_{i=1}^{numFrm(SSP_n)} size(p_i) - \sum_{j=1}^{numFrm(SSLP_n)} size(q_j)$$

Assuming that no NACK packets are lost themselves on their way from the client to the server[2], after adding some network delay, we have a very exact knowledge of the $real\_netBW$ for $Ssec_n$. Since we have to rely on stable measurements, we wait a full second until we really take the values for further calculations.

With this knowledge of this already outdated $real\_netBW$, we want to set the new streamout rate. Generally, we could fully take the resulting (but outdated) $real\_netBW$ as the new $netBW$ used as the new streamout rate for $Ssec_{n+1}$.

$$netBW_{Ssec_{n+1}} = real\_netBW_{Ssec_{n-1}}$$

This approach would have two big disadvantages:

- Since we always only measure sent packet sizes minus data loss, we cannot detect an *increase* of available bandwidth.

[2]Those NACK packets are sent from the client to the server, so it is the other direction, which is very likely uncongested.

- Other high bandwidth streams could starve, since we would always try to fill up the available bandwidth as much as possible. This problem arises especially in the Internet, where TCP traffic would be virtually eliminated by greedy and ruthless UDP bursts. To avoid this, we need to be *TCP friendly [119].*

Many approaches compete to be the most TCP-friendly congestion control [120, 121, 122, 123, 124, 125]. Algorithmic complexity and environmental restrictions of the above cited TCP-friendly algorithms forced us to refrain from their usage. Find a more detailed comparison and critical analysis of different TCP friendly congestion control approaches in [126] and [127]. With the following approach, a general TCP friendliness could be achieved. Although TCP uses additive increase and multiplicative decrease (AIMD) on bandwidth adjustment, we implemented a three-level adjustment scheme which was proposed coarsly similar in [128] and [129].

The ViTooKi network implementation uses packet loss thresholds to adjust the future network bandwidth estimation. In contrast to wireless networks, on wired networks, virtually no packet loss is caused by errors on the cable. So packet loss is always a sign for congestion and we have to react by reducing our streamout bandwidth. When modern Internet routers are available on the packets' way to their destination, those routers can even drop some packets in adavance as a forewarning, whenever the fill level of their internal packet queues are getting full. This is called *random early detection (RED)* [130]. So when routers are dropping arbitrary packets in advance, the ViTooKi server is able to adjust the $netBW$ (and the data rate by adaptation) even before the network really heavily congests and hereby also avoids unnecessary retransmissions.

If the packet loss ratio is above a critical value, we decrease the $netBW$ for $Ssec_{n+1}$ by a substatial, but not too large percentage of only 20% (TCP would drop down by 50%). If packet loss is below a certain value (eg. 1%), we increase our $netBW$, which allows us to better use available bandwidth. If packet loss is steady within acceptable bounds, we keep our bandwidth steady too. Following this strategy over a longer period of time, we converge to the well-known TCP sawtooth graph for bandwidth, but with smaller drop-downs. On the other hand we are less greedy and introduce plateaus and steps. This allows us to maintain a higher and more stable average bandwidth (see Figure 8.5).

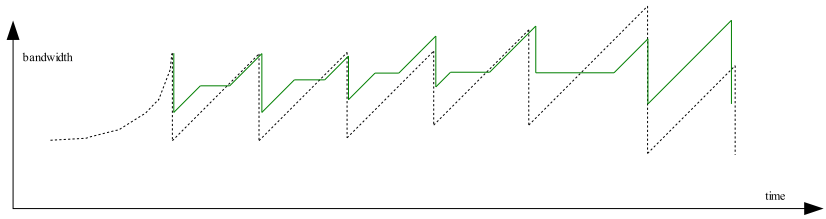


Figure 8.5: TCP sawtooth vs. our TCP friendly approach

8.2.2 Adaptation by B-frame Dropping

Calculating and adjusting the *netBW* to fit the real network bandwidth is an important issue. Still, if we cannot stream with the precalculated and needed *streamBW*, we will run out of data on the client side sooner or later. Therefore we have to adapt the data meant for the next streamout second to exactly match the *netBW*. This allows the server to send out enough packets worth for one streamout second, but with less "quality". All further streamout seconds are totally independent from this adaptation, and the basis of the pre-calculated *streamBW* will allow that the buffer bounds on the client will not be endangered by any means.

As discussed in Section 3.2.1, the cheapest and fastest adaptation is B-frame dropping. This could simply mean not to stream out 30 frames per second with the pattern `IBBPBBPBBPBBPBBPBBPBBPBBPBBPBB`, but to drop 30% of the B-frames, by any of the previously discussed frame prioritization algorithms, where unique priority values are assigned to the frames.

8.2.2.1 Priority-based Dropping Using a Hint File

Since there are better methods for gaining good priority values than simple timely uniform distribution, these (mostly off-line calculated) priority values can be stored in special hint-files. When using good hint-files, the system knows even better which frames can be dropped without losing too much quality.

Eg. those hint-files can be generated with a quality aware algorithm (QCTVA) (see Section 4.5.5), which works in the uncompressed domain on the basis of PSNR values. This file could also be written by an MPEG-7 based tool to annotate metadata information. Whenever streamout starts, the frames are dropped or kept according to their priority values by using the ViTooKi *smoothing & adaptation engine*.

Timely uniform distribution of dropped frames is relatively cheap in comparison to other "intelligent" off-line built prioritization algorithms and it avoids choppiness. The following example pattern nicely shows the timely uniform dropping behaviour: `IB-PB-PB-PB-PB-PB-PB-PB-PB-`. This is the default prioritization algorithm within ViTooKi, if no pre-calculated priority hint-file is available. Section 4.5.2 shows, how this timely distributed dropping behaviour is simply mapped to unique priority values. The following approach always uses prioritized frames, no matter where the priorities stem from.

Since the server-side RTP class knows about all packets' priorities in the *preQ*, it is easy to extract those packets that should be sent out in the next streamout second. Further, the unimportant packets are dropped.

8.2.2.2 The Smoothing and Adaptation Engine

For adaptation within ViTooKi, the problem of different frame sizes and varying bitsizes of video seconds ($VsecBS$) has to be tackled, since those variances only average out to $streamBW$ for the full video. According to Section 2.2.3, if the pre-calculated $streamBW$ is streamed out regularily every second, sending as many frames as possible, the client will be able to display the full stream (assuming a large enough prefetch time). One streamout second ($Ssec$) might contain parts of various $Vsecs$, depending on each $Vsec's$ bitsize $VsecBS$. For adaptation, the system has to adapt each streamout second ($Ssec$) to fit into the given $netBW$, which will finally reduce the overall average bandwidth.

According to the example in Figure 8.6, we assume a $streamBW$ of 137 kbps but the measured network bandwidth is only 110 kbps. So to fit the available $netBW$, the data has to be reduced by 20%. The frames are sorted by priorities and, in a loop, the highest priority numbers are chopped off until the $netBW$ can be sufficed. The *smoothing & adaptation engine* does not really drop frames immediately, but only marks them for dropping. When a frame should be sent out, a special *dropMe* flag stored with this frame is checked for the final decision.

Please note, that the prior discussion and Figure 8.6 always assumed priority values and adaptation of full frames. Because of the underlying network, the *preQ* only stores the already demultiplexed packets at MTU size, so all full frames are split

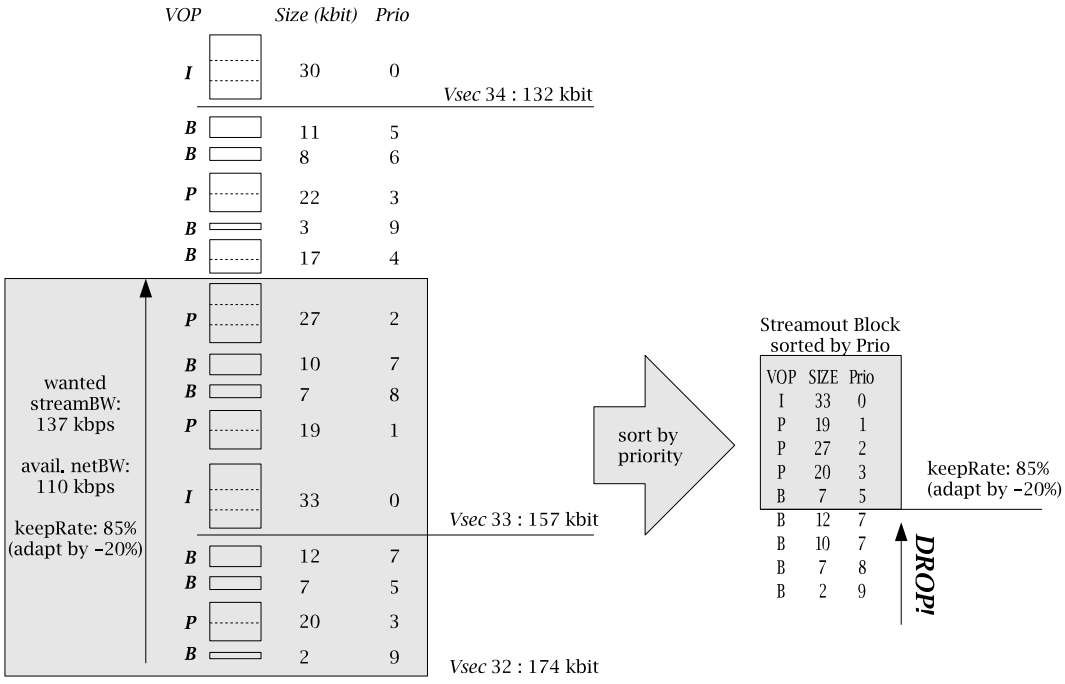


Figure 8.6: Priority-based B-frame adaptation

up into pieces. Since adaptation has to happen on a frame-by-frame basis, multiple packets which belong together have to be all marked for dropping. This gets even more complicated by the fact, that some frames might be already partially sent, so the remaining packets have to be excluded from adaptation to minimize wasted bandwidth. Setting up those rules, the remaining packets of already partially dropped frames have to be forcibly marked. This assures, that only full frames are either sent out or dropped.

8.2.3 Retransmission of Lost Packets

For retransmission, the NACK messages introduced in Section 7.3 are used by the client-side to send retransmission requests and the extension on RTP retransmission discussed in Section 7.4 is used by the server-side to resend important packets.

Packet loss on the client-side is detected by missing RTP sequence numbers, so a NACK packet is sent out. The server inserts the retransmission packet depending

on its timestamp, which determines if it will still arrive on time. If any retransmitted packet gets lost, the client has to re-request this packet again. The loss of a retransmission packet can be identified either by a missing RTP sequence number in the retransmission stream, or – if already the according NACK request was lost – by a timer on the retransmission request. Re-requests of lost retransmission packets are done on the original stream's feedback channel. If there would be NACKs and re-retransmissions on the retransmission channel itself, the doubly sent packets would loose their original RTP sequence number themselves and could not be integrated into the original stream properly. Further they would break the RTCP network statistics since the accumulated bitrate would be an inseparable combination of normal data and retransmission packets.

Still, it is important for "fair use" bandwidth sharing or bandwidth reservations, to always see the cumulated bandwidth of the original bitrate and the bitrate needed for retransmissions. So ViTooKi has to adapt the original stream even more, in order to make room for the necessary retransmissions (retransmissions are always sent out with high priority). It is assumed, that after adapting to the newly calculated bandwidth, the network stabilizes again and the number of retransmissions is hereby also decreasing. When the newly calculated network bandwidth fits the real bandwidth again, no packet loss will occur.

8.2.4 Adaptation vs. Buffer Management

8.2.4.1 Client-Side Buffer

The server side is capable of estimating the buffer fill level of the client side, since it knows which packets (respectively their timestamps) were already sent out. There is still some uncertainty if a packet has already reached the client and was stored in the client's *preQ*. To solve that, the system needs quite adequate estimates on the network delay, which can be calculated on the basis of NACK packets for already sent packets. Exactly speaking, the packet's sendout time and the incoming NACK packet only allows calculation of the full round-trip time (RTT), but this is taken as a conservative approach. The general formula $delay = RTT/2$ does not hold anyway, if the network is congested only in one direction. But exactly this situation is very

likely to happen for streaming, since the upstream channel is only used for RTCP packets whereas the high data load is congesting the downstream channel. Because of this, some unexactness is accepted and the full RTT is taken into account.

$$Buf\_ahead_n = (last\_sent\_timestamp - RTT) - Ssec_n$$

Buf_ahead_n reflects the amount of seconds of still stored data in the client buffer, because the server's $Ssec_n$ is taken as the equivalent of the client's actual playout time $Psec_n$. $last\_sent\_timestamp$ in the above formula is considered as a floating point number describing the presentation time in seconds of the currently sent out frame.

8.2.4.2 Streaming Strategies

For the following, the pre-calculated $streamBW$ is used as the wanted streamout rate, whereas $netBW$ depicts the really available network bandwidth, which is normally varying within a certain range around the $streamBW$. Taking into account not only the ability to adapt the data to meet a certain $netBW$, but also the previously discussed client buffer fill level, there are various choices with the available network bandwidth:

Available *netBW* exceeds *streamBW* If the available $netBW$ exceeds $streamBW$, instead of only sending out $streamBW$ and leave some bandwidth unused, ViTooKi sends more than $streamBW$, which will fill up the client buffer faster. Nevertheless, if the available network bandwidth is extremely high, our TCP-friendly system will not use all of it, just a little more than $streamBW$ (eg. +15%). This scenario will happen on a local network, where eg. a 390 kbps video is watched by the user. An intelligent system (like ours) will not "stream" with the maximum of eg. 1000 kbps, since this would degrade streaming to normal file transfer and obviously would require enormous client buffer capacities.

Still, under perfect network conditions, when the $streamBW$ is always exceeded even by a little, this will lead to high and overfull buffers, too. Keeping the buffer fill level always very high does not break any of the previously defined and calculated $streamBW$ requirements and obviously still guarantees seamless displaying of the

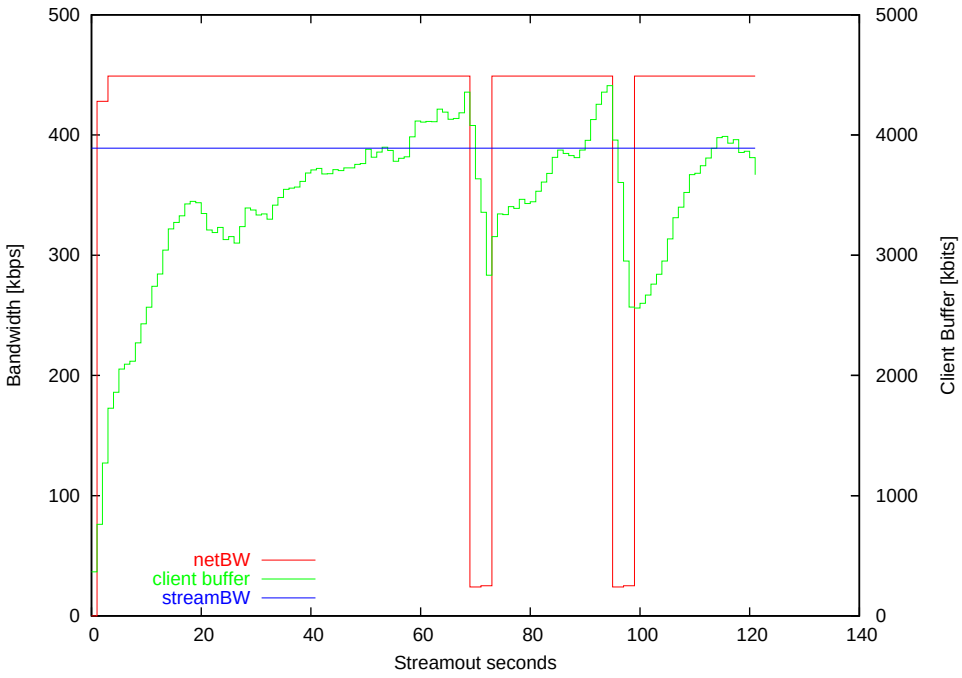


Figure 8.7: Overfull buffers force $netBW$ reduction

video. Only if the buffer fill level reaches a high-water level, the streamout rate has to be massively reduced to not completely over-fill the buffers [3].

Figure 8.7 shows a streamout scenario with a $streamBW$ of 390 kbps. Since the network is never congested in this scenario, $netBW$ is increased to a steady level of 450 kbps. Buffers are filled very fast and at $Ssec_{67}$ they reach the high-water level of 90%. Therefore $netBW$ is reduced drastically to drain buffers again until they fall below 70% filling level. After that, $netBW$ is reset to the old value of 450 kbps. The same procedure is triggered again at $Ssec_{95}$.

Even this heavy draining phase will not break the $streamBW$ requirements, as long as the buffers are maximally drained by the same amount of the previously extra-streamed data. So draining is safe as long the following equation holds:

[3]Please note, that this strategy provides more stability over possibly future network fluctuations, but an almost full buffer also introduces long delays until the buffer is played out, whenever the system performs stream switching.

$$\sum_{k=1}^{n} streamBW \leq \sum_{k=1}^{m} (streamBW + Einc_k) + \sum_{k=m+1}^{n} (streamBW - Edec_k), \forall m < n$$

In the time from $Ssec_1 \ldots Ssec_n$, instead of always streaming out $streamBW$, if $netBW$ allows it, the system first increases each $streamBW$ by some amount $Einc_k$ in the time from $Ssec_1 \ldots Ssec_m$, then the streaming rate can be dropped to $streamBW - Edec_k$ later from $Ssec_{m+1} \ldots Ssec_n$, so that the overall sum is still larger or equal to always streaming the exact $streamBW$.

Available *netBW* is below *streamBW* If $netBW$ falls below $streamBW$, and the client buffer is in a medium fill state, the system adapts to meet the $netBW$ as described in Section 2.2.3. If the client buffer is already overly full, it just sends out unadapted data up to $netBW$ kbits. Apparently, this second approach will drain the client buffer faster but video quality is kept constant. This also envisions the fact that people are distracted by too frequent changes in quality [85]. This draining will also not break the requirements, as long as there was a high bandwidth phase before and the above formula holds.

Fill up Buffers regardless to available *netBW* To interact even more intelligently with the client buffer, the system will also adapt the data by some small percentage even if $netBW$ equals $streamBW$. This is useful whenever the client-side buffer is below a low-level watermark and buffer underrun is near. By this the system sends the exact $netBW$, but more (adapted) $Vsecs$. This will fill up the client buffer even more ($Vsec$-wise, not byte-wise) and brings back the client buffer in a stable state faster.

8.3 Switching

To prepare a multimedia presentation for *stream switching*, several quality versions of this video have to be encoded. Since the ViTooKi MuViSever is measuring the available network every second, it decides to switch down, whenever there is too

much adaptation needed (eg. it is not useful to drop more than 10 fps) or the actual *netBW* falls below 75% of the actual stream's bandwidth. It decides to switch up whenever the bandwidth is 30% above the actual stream's *streamBW*.

Whenever available bandwidth is above or below these thresholds, the `CacheManager` class, which controls all available video streams, is queried for the corresponding `MetaObject` of the actually streamed video, whereas this `MetaObject` stores all available versions of the stream, the so-called *switch set*. If another version fits better to the newly measured bandwidth, the next possible switchpoint, reflected by an I-frame, is searched. Then the old stream is continued to be streamed out until the switchpoint is reached and all old frames until then are sent out to the client (see Figure 8.8). After that, the new stream is set active, a header frame with the new decoder configuration is sent and then the server continues with the next I-frame.

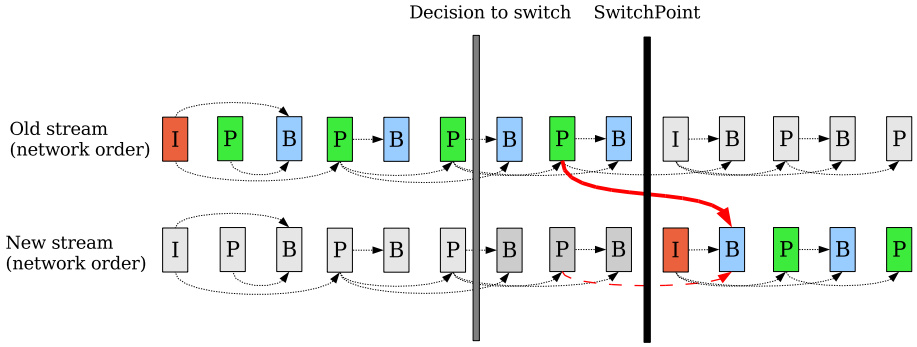


Figure 8.8: Streams are switched at the next available I-frame

This is done fully server-driven, so the client only recognizes stream switching by receiving a header frame with the new decoder configuration. MuViPlayer resets the decoder and, if the new stream is coded in a different resolution, it does the necessary up-scaling from eg. QCIF to CIF, so the user is distracted as little as possible.

As depicted in Figure 8.8, the B-frame directly after the new I-frame would need a second reference frame, which would be only available from the new stream, but there is only the old P-frame at hand, instead. Fortunately, this is not so problematic as long as both streams use the same frame pattern and the P-frames are therefore at the same positions and are coding the same visual information. Only the quantized quality differs, so in case of downswitching we get even better results when using motion vector references to higher quality macro blocks. Note, that if the P-frames

are on different positions in the streams, this might lead to remarkable drift on the wrongly coded motion vectors in this B-frame. For all used *switch sets*, the coded frame patterns where the same, only quantization was changed, so no drift problems could occur.

CHAPTER

9 Conclusion and Future Work

This work has evaluated different aspects of video streaming in best effort networks, not only as self-standing topics but – as a novelty – also combined together in a fully functional streaming environment.

Starting with a basic analysis of today's variable bitrate encoding and the associated problems of a smooth and buffer-aware streamout, a new algorithm for calculation of a constant streamout bandwidth under acceptance of a certain prefetching time was proposed. Notably, this calculated streamout bandwidth recognizably lies above the average bandwidth, since the average bandwidth is is not enough for constant streamout and leads to buffer underrun.

Using this streamout bandwidth will not be possible at all times in a best effort network, where packet loss and network congestion occur. Only different ways of adaptation can help to reduce the needed bandwidth and hereby prevent buffer underflows. Multiple approaches of scalability for *in-stream* adaptation were discussed, and temporal scalability with B-frame dropping, being the most commonly available scalability enhancement to modern video codecs, was analyzed in detail. Not only the different numbers of B-frames between reference frames and their quantization values were evaluated, but also different ways of prioritization in terms of visual quality or timely uniform distribution were discussed and measured.

For very high and long-term bandwidth fluctuations, the limitations of in-stream adaptation were outlined, and an additional coarse-grained method of adaptation was introduced. *Stream switching*, also a well-known approach for common streaming systems like RealPlayer, was analyzed. Feasible stream combinations for switch sets

were proposed. Measurements on congested networks showed the necessity of stream switching to prevent buffer underflow. Further, combining finer-grained adaptation by B-frame dropping with coarse-grained adaptation by stream switching proved to be even more successful in stabilizing the client buffer and to provide a better visual quality than simple stream switching.

All streaming was performed using Internet standards like RTSP, RTP and RTCP. Some shortcomings in those protocols were discussed and new extensions to RTP and RTCP where evaluated in terms of their performance in depth. The extensions for immediate RTCP feedback and RTP retransmission greatly improve visual quality of the decodable video, and hereby are essential extensions for all above technologies like fine- and coarse-grained adaptation and buffer smoothing. Especially immediate RTCP feedback is necessary to adjust the streamout rate in a TCP-friendly way and to react fastly and accurately to bandwidth changes with different adaptation strategies.

Future work will be the large-scale evaluation of the ViTooKi multimedia streaming environment, as an incorporation of the combined streaming strategies and multidimensional adaptation within a real streaming environment on the Internet. Real users will have to evaluate the qualitative improvements of different prioritization schemes and adaptation methods. In an already started new project, future topics like wireless network support, where streaming is subject to very high and fast bandwidth fluctuations, will be discussed. Parallel streaming of many concurrently connected clients has to be evaluated as well. This will further raise the question of adaptation and retransmission in multicast networks, where multiple clients are connected to a single streaming server. This includes analysis of layered multicast streaming and per-client or shared retransmission channels.

APPENDIX

A Implementation Details

A.1 Recursive Generation of Timely Uniform Distributed Priority Values

The following algorithm is used for timely uniform distribution (see Section 4.5.2) and sets up a table with frame priorities. Basically, it implements a recursive depth-search, which is limited to a certain depth. At each depth, left and right traversals are started, which sets ascending priority numbers to the alternating tree halves.

```
int main() {
 int prioTable[patternSize], actpos=patternSize;
 int maxdepth=(int)((log((double)patternSize)/log(2.0)) + 1);
  for (int i=0; i<maxdepth; i++)
    patSplit(prioTable,patternSize,i,&actpos);
}

void patSplit(int *pat, int len, int maxdepth, int *actpos) {
 int i=1, j=1;

  if (*actpos == len) { // init table
    for (int k=0; k<len;k++)
      pat[k]=-1;
    patSplitRec(pat, len, 0, 0, 0, actpos);
  }
```

```
  while ((i!=0) || (j!=0))  {
    i = patSplitRec(pat, len, 0, maxdepth, 1, actpos);   // next node RIGHT
    j = patSplitRec(pat, len, 0, maxdepth, 0, actpos);   // next node LEFT
  }
}

int patSplitRec(int *pat, int len, int actdepth,
                int maxdepth, int gowhere, int *actpos) {
 int half = len/2, *middle = &pat[half], ret;

if (actdepth < maxdepth) {
  if (gowhere) {
    ret = patSplitRec(pat, half, actdepth+1, maxdepth, gowhere, actpos);
    if (ret == 0)
      ret = patSplitRec(pat+half, len-half, actdepth+1,
                        maxdepth, gowhere, actpos);
      return ret;
    } else {
      ret = patSplitRec(pat+half, len-half, actdepth+1,
                        maxdepth, gowhere, actpos);
      if (ret == 0)
        ret = patSplitRec(pat, half, actdepth+1, maxdepth, gowhere, actpos);
      return ret;
    }
  } else {
    if (*middle == -1) {
      *middle = (*actpos)--;
      return 1;
    } else {
      printf("FULL\n");
      return 0;
    }
  }
}
```

A.2 Extensions for RTCP-based Feedback

The discussed extension for RTCP-based feedback was implemented according to the Internet draft [7] and its performance gains were measured in Section 7.5. In the following, detailed information is given how this implementation was done and how it can be used by other applications, including the necessary code fragments. We have extended the *UCL Common Multimedia Library* [113] of the University College London to support RTCP Feedback. For simplicity issues, only *Immediate Feedback* and *Regular RTCP Mode* are implemented. Possible feedback messages are restricted to the *Transport Layer*, so there only might be *ACK* and/or *NACK* messages on RTP packets but not on whole frames. This might increase the amount of feedback packets, but will reduce the amount of retransmission packets, since only the needed frame fragment will be resent. Note that data packets are more likely larger than feedback packets.

A.2.1 Receiver Side

To enable and hereby automatically include and send feedback on the receiver side with every RTCP packet, the UCL idea of settable RTP behaviour options is extended:

```
rtp_set_option(session, RTCP_OPT_FB_ACK, TRUE);
rtp_set_option(session, RTCP_OPT_FB_NACK, TRUE);
```

It is the application's task to decide if *ACK* or even both *ACK & NACK* is needed and really feasible with respect to network constraints.

When at least one feedback message type is enabled, the RTP library keeps track of incoming packets and stores bitmasks of missing and arrived packets. Whenever `rtcp_send_ctrl(...)` is called, all necessary feedback packets are generated and added to one large RTCP compound packet with *RTCP Sender and Receiver Reports* and *SDES* information.

To choose between *immediate* and *regular* RTCP feedback intervals, we have extended the `rtcp_send_ctrl(...)` function:

```
rtp_send_ctrl(session, rtp_ts, NULL, RTCP_NORMAL);
rtp_send_ctrl(session, rtp_ts, NULL, RTCP_IMMEDIATE);
```

If the option `RTCP_NORMAL` is chosen, the call might even return without sending anything, since the maximum RTCP bandwidth might be reached or the minimum time interval is not yet over (here, the five seconds of minimum interval and five percent of overall bandwidth apply). When set to `RTCP_IMMEDIATE`, all gathered information is sent out immediately, regardless of any restricting RTCP sending statistics and rules.

Explicit NACKs With the above described behaviour, the UCL library only sends NACKs (and ACKs) once, whenever `rtcp_send_ctrl(...)` is called. Since RTP is normally used over UDP, also NACKs could be lost. To allow an application to selectively re-request already NACKed packets, we introduced the following function:

```
rtp_send_explicit_nack(session, ssrc, rtp_seq, rtp_ts, NULL);
```

This immediately generates an RTCP receiver report and adds one NACK request for the given RTP sequence number, but does not send it yet. So again, a call of

```
rtp_send_ctrl(session, rtp_ts, NULL, RTCP_IMMEDIATE);
```

is needed, to send the RTCP receiver report with the explicit NACK.

A.2.2 Sender Side

If the receiver sends extended feedback messages (ACK and/or NACK), the sender has to somehow parse them to react accordingly.

Since the RTCP feedback is included in the compound RTCP packets, we only have to extend our applications' `RtpCallback` routine, which is called by the library, whenever new packets arrive. The library already splits the compound packet and calls the `RtpCallback` for every message part from the packet. We only had to introduce a new message type: `RX_FB`. Table A.1 shows an example implementation. Note that ACK and NACK packets are two consecutive calls to our `RtpCallback` routine! In eg. the ACK case, if one packet is not ACKed (if the bit is not set), we cannot tell right there, if this really means that a packet was lost. We have to compare it to the NACK bit of the according RTCP feedback packet.

The following example code (Table A.2) shows how to parse the feedback message and is called from the above `RtpCallback` routine. Basically, in the NACK case, the `for` loop runs through the bitmask. If there was a set bit (1), it writes out the character `D` for *Dropped*. In the ACK case, we have to first check the *Range* bit. Accordingly, we write out the following number of ACKed packets or the bit mask again.

```
void Rtp::RtpCallback(struct rtp *session, rtp_event * e) {
  rtcp_fb         *fb;

  switch (e->type) {
  case RX_RTP:          /* RTP data packet */
    printf("RX_RTP SSRC = 0x%08x\n", e->ssrc);
    /* do something */
    break;
  case RX_SR:             /* sender report */
    printf("RX_SR SSRC = 0x%08x\n", e->ssrc);
    /* do something */
    break;
  case RX_RR:           /* receiver report */
    printf("RX_RR SSRC = 0x%08x\n", e->ssrc);
    /* do something */
    break;
  case RX_FB:           /* feedback report */
    printf("RX_FB SSRC = 0x%08x\n", e->ssrc);
    fb = (rtcp_fb*)e->data;
    Rtp::fb_print(session,e->ssrc, fb);
    free(e->data);
    break;
  default:
    break;
  }
}
```

Table A.1: Example of `RtpCallback` routine with RTCP feedback support

```
void Rtp::fb_print(struct rtp *session, uint32_t ssrc, rtcp_fb *fb) {
  int i;

  printf("RX_FB SSRC = 0x%08x  ", ssrc);
  switch (fb->subtype) {
  case RTCP_FB_FMT_NACK:
    printf("NACK BLP from %6i: ",fb->fci.fb_ack.pid);
    for (i=0; i < 15; i++) {
      if ((fb->fci.fb_nack.blp & ((uint64_t)1 << i)) != 0) {
        printf("D");                          /* Dropped */
      } else {
        printf(".");          /* unknown (might be ACKed) */
      }
    }
    break;
  case RTCP_FB_FMT_ACK:
    printf("ACK  BLP from %6i: ",fb->fci.fb_ack.pid);
    if (fb->fci.fb_ack.r == 1) {                  /* range */
      printf(" to %6i",fb->fci.fb_ack.blp+fb->fci.fb_ack.pid);
    } else                                       /* bitmap */
      for (i=0; i < 15; i++) {
        if ((fb->fci.fb_ack.blp & ((uint64_t)1 << i)) != 0) {
          printf("r");                          /* received */
        } else {
          printf(".");        /* unknown (might be NACKed) */
        }
      }
    break;
  default:
    printf("FATAL: unknown Feedback Format!");
    ::exit(1);
  }
  printf("\n");
}
```

Table A.2: Example of RTCP feedback message parsing

APPENDIX

B ViTooKi Convenience Functionality

The following section will describe some ViTooKi convenience functions which can be used for better analysis of adaptation methods or received quality.

B.1 Statistics Dumps

When the *MuViServer* and *MuViPlayer* are used in the debug build, they create some output files for each instantiated *Statistics* object. The statistics class distinguishes between

- *streamout seconds*, where data comes in at any IO-Input before it is buffered or processed, and
- *playout seconds*, where the data is played out (displayed) to the client, sorted and ordered by their associated timestamps.

This allows a distinct analysis of streamed bandwidth or frame rate, and the finally stored or displayed frames.

Streamout statistics are stored in the files `rtp_stat-streamout-server-`N and `rtp_stat-streamout-client-`N resp., where N is an ascending number, reflecting the actual counter of the instantiated `Statistics` class objects. *Playout* statistics are stored in the files `rtp_stat-playout-client-`N and `rtp_stat-playout-server-`N, again with N as the ascending instance counter.

The following lists give an overview of how to interpret the file contents. All bandwidth or buffer values are in Kilobytes.

- `Statistics::writeStreamoutSecServerStats`
  writes server-side streamout statistics as tab-seperated values to `rtp_stat-streamout-server-`N, and is called every second.
  - the according streamout second,
  - $netBW$ (what we can stream out in this second)
  - data bandwidth, which was sent out in one single attempt
  - retransmitted data bandwidth
  - NACKed bandwidth
  - accumulated real $Ssec$ bandwidth (dataBW+RtxBW-NackedBW), will slightly differ with $netBW$ because of coarse-grained packet sizes
  - packet loss in percent
  - estimated actual PlayoutSec, which is always behind the streamoutSec
  - estimated amount of $Vsecs$ in client buffer
  - rate of adaptation in percent
  - size of adaptation window in $Vsec$ seconds (normally 1.0)
  - estimated client buffer fill level in Kilobyte
- `Statistics::writeStreamoutSecClientStats`
  writes client-side streamout statistics to `rtp_stat-streamout-client-`N.
  - the according streamout second,
  - data bandwidth, which was received at first attempt
  - received retransmission bandwidth
  - accumulated real $Ssec$ bandwidth (dataBW+RtxBW)
  - actual PlayoutSec, which is always behind the streamoutSec
  - amount of $Vsecs$ in client buffer
  - estimated client buffer fill level in Kilobyte
  - real client buffer fill level in Kilobyte

Note, that client streamout statistics do not have any NACKed bandwidth sizes, since the client only recognizes missing packets, but not their sizes.

- `Statistics::writePlayoutSecStats`
  is used for both, server-side (`rtp_stat-playout-server-`N) and client-side playout statistics (`rtp_stat-playout-client-`N) and writes the following values:
  - the according playout second,
  - the undecoded FPS
  - the really decoded FPS, eg. after ESSync dropping
  - data bandwidth, which arrived at first attempt
  - retransmitted data bandwidth
  - NACKed bandwidth
  - accumulated real bandwidth (dataBW+RtxBW-NackedBW) of this $Vsec$
  - average PSNR of this second

  Note again, that client playout statistics dont have any NACKed bandwidth sizes, since the client only recognizes missing packets, but not their sizes.

B.2 Priority Files

As described earlier, the adjustment of the $streamBW$ to available $netBW$ has to be done by adaptation. To drop B-frames by various prioritization algorithms, ViTooKi allows the storage of *priority files*.

When an `.mp4`-file is opened with the *MuViServer* for demuxing and streaming, all demultiplexed elementary streams are stored to the `demux` directory with the following file names, where the trailing numbers are the elementary stream Ids:

```
bsboth.mp4
   demux/bsboth.mp4.conf
   demux/localfile_bsboth.mp4.video.1
```

```
demux/localfile_bsboth.mp4.video.1.hint
demux/localfile_bsboth.mp4.audio.4
demux/localfile_bsboth.mp4.audio.4.hint
```

To generate the `.hint`-files, the `ffMP4IO` and `isoMP4IO` classes store timestamps, frame types, but also frame priorities with each issued method-call to `MP4IO::getFrame()`. For predefined priorities, it is searched for a file in the same directory as eg. `bsboth.mp4`, which has to be named similarily like the demuxed elementary stream: `bsboth.mp4.video.1.prio`. The file format is defined as follows:

```
Prio 0  Frame 0 Type I Bytes 3842
Prio 0  Frame 1 Type P Bytes 2129
Prio 13 Frame 2 Type B Bytes 907
Prio 21 Frame 3 Type B Bytes 1039
Prio 8  Frame 4 Type B Bytes 1003
Prio 22 Frame 5 Type B Bytes 912
Prio 0  Frame 6 Type P Bytes 1914
Prio 16 Frame 7 Type B Bytes 835
...
```

Note that reference frames (I- and P-frames) have a priority value of zero, which denotes their high importance.

If no `.prio`-file is found, priority values are calculated on-the-fly in `MP4IO::getFrame()` according to the proposed timely uniform distribution scheme, where each I- and P-frame is assigned a zero-priority, whereas B-frames are prioritized by fast table lookups according to the proposed algorithm for timely uniform distribution priority values.

B.3 PSNR Calculation

For various measurements and quality-wise comparisons of different adaptation algorithms, it is important to compare the displayed video with the very original video, which was used at creation (encoding) time.

Whenever *MuViPlayer* is used for video playback of eg. `bsboth.mp4`, it looks for a locally stored file called `localfile_bsboth.mp4.video.1.yuv`, which is (or is pointing to) the very original YUV video[1].

If such a file is found, each really displayed and decoded frame is further compared to the original frame and its PSNR value is calculated. This also works in the special case, where frames were dropped or were undecodable/unavailable, because the previously available frame is simply "replayed" to the PSNR comparison adaptor.

As mentioned above, the *Statistics* class is storing average PSNR values for *playout* seconds and they are dumped to the output files. This PSNR calculation is implemented as an `UncompressedVideoAdaptor`, where the original YUV-stream is passed to the constructor as a `YUVStreamIO` object.

B.4 YUVDump Adaptor

Another `UncompressedVideoAdaptor`, which is integrated into the *MuViPlayer*, is the `YUVDump` adaptor. Whenever a video is chosen for playback and PSNR calculation is invoked, each by-passing YUV-frame is stored to a file called `localfile_bsboth.mp4.video.1.dump.yuv`.

Again, missing frames are compensated by "replaying" and doubly storing the previous frame multiple times, so the dumped YUV-stream has the same length and framerate as a decoded and unadapted stream would have.

[1]The very original is different from the already encoded and then decoded video without applied adaptation! The already encoded and then decoded video will give infinite values, since this compares the very same streams (at least for the non-dropped frames).

Bibliography

[1] Jim W. Roberts, "Traffic theory and the Internet", IEEE Communications, January 2001.

[2] Gregor Gaertner and Vinny Cahill, "Understanding link quality in 802.11 mobile ad hoc networks", IEEE Internet Computing, January 2004.

[3] Buck Krasic, Kang Li, and Jonathan Walpole, "The case for streaming multimedia with TCP", Lecture Notes in Computer Science, LNCS 2158, pp. 213–218, January 2001.

[4] E. Gurses, G. Bozdagi Akar, and N. Akar, "Selective frame discarding for video streaming in TCP/IP networks", Packet Video Workshop, Nantes, France, April 2003.

[5] H. Schulzrinne and S. Casner, "RFC 1890: RTP profile for audio and video conferences with minimal control", January 1996.

[6] Jose Rey, David Leon, Akihiro Miyazaki, Viktor Varsa, and Rolf Hakenberg, "RTP retransmission payload format", draft-ietf-avt-rtp-retransmission-04.txt, December 2002, expires May 2003.

[7] Joerg Ott, Stephan Wenger, Noriyuki Sato, Carsten Burmeister, and Jose Rey, "Extended RTP profile for RTCP-based feedback (RTP/AVPF)", draft-ietf-avt-rtcp-feedback-04.txt, October 2002, expires April 2003.

[8] Rob Koenen, "Overview of the MPEG-4 standard", ISO/IEC JTC1/SC29/WG11 N4030, March 2001.

[9] Mingzhe Li, Mark Claypool, and Robert Kinicki, "MediaPlayer versus RealPlayer - A comparison of network turbulence", Proceedings of the ACM SIGCOMM Internet Measurement Workshop, Marseille, France, November 2002.

[10] Wu chi Feng and Jennifer Rexford, "A comparison of bandwidth smoothing techniques for the transmission of prerecorded compressed video", IEEE INFOCOM, April 1997, pp. 58-66.

[11] Wu chi Feng, Ming Liu, and Chi Chung Lam, "A movie approximation technique for the implementation of fast bandwidth smoothing algorithms", SPIE/IS&T Multimedia Computing and Networking, February 1998.

[12] James D. Salehi, Zhi-Li Zhang, James F. Kurose, and Don Towsley, "Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing", ACM SIGMETRICS, May 1996.

[13] Simon S. Lam, Simon Chow, and David K. Y. Yau, "An algorithm for lossless smoothing of MPEG video", ACM SIGCOMM, August 1994.

[14] Marwan Krunz and Herman Hughes, "A traffic model for MPEG-Coded VBR streams", ACM SIGMETRICS, 1995.

[15] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "RFC 2205: - Resource ReSerVation Protocol (RSVP)", September 1997.

[16] Chia-Hui Wang, Jan-Ming Ho, Ray-I Chang, and Shun-Chin Hsu, "A control-theoretic method of rate-based flow control of multimedia communication", Tech. Rep. TR-IIS-01-007, Institute of Information Science, Academia Sinica, Taipei, Taiwan, 2001.

[17] Subhabrata Sen, Jennifer Rexford, Jayanta K. Dey, and James F. Kurose, "Online smoothing of variable-bit-rate streaming video", IEEE Transactions on Multimedia, vol. 2, no. 1, pp. 37–48, 2000.

[18] Zhi-Li Zhang, Srihari NElakuditi, Rahul Aggarwal, and Rose P. Tsang, "Efficient selective frame discard algorithms for stored video delivery across resource constrained networks", Journal of Real-Time Imaging, 2000.

[19] Wu chi Feng, Brijesh Krishnaswami, and Arvind Prabhudev, "Proactive buffer management for the streamed delivery of stored video", ACM Multimedia Conference, September 1998.

[20] "Microsoft MPEG-4 Video Reference Software", http://megaera.ee.nctu.edu.tw/mpeg/Ref_Software/.

[21] Klaus Leopold, "Quality controlled temporal video adaptation", M.S. thesis, University of Klagenfurt, January 2003.

[22] Anthony Vetro, Charilaos Christopoulos, and Huifang Sun, "Video transcoding architectures and techniques: An overview", IEEE Signal Processing Magazine, pp. 18–29, March 2003.

[23] Bernhard Penz, "Video transcoding and its application in a video proxy", M.S. thesis, University of Klagenfurt, February 2003.

[24] Gregory Conklin, Gary Greenbaum, Karl Lillevold, Alan Lippman, and Yuriy Reznik, "Video coding for streaming media delivery on the Internet", IEEE Transactions on Circuits and Systems for Video Technology, vol. 11, no. 3, March 2001.

[25] Fernando Pereira and Touradj Ebrahimi, *The MPEG-4 Book*, Prentice Hall, 2002.

[26] Gertjan Keesman, Robert Hellinghuizen, Fokke Hoeksema, and Geert Heideman, "Transcoding of MPEG bitstreams", Signal Processing: Image Communication, pp. 481–500, September 1996.

[27] Huifang Sun, Wilson Kwok, and Joel Zdepski, "Architectures for MPEG compressed bitstream scaling", IEEE Trans. on Circuits and Systems for VideoTechnology, pp. 191–199, October 1995.

[28] Zhijun Leiand Nicolas D. Georganas, "Rate adaptation transcoding for precoded video streams", Proceedings of ACM Multimedia, pp. 127–136, December 2002.

[29] Rob Koenen, "Profiles and levels in MPEG-4: Approach and overview", Image Communication Journal. Tutorial Issue on the MPEG-4 Standard, vol. 15, no. 1-2, January 2000.

[30] V.G. Ruiz, M.F. Lopez, I. Garcia, and E.M.T. Hendrix, "JPEG2000 vs. JPEG in MPEG encoding", Proceedings of the First International Workshop on Interactive Rich Media Content Production: Architectures, Technologies, Applications, Tools, Lausanne, Switzerland, pp. 61–67, October 2003.

[31] George Fankhauser, Marcel Dasen, Nathalie Weiler, Bernhard Plattner, and Burkhard Stiller, "WaveVideo – an integrated approach to adaptive wireless video", ACM Monet, Special Issue on Adaptive Mobile Networking and Computing, 1999.

[32] Weiping Li, "Overview of fine granularity scalability in MPEG-4 video standard", IEEE Trans. Circuits and Systems for Video Technology, vol. 11, no. 3, March 2001.

[33] Matthias Ohlenroth and Hermann Hellwagner, "Quality adaptation options of MPEG-4 video streams", Tech. Rep. TR/ITEC/01/1.03, University of Klagenfurt, December 2001.

[34] Philippe de Cuetos, Martin Reisslein, and Keith W. Ross, "Evaluating the streaming of FGS-encoded video with rate-distortion traces", Tech. Rep. RR-03-078, Institut Eurecom, June 2003.

[35] Mihaela van der Schaar and Hayder Radha, "Adaptive motion-compensation fine-granular-scalability (AMC-FGS) for wireless video", IEEE Transactions on Circuits and Systems for Video Technology, pp. 360–371, June 2002.

[36] Feng Wu, Shipeng Li, and Ya-Qin Zhang, "A framework for efficient progressive fine granularity scalable video coding", IEEE Transactions on Circuits and Systems for Video Technology, pp. 332–344, March 2001.

[37] "MoMuSys: Mobile multimedia systems", ISO/IEC 14496 MPEG-4 Video Reference Software, ACTS-AC098, 1995-1999, Partners: Bosch, Siemens, University of Hamburg and Madrid, Deutsche Telekom, Heinrich Hertz Institut, and more.

[38] Rakesh Dugad and Narendra Ahuja, "A scheme for spatial scalability using nonscalable encoders", IEEE Trans. Circuits and Systems for Video Technology, vol. 12, pp. 620–627, July 1996.

[39] Qi Wang, Feng Wu, Shipeng Li, Yuzhuo Zhong, and Ya-Qin Zhang, "Fine-granularity spatially scalable video coding", IEEE International conference on Acoustics, Speech and Signal Processing (ICASSP), Salt Lake City, pp. 1801–1804, May 2001.

[40] Chuohao Yeo, "An investigation of methods for digital television format conversions", M.S. thesis, Massachusetts Institute of Technology, May 2002.

[41] Touradj Ebrahimi and Murat Kunt, "Object-based video coding", Handbook of Image and Video Processing, pp. 585–596, 2000.

[42] Yasser Pourmohammadi-Fallah, Kambiz Asrar-Haghighi, and Hussein Alnuweiri, "Internet delivery of MPEG-4 object-based multimedia", IEEE Multimedia, vol. 10, pp. 68 –78, July 2003.

[43] Alexandre R.J. François and Gérard G. Medioni, "Adaptive color background modeling for real-time segmentation of video streams", Proceedings of the International Conference on Imaging Science, Systems, and Technology, Las Vegas, NA, pp. 227–232, June 1999.

[44] C. Schnorr and W. Peckard, "Motion-based identification of deformable templates", International Conference on Computational Analysis of Images and Patterns, Prague, September 1995.

[45] Anthony Vetro, Huifang Sun, and Yao Wang, "Object-based transcoding for adaptable video content delivery", IEEE Transactions on Circuits and Systems for Video Technology, vol. 11, no. 3, March 2001.

[46] Jose M. Martinez, "Overview of the MPEG-7 standard", ISO/IEC JTC1/SC29/WG11 N4031, March 2001.

[47] D. Irwin, *Visual Memory Within and Across Fixations*, chapter Eye movements and Visual Cognition: Scene Preparation and Reading, pp. 146–165, New York: Springer-Verlag, 1992.

[48] Javed I. Khan and Olewg Komogortsev, "Dynamic object window prediction for perceptual coding of wide format video with eye-gaze tracking", Tech. Rep. TR2002-11-01, Kent State University, November 2002.

[49] Scott Daly, Kristine Matthews, and Jordi Ribas-Corbera, "Visual eccentricity models in face-based video compression", Proceedings of SPIE: Human Vision and Electronic Imaging IV, vol. 3644, pp. 152–166, 1999.

[50] Mihaela van der Schaar and Hayder Radha, "A hybrid temporal-SNR fine-granular scalability for Internet video", IEEE Transactions on Circuits and Systems for Video Technology, pp. 360–371, March 2001.

[51] Feng Wu, Shipeng Li, Rong Yan, Xiaoyan Sun, and Ya-Qin Zhang, "Efficient and universal scalable video coding", ICIP 2002, Rochester, pp. 37–40, September 2002.

[52] Chung-Neng Wang, Chia-Yang Tsai, Yao-Chung Lin, Han-Chung Lin, Hsiao-Chiang Chuang, Jin-He Chen, Kin Lam Tong, Feng-Chen Chang, Chun-Jen Tsai, Tihao Chiang, Shuh-Ying Lee, and Hsueh-Ming Hang, "FGS-based video streaming test bed for media coding and testing in streaming environments", ISO/IEC JTC1/SC29/WG11 MPEG2002/M9182, December 2002.

[53] Sam Lerouge, Peter Lambert, and Rik Van de Walle, "Multi-criteria optimization for scalable bitstreams", Visual Content Processing and Representation, 8th International Workshop, VLBV 2003, Madrid, pp. 122–130, September 2003.

[54] Anthony Vetro, "MPEG-21 Requirements on Digital Item Adaptation", ISO/IEC JTC1/SC29/WG11 N4515, December 2001.

[55] Sylvain Devillers, "Bitstream syntax definition language (BSDL): An input to MPEG-21 content representation", ISO/IEC JTC1/SC29/WG11 M7053, March 2001.

[56] Michael Kropfberger and Peter Schojer, "ViTooKi – The Video ToolKit", http://ViTooKi.sourceforge.net.

[57] Peter Schojer, Laszlo Böszörmenyi, and Hermann Hellwagner, "An adaptive MPEG-4 proxy cache", Tech. Rep. TR/ITEC/02/2.07, University of Klagenfurt, June 2002.

[58] Peter Schojer, Laszlo Böszörmenyi, Hermann Hellwagner, Bernhard Penz, and Stefan Podlipnig, "Architecture of a Quality Based Intelligent Proxy (QBIX) for MPEG-4 Videos", ACM World Wide Web Conference, May 2003.

[59] Jürg Bolliger, *A Framework for Network-Aware Applications*, Ph.D. thesis, ETH Zurich, 2000.

[60] Dietmar Jannach, Klaus Leopold, Hermann Hellwagner, and Christian Timmerer, "A knowledge based approach for multi-step media adaptation", Proceedings of the 5th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS), April 2004.

[61] Kentarou Fukuda, Naoki Wakamiya, Masayuki Murata, and Hideo Miyahara, "QoS mapping between user's preference and bandwidth control for video transport", Proceedings of Fifth IFIP International Workshop on Quality of Service (IWQoS), pp. 291–302, May 1997.

[62] G. Ghinea and J.P. Thomas, "QoS impact on user perception and understanding of multimedia video clips", Proceedings of ACM Multimedia, Bristol, United Kingdom, pp. 49–54, 1998.

[63] Anna Bouch and M. Angela Sasse, "Network quality of service: What do users need?", Proceedings of the 4th International Distributed Conference (IDC'99), pp. 78–90, 1999.

[64] Peter Schojer, Laszlo Böszörmenyi, and Hermann Hellwagner, "QBIX-G - a transcoding multimedia proxy", Tech. Rep. TR/ITEC/04/2.16, University of Klagenfurt, August 2004.

[65] "FFmpeg", http://ffmpeg.sourceforge.net/.

[66] "XviD", http://www.xvid.org/.

[67] Tamer Shanableh and Mohammed Ghanbari, "The importance of the bi-directionally predicted pictures in video streaming", IEEE Trans. Circuits and Systems for Video Technology, vol. 11, no. 3, March 2001.

[68] Jenq-Neng Hwang, Tzong-Der Wu, and Chia-Wen Lin, "Dynamic frame-skipping in video transcoding", Proceedings of the IEEE Workshop on Multimedia Signal Processing, pp. 616–621, December 1998.

[69] Jeongnam Youn, Ming-Ting Sun, and Chia-Wen Lin, "Motion vector refinement for high-performance transcoding", IEEE Transactions on Multimedia, vol. 1, no. 1, pp. 30–40, 1999.

[70] Kun Tan, Richard Ribier, and Shih-Ping Liou, "Content-sensitive video streaming over low bitrate and lossy wireless network", ACM Multimedia, pp. 512–515, 2001.

[71] D. A. Robinson, "The mechanics of human smooth pursuit eye movement", The Journal of Physiology, vol. 180, 1965.

[72] Matthias Ohlenroth and Hermann Hellwagner, "A protocol for adaptation-aware multimedia streaming", Proceedings of ICME, Baltimore, July 2003.

[73] Matthias Ohlenroth, *Network-based Adaptation of Multimedia Contents*, Ph.D. thesis, University of Klagenfurt, September 2003.

[74] Sandeep Bajaj, Lee Breslau, and Scott Shenker, "Uniform versus priority dropping for layered video", in *SIGCOMM*, September 1998, pp. 131–143.

[75] Tianming Liu, H.J. Zhang, and Fei-Hu Qi, "Perceptual frame dropping in adaptive video streaming", Proceedings of IEEE International Symposium on Circuits and Systems, Arizona, May 2002.

[76] Wai tian Tan, *Video Compression and Streaming over Packet-switched Networks*, Ph.D. thesis, University of California at Berkeley, 2000.

[77] Klaus Leopold, Hermann Hellwagner, and Michael Kropfberger, "QCTVA - quality controlled temporal video adaptation", Proceedings of SPIE Vol. 5242, pp. 163–174, 2003.

[78] Michael Kropfberger and Hermann Hellwagner, "Evaluation of RTP immediate feedback and retransmission extensions", Proceedings of ICME 2004, June 2004.

[79] Jae Chung, Mark Claypool, and Yali Zhu, "Measurement of the congestion responsiveness of RealPlayer streaming video over UDP", Proceedings of the Packet Video Workshop (PV), April 2003.

[80] Roger Karrer and Thomas Gross, "Dynamic handoff of multimedia streams", Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video, pp. 125–133, 2001.

[81] Philippe Gentric, "Requirements and use cases for stream switching", draft-gentric-mmusic-stream-switching-req-00.txt, May 2003, expired November 2003.

[82] Linda Wu, Rosen Sharma, and Brian Smith, "Thin streams: An architecture for multicasting layered video", in *NOSSDAV'97*, May 1997.

[83] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang, "A reliable multicast framework for light-weight sessions and application level framing", IEEE/ACM Transactions on Networking, vol. 5, no. 6, pp. 784–803, 1997.

[84] Reza Rejaie, Mark Hanley, and Deborah Estrin, "Layered quality adaptation for Internet video streaming", IEEE JSAC. Special Issue on Internet QoS, Winter 2000.

[85] Reza Rejaie, *An End-To-End Architecture for Quality Adaptive Streaming Applications in the Internet*, Ph.D. thesis, University of Southern California, December 1999.

[86] Philippe de Cuetos, Despina Saparilla, and Keith W. Ross, "Adaptive streaming of stored video in a TCP-friendly context: Multiple versions or multiple layers", International Packet Video Workshop, Kyongju, Korea, April 2001.

[87] Taehyun Kim and Mostafa H. Ammar, "A comparison of layering and stream replication video multicast schemes", Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video, pp. 63–72, 2001.

[88] Jian Lu, "Signal processing for Internet video streaming: A review", Proceedings of SPIE Image and Video Communications and Processing, 2000.

[89] Mihaela van der Schaar, "Using S-frames for fast switching between FGS streams and switching between MC-FGS structures to limit prediction-drift", ISO/IEC JTC1/SC29/WG11 M8140, March 2002.

[90] Marta Karczewicz and Ragip Kurceren, "The SP- and SI-frames design for H.264/AVC", IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, pp. 637–644, July 2004.

[91] "Objective Perceptual Video Quality Measurement Using a JND-Based Full Reference Technique", Technical Report T1.TR.PP.75-2001, Alliance for Telecommunications Industry Solutions ATIS, October 2001.

[92] Volker Zota, "Kompressionisten - Aktuelle Video-Codecs im Vergleich", Computer-Technik c't Magazine, October 2003, p. 146.

[93] "OpenDivX", http://www.projectmayo.com/.

[94] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RFC 1889: RTP: a transport protocol for real-time applications", January 1996.

[95] D. McGrew and D. Oran, "The secure real time protocol", IETF Internet-Draft, draft-mcgrew-avt-srtp-00.txt, November 2000.

[96] Y. Kikuchi, T. Nomra, and S. Fukungaga, "RFC 3016: RTP payload format for MPEG-4 audio/visual streams", November 2000.

[97] J. van der Meer, D. Curet, E. Gouleau, S. Relier, C. Roux, P. Clement, and G. Cherry, "RTP payload format for MPEG-4 flexmultiplexed streams", Internet draft, draft-curet-avt-rtp-mpeg4-flexmux-02.txt, November 2002.

[98] Matthias Ohlenroth and Hermann Hellwagner, "RTP-packetization of MPEG-4 elementary streams", Tech. Rep. TR/ITEC/02/1.01, University of Klagenfurt, March 2002.

[99] D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar, "RFC 2250: RTP payload format for MPEG1/MPEG2 video", January 1998.

[100] C. Perkins, I. Kouvelas, O. Hodson, V. Hardman, M. Handley, J.C. Bolot, A. Vega-Garcia, and S. Fosse-Parisis, "RFC 2198: RTP payload for redundant audio data", IETF, September 1997.

[101] L. Berc, W. Fenner, R. Frederick, S. McCanne, and P. Stewart, "RFC 2435: RTP payload format for JPEG-compressed video", October 1998.

[102] Jan van der Meer, David Mackie, and Viswanathan Swaminathan, "Internet streaming media alliance implementation specification", August 2001, Version 1.0.

[103] M. Handley and V. Jacobson, "RFC 2327: Session description protocol (SDP)", IETF, April 1998.

[104] H. Schulzrinne, A. Rao, and R. Lanphier, "RFC 2326: Real time streaming protocol (RTSP)", IETF, April 1998.

[105] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616: Hypertext Transfer Protocol – HTTP/1.1", June 1999.

[106] Philippe Gentric, "RTSP stream switching", draft-gentric-mmusic-stream-switching-01.txt, January 2004, expires July 2004.

[107] J. Bormans, J. Gelissen, and A. Perkis, "MPEG-21: The 21st century multimedia framework", IEEE Signal Processing Magazine, vol. 20, pp. 53 – 62, March 2003.

[108] Ian Burnett et.al, "MPEG-21: Goals and achievements", IEEE Multimedia, vol. 10, pp. 60–70, October 2003.

[109] Ingo Wolf, Bernhard Feiten, Teodora Guenkova-Luy, Andreas Schorr, Franz Hauck, and Andreas J. Kassler, "MPEG-21 DIA based delivery using SDPng and RTP", ISO/IECT JTC1/SC29/WG11 M10996, July 2004.

[110] Christian Leicher, "Hierarchical encoding of MPEG sequences using priority encoding transmission (PET)", Tech. Rep. TR-94-058, International Computer Science Institute, Berkeley, CA, November 1994.

[111] Andres Albanese and Giancarlo Fortino, "Robust transmission of MPEG video streams over lossy packet-switching networks by using PET", Tech. Rep. TR-99-014, International Computer Science Institute, Berkeley, CA, June 1999.

[112] Mike Piecuch, Ken French, George Oprica, and Mark Claypool, "A selective retransmission protocol for multimedia on the Internet", Proceedings of SPIE International Symposium on Multimedia Systems and Applications, Boston, November 2000.

[113] University College London, "UCL common multimedia library", http://www-mice.cs.ucl.ac.uk/multimedia/software/common/.

[114] "MPEG4IP", http://mpeg4ip.sourceforge.net/.

[115] "Helix Community", https://helixcommunity.org/.

[116] "VideoLAN", http://www.videolan.org/.

[117] Haifeng Xu, Joe Diamand, and Ajay Luthra, "Client architecture for MPEG-4 streaming", IEEE Multimedia, vol. 11, pp. 16–23, April-June 2004.

[118] Peter Schojer, *QBIX-G - A Quality Based Intelligent Proxy Gateway*, Ph.D. thesis, University of Klagenfurt, 2004.

[119] Sally Floyd and Kevin Fall, "Promoting the use of end-to-end congestion control in the Internet", IEEE/ACM Transactions on Networking, August 1999.

[120] Shanwei Cen, Calton Pu, and Jonathan Walpole, "Flow and congestion control for Internet streaming applications", Tech. Rep. CS-97-03, Oregon Graduate Institute of Science and Technology, 1998.

[121] Reza Rejaie, Mark Handley, and Deborah Estrin, "RAP: An end to end rate-based congestion control mechanism for realtime streams in the Internet", Proceedings of IEEE Infocom, New York, March 1999.

[122] YoungGook Kim, JongWon Kim, , and C.-C. Jay Kuo, "Smooth and fast rate adaptation mechanism (SFRAM) for TCP-friendly Internet video", Proceedings of the International Packet Video Workshop, Sardinia, Italy, May 2000.

[123] Dorgham Sisalem and Henning Schulzrinne, "The loss-delay based adjustment algorithm: A TCP-friendly adaptation scheme", in *Proceedings of NOSSDAV*, Cambridge, UK., 1998.

[124] Dorgham Sisalem and Adam Wolisz, "LDA+: A TCP-friendly adaptation scheme for multimedia communication", in *IEEE International Conference on Multimedia and Expo (III)*, 2000, pp. 1619–1622.

[125] Michael Welzl, *Scalable Performance Signalling and Congestion Avoidance*, Ph.D. thesis, Technische Universität Darmstadt, 2002.

[126] Jörg Widmer, Robert Denda, and Martin Mauve, "A survey on TCP-friendly congestion control", IEEE Network, vol. 15, no. 3, pp. 28–37, 2001.

[127] Ingo Buchbauer, "Flusskontrolle und QoS-Feedback für Multimedia-Streaming-Protokolle", M.S. thesis, University of Klagenfurt, May 2003.

[128] I. Busse, B. Deffner, and H. Schulzrinne, "Dynamic QoS control of multimedia applications based on RTP", First International Workshop on High Speed Networks and Open Distributed Platforms, St. Petersburg, Russia, June 1995.

[129] Yeali S. Sun, Fu-Ming Tsou, and Meng Cheng Chen, "Predictive flow control for TCP-friendly end-to-end real-time video on the Internet", Computer Communications, pp. 1230–1242, August 2002.

[130] Sally Floyd and Van Jacobson, "Random early detection gateways for congestion avoidance", IEEE Transactions on Networking, vol. 1, no. 4, pp. 397–413, August 1993.