Prüfer:	Prof. Dr. Paul Levi
Betreuer:	Dipl. Inf. Niels Mache

begonnen am:	12.6.97
beendet am:	12.12.97
CR-Klassifikation:	C.2.2, H.5.1, I.3.2, I.4.2

Studienarbeit Nr. 1658

Entwicklung eines Protokolls für die Übertragung von Video- und Audiodaten über unzuverlässige Netzwerkverbindungen

Dirk Farin

Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Inhaltsverzeichnis

Kapitel 1

Einleitung

Videoübertragungen sind ein Medium, das immer mehr in neue Anwendungsbereiche eindringt. Waren bis vor kurzem noch das öffentliche Fernsehen und Firmen mit Bedarf an teuren Videokonferenzen die einzigen Anwender, so entstehen immer mehr low-cost Anwendungen wie Bildtelefonie, eine fortschreitende Divertisierung des Fernsehens in eine große Zahl von Spartenkanälen, Fernüberwachungs- und Fernsteuerungsaufgaben bis hin zu Video on Demand-Systemen. Diese neuen Anwendungen sind nur realisierbar, wenn eine Übertragungstechnik benutzt wird, die deutlich preisgünstiger ist und eine wesentlich geringere Übertragungsbandbreite benötigt als die bisherigen analogen Videosysteme. Für VOD-Datenbanken (Video on Demand) bietet es sich an, die bisher bestehende Infrastruktur des Internets auch für die Videoübertragung zu benutzen, ohne daß das Internet komplett für diesen neuen Dienst (z.B. durch neue Routingprotokolle) angepaßt werden muß. In seiner jetzigen Form ist für das Internet noch kein Protokoll im allgemeinen Einsatz, das die für Videoübertragungen wichtige Bandbreitenreservierung unterstützen würde. Eine schnelle Übertragung ist derzeit nur möglich, wenn Paketverluste bei der Übertragung in Kauf genommen werden.

In dieser Arbeit soll ein Protokoll entwickelt werden, das die Echtzeitübertragung von Video- und Audiodaten auf einem unzuverlässigen Netzwerk ohne garantierte Laufzeiten ermöglichen soll. Dabei liegt der Schwerpunkt auf einer One-to-one Übertragung, d.h. die Anzahl der Clients, die auf einen Server zugreift, soll als gering angenommen werden. Der Server kann somit den Datenstrom in Echtzeit verändern und an die aktuelle Netzwerkauslastung und Performance des Clients anpassen. Somit ist eine in jeder Situation optimale Übertragung möglich.

Aufgrund der großen Verbreitung des MPEG-Standards soll ein System, das auf diesem Standard beruht, den Schwerpunkt dieser Arbeit bilden. Das vorgestellte Protokoll ist aber unabhängig von einer speziellen Videocodierung, und die entwickelte Software sollte sich ohne großen Aufwand um andere Codierungsverfahren erweitern lassen.

Kapitel 2

Prinzipien der Videokompression

2.1 Warum überhaupt komprimieren ?

Das heute übliche analoge PAL-FBAS Videoverfahren benutzt eine Vollbildfrequenz von 25 Hz und eine vertikale Auflösung von 625 Zeilen pro Vollbild. Für das eigentliche Videosignal werden aber nicht alle benutzt, da in diesen Zeilen oft Zusatzinformationen wie Videotext oder Meßsignale untergebracht sind. Meist geht man deshalb von 576 sichtbaren Zeilen aus und erhält damit (bei einer Digitalisierung mit quadratischen Pixeln) aufgrund des Seitenverhältnisses von 4:3 eine Breite von 768 Pixel. Mit einer Quantisierung von 8 bit pro Primärfarbe ergibt sich eine Datenmenge von $576 \cdot 768 \cdot 3 \cdot 1\ Byte = 1,327,104\ Byte$, wofür man eine Übertragungskapazität von $1,327,104\ Byte \cdot \frac{25}{s} \approx 31\ \frac{MByte}{s} \approx 253\ \frac{MBit}{s}$ bräuchte.

Eine unkomprimierte digitale Übertragung ist somit aus wirtschaftlichen Gründen fast unmöglich. Zukünftige Videoformate wie HDTV mit noch höheren Auflösungen als PAL wären nicht realisierbar. Aber auch für Anwendungen, bei denen nicht so hohe Auflösungen und Bildraten erforderlich sind (wie z.B. Bildtelefone oder Videokonferenzsysteme) erfordern eine erhebliche Datenkompression, da deren Übertragungskanäle meist sehr schmalbandig sind (z.B. 64 kbit/s für eine ISDN-Leitung).

Hinzu kommt meist auch noch das Audiosignal, das aber in Vergleich zum unkomprimierten Video ein relativ geringes Datenvolumen hat. Im komprimierten Fall darf das Datenvolumen des Audiosignals aber nicht vernachlässigt werden, da es sich i.A. nicht so sehr komprimieren läßt wie das Videosignal.

Verlustlose Kompressionsalgorithmen sind nur bei Bildern ohne wesentliches Rauschen (z.B. Computeranimationen) effektiv, bringen aber bei natürlichem Bildmaterial aufgrund des dort immer vorhandenen Bildrauschens keine nennenswerte Kompression. Gängige Verfahren wie LZW sind für Echtfarbdaten nicht besonders geeigent, selbst ein Kompressionsfaktor 2 ist damit selten erreichbar.

Die derzeit besten Algorithmen zur verlustlosen Bildkompression sind „Lossless JPEG Huffman" (4.08 bits/pixel), „FELICS" (3.76 bpp), „Lossless JPEG arithmetic" (3.40 bpp), „LOCO-I" [WSS96] (3.18 bpp) und „CALIC-arithmetic" (3.05 bpp), (in aufsteigender Leistungsfähigkeit und Komplexität). Die bits/pixel-Angaben sind natürlich nur Mittelwerte, die in der Praxis wirklich erzielten Werte sind stark vom Bildtyp abhängig. Selbst mit CALIC-arithmetic würde das Datenvolumen pro Bild immer noch ca. 1.35 MBit betragen, also ca. 34 MBit/s (entspricht Kompressionsfaktor 7.8), was immer noch zu hoch für eine ökonomische Übertragung ist. Für noch höhere Kompressionsraten (ca. Faktor 100) kommen deshalb nur verlustbehaftete Verfahren in Frage.

2.2 RGB-, YUV- und YCrCb-Farbsysteme

In Farbbildschirmen werden zur Farbmischung die drei Farben rot, grün und blau benutzt, da die drei Typen von *Zapfen* im menschlichen Auge eben besonders auf diese Farben reagieren.[1] Dabei sind die Zapfen für grünes Licht am empfindlichsten und die für blaues am unempfindlichsten. Da im Auge viel mehr Stäbchen als Zapfen vorhanden sind, ist die Farbauflösung niedriger als die Helligkeitsauflösung. Dies macht man sich zunutze, indem man die Farben in einen Farbraum transformiert, in dem ein Kanal die Gesamthelligkeit enthält und zwei weitere die Farbinformation.[2]

Im PAL-System wird dazu der YUV-Raum benutzt. Y ist dabei der Helligkeitskanal und berechnet sich zu:

$$Y = 0.212671 \cdot R' + 0.715160 \cdot G' + 0.072169 \cdot B'$$

U und V ist $U = 0.492111(B' - Y)$ und $V = 0.877283(R' - Y)$, R', G', B' sind die gammakorrigierten Werte zu R, G, B.

Im Computerbereich wird dagegen häufig der YCrCb-Farbraum benutzt. Bei MPEG wird dabei diese Transformation von RGB nach YCrCb benutzt:

$$\begin{pmatrix} Y \\ Cr \\ Cb \end{pmatrix} = \begin{pmatrix} 0.257 & 0.504 & 0.098 \\ 0.439 & -0.368 & -0.071 \\ -0.148 & -0.291 & 0.439 \end{pmatrix} \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} + \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix}$$

Und umgekehrt:

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \begin{pmatrix} 1.164 & 1.596 & 0 \\ 1.164 & -0.813 & -0.392 \\ 1.164 & 0 & 2.017 \end{pmatrix} \begin{pmatrix} Y \\ Cr \\ Cb \end{pmatrix} + \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix}$$

2.3 Interlace

Um die Bildwiederholfrequenz zu erhöhen, ohne die Datenrate erhöhen zu müssen, wird beim analogen-FBAS System nur jede zweite Zeile des Bildes übertragen. Man erhält somit zwei Halbbilder mit jeweils der halben Zeilenzahl. Dabei wird nicht ein vorhandenes Vollbild in zwei Halbbilder zerteilt, sondern zwei Halbbilder mit der doppelten Bildfrequenz aufgenommen.

Zwar beträgt die Vollbildfrequenz bei PAL dann immer noch 25 Hz, da aber für ein Vollbild zwei Halbbilder nacheinander aufgebaut werden, ergibt sich eine Halbbildwiederholfrequenz von 50 Hz, was das wahrgenommene Flimmern deutlich verringert.

Dieses Verfahren bereitet aber große Schwierigkeiten, wenn vorhandenes Bildmaterial im Interlace-Format mit einer anderen Bildwiederholfrequenz oder auf Computermonitoren, die in der Regel nicht mit dem Interlaceverfahren arbeiten, dargestellt werden sollen. Man betrachte eine Szene, in der sich eine scharfe vertikale Kante in horizontaler Richtung bewegt. Da sich diese Kante in zwei aufeinanderfolgenden Halbbildern nicht mehr an der gleichen Position befindet, entsteht an der Kante ein Zick-Zack-Muster (der sogenannte „Mäusezähnchen-Effekt“, siehe Abbildung 2.1).[3]

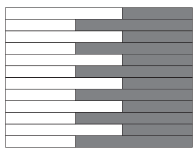


Abbildung 2.1: der Mäusezähncheneffekt

[1] Die *Stäbchen* im Auge reagieren nur auf Gesamthelligkeit.

[2] Die Transformation ist abhängig von der genauen Wellenlänge der Farben rot, grün und blau. Hier wird die Lage, die in Recommendation ITU-R 709 beschrieben ist, angenommen.

[3] Wendet man auf ein Vollbild mit „Mäusezähnchen-Effekt“ eine Transformation in den Frequenzbereich an (wie z.B. DCT), dann treten diese Zähnchen als sehr hohe vertikale Frequenzen in Erscheinung. Deshalb definiert MPEG-2 eine von MPEG-1 abweichende, alternative ZigZag-Ordnung, in der die DCT-Koeffizienten so angeordnet werden, daß hohe vertikale Frequenzen weiter vorne erscheinen.

2.4 Physiologische Effekte der Filmwahrnehmung

2.4.1 Activity masking

An Bildstellen, an denen sich kontrastreiche Kanten befinden, hat das Auge Schwierigkeiten, kleine Variationen der Helligkeit wahrzunehmen. Dieselben Variationen in „weichen“ Bildbereichen werden aber hingegen gut wahrgenommen. In quantisierungsbasierten Kompressionsverfahren kann es deshalb von Vorteil sein, die Anzahl der Quantisierungsstufen innerhalb eines Bildes zu variieren.

Allerdings zeigen Arbeiten von B. E. Rogowitz und Brian A. Wandell [Rog83, Wan95], daß dies ein komplexer Effekt ist, der in bestimmten Situationen nicht so einfach wie oben beschrieben zu gelten scheint.

Es darf jedoch daraus der Schluß gezogen werden, daß es zur Reduktion der Datenmenge nicht besonders gut ist, hohe Frequenzen einfach abzuschneiden, da dann scharfe Kanten verschwimmen würden. Ein besseres Verfahren ist dagegen, bei hohen Frequenzen die Anzahl der Quantisierungsstufen zu verringern.

2.4.2 Temporal masking

Bei einem plötzlichen Szenenwechsel braucht das Sehsystem des Menschen einige Zeit, um sich der neuen Szene anzupassen und Details darin wahrzunehmen. Ein ähnlicher Effekt existiert auch für den umgekehrten Fall, daß nämlich Bildartifakte kurz vor einem Szenenwechsel nicht mehr wahrgenommen werden.

Dieser Effekt ist äußerst hilfreich für die Kompression, da es an Szenenwechseln, bei denen sich der Bildinhalt in der Folgeszene von der letzten wesentlich unterscheidet, wahrscheinlich ist, daß bei gleichen Codierungsparametern ein erhöhtes Datenvolumen erzeugt wird. Aufgrund des Temporal masking Effektes kann man ohne weiteres die Codierungsqualität an diesen Stellen senken, um das Datenvolumen pro Bild konstant zu halten.

2.5 Methoden zur Videokompression

2.5.1 DCT-Transformation

Der eindimensionale Fall

Die Diskrete Cosinus Transformation (DCT) ist ähnlich der schnellen Fouriertransformation (FFT) eine Transformation zwischen Ortsraum und (Orts-)Frequenzraum. Die eindimensionale DCT ist definiert als

$$F(u) = \frac{C(u)}{2} \sum_{x=0}^{7} f(x) \cos((2x+1)u\pi/16)$$

mit

$$C(u) = \begin{cases} 1/\sqrt{2} & \text{für } u = 0 \\ 1 & \text{für } u > 0 \end{cases}$$

und die dazugehörige inverse Transformation (IDCT) als

$$f(x) = \sum_{u=0}^{7} \frac{C(u)}{2} F(u) \cos((2x+1)u\pi/16)$$

Abbildung 2.2 auf der nächsten Seite zeigt die Basisfunktionen für die 8-Punkt DCT.

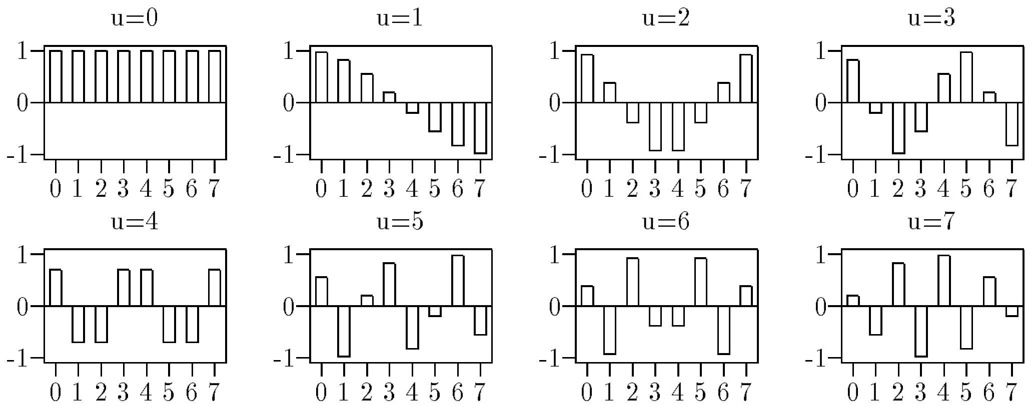


Abbildung 2.2: Basisfunktionen der eindimensionalen 8-Punkt DCT

Man beachte die Eigenschaft, daß die DCT einer konstante Funktionen $f(x) = a$ nur bei $u = 0$ von 0 verschieden ist:

$$
\begin{aligned}
F(0) &= \frac{1}{2\sqrt{2}} \sum_{x=0}^{7} a\cos((2x+1)u\pi/16) \\
&= \frac{1}{2\sqrt{2}} 8a\cos 0 \\
&= \frac{4}{\sqrt{2}} a
\end{aligned}
$$

Es sei $k \in \mathbb{N}_0$ das größte k so daß: $u = 2^k \cdot l, l \in \mathbb{N}$. Dann gilt:

$$
\begin{aligned}
F(u > 0) &= \frac{a}{2} \sum_{x=0}^{7} \cos((2x+1)u\pi/16) \\
&= \frac{a}{2} \sum_{m=0}^{2^k-1} \sum_{x=0}^{(8/2^k)-1} \cos((2x+1+2\cdot\frac{8m}{2^k})u\pi/16) = \frac{a}{2} \sum_{m=0}^{2^k-1} \sum_{x=0}^{(8/2^k)-1} \cos((2x+1)u\pi/16 + ml\pi) \\
&= \frac{a}{2} \sum_{m=0}^{2^k-1} \sum_{x=0}^{(4/2^k)-1} \cos((2x+1)u\pi/16 + ml\pi) + \cos((2x+1+2\frac{4}{2^k})u\pi/16 + ml\pi) \\
&= \frac{a}{2} \sum_{m=0}^{2^k-1} \sum_{x=0}^{(4/2^k)-1} \cos((2x+1)u\pi/16 + ml\pi) + \cos((2x+1)u\pi/16 + \frac{1}{2}l\pi + ml\pi) \\
&= \frac{a}{2} \sum_{m=0}^{2^k-1} \sum_{x=0}^{(4/2^k)-1} \cos((2x+1)u\pi/16 + ml\pi) + \cos((\frac{8}{2^k} - (2x+1))u\pi/16 + \frac{1}{2}l\pi + ml\pi) \\
&= \frac{a}{2} \sum_{m=0}^{2^k-1} \sum_{x=0}^{(4/2^k)-1} \cos((2x+1)u\pi/16 + ml\pi) + \cos(-(2x+1)u\pi/16 + \underbrace{l\pi}_{l \text{ ist ungerade, s.o.}} + ml\pi) \\
&= 0
\end{aligned}
$$

Der zweidimensionale Fall

Für die Bildverarbeitung erweitert man die eindimensionale DCT zur zweidimensionalen DCT:

$$F(u,v) = \frac{C(u)}{2}\frac{C(v)}{2}\sum_{y=0}^{7}\sum_{x=0}^{7} f(x,y)\cos((2x+1)u\pi/16)\cos((2y+1)v\pi/16)$$

Die IDCT im zweidimensionalen Fall lautet dementsprechend:

$$f(x,y) = \sum_{u=0}^{7}\sum_{v=0}^{7}\frac{C(u)}{2}\frac{C(v)}{2}F(u,v)\cos((2x+1)u\pi/16)\cos((2y+1)v\pi/16)$$

Abbildung 2.3 zeigt die Basisfunktionen für die 8×8-Punkt DCT. Diese Transformation eignet sich

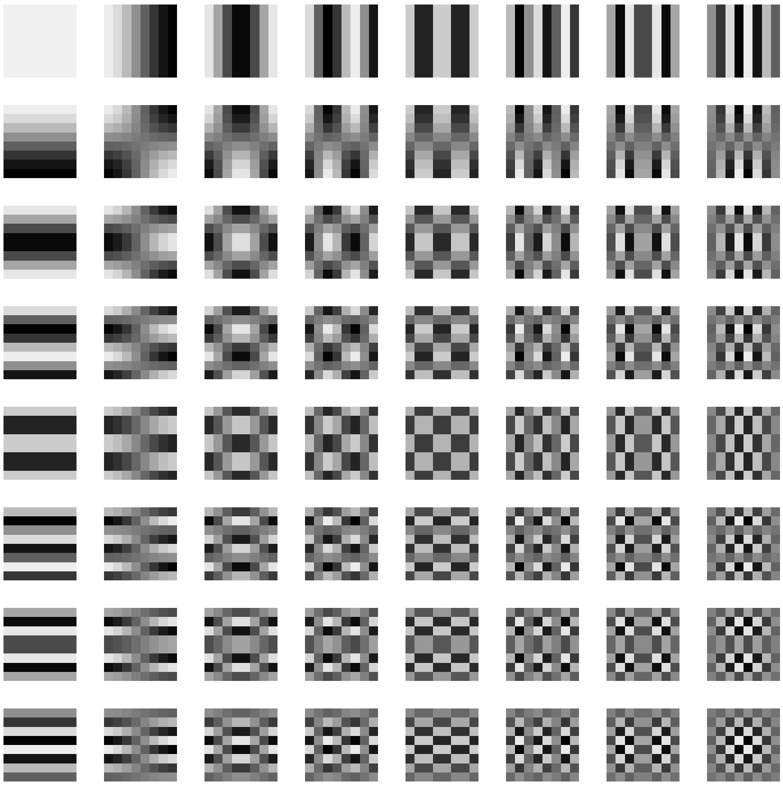
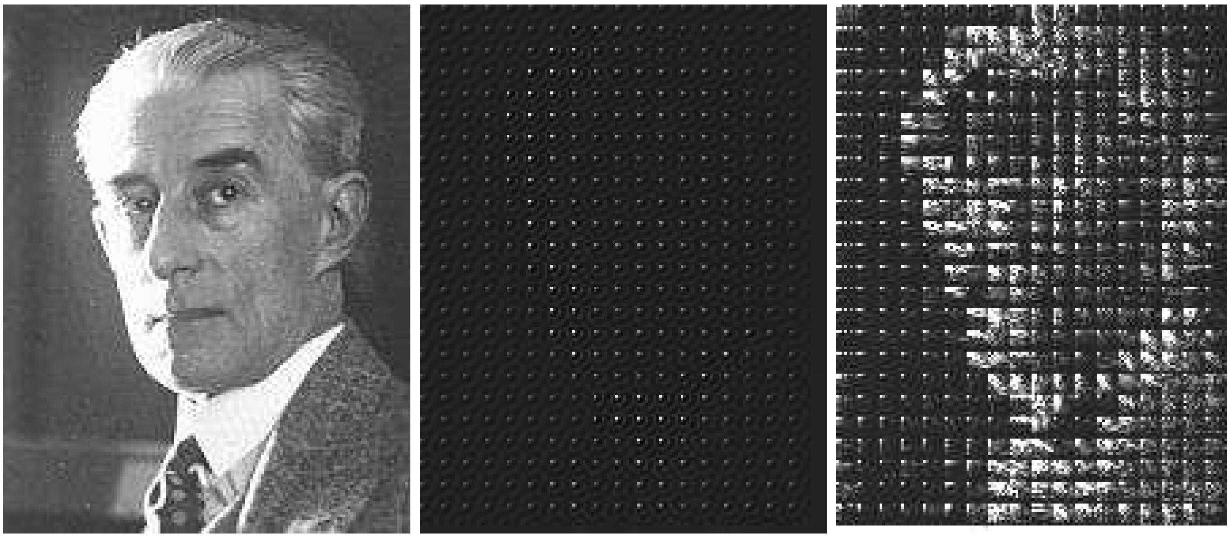


Abbildung 2.3: Basisfunktionen der zweidimensionalen 8×8-Punkt DCT

(im 2-dimensionalen Fall) sehr gut für Videobilder, da diese Bilder sehr große Flächen enthalten, die sich in der Helligkeit (und Farbe) nur wenig ändern. Durch die DCT wird dieser gesamte Gleichanteil auf den Koeffizienten $F(0,0)$ reduziert. In Abbildung 2.4 ist ein Bild zusammen mit seinen DCT-Koeffizienten dargestellt. Um kleinere Koeffizienten hervorzuheben, sind in Abbildung 2.4(c) die logarithmierten Beträge der Koeffizienten dargestellt. Auch gleichmäßige Gradienten im Bild werden sehr gut durch $F(u,v)$ mit $u,v \leq 1$ angenähert.

Nach der DCT-Transformation folgt in der Regel ein Quantisierungsprozess. Dabei können höhere Frequenzen gröber quantisiert werden (s. Abschnitt 2.4.1 auf Seite 7). Da nach der Quantisierung kleine Koeffizienten zu 0 werden, wird die Koeffizientenmatrix in den Bereichen höherer



(a) Originalbild (b) DCT-Koeffizienten, linear (c) DCT-Koeff., logarithmisch

Abbildung 2.4: Originalbild und seine DCT-Transformation

Frequenzen fast nur Nullen enthalten. Ordnet man die Koeffizienten ungefähr nach aufsteigenden Frequenzen (siehe Abbildung 2.5), so läßt sich diese Folge sehr gut mit einer Lauflängencodierung komprimieren.

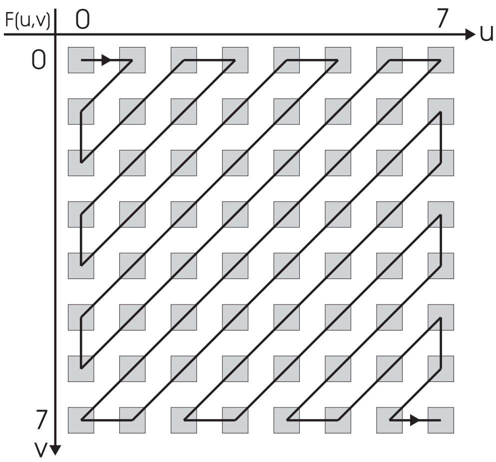


Abbildung 2.5: ZigZag-Ordnung von MPEG

schnelle Implementierungen

Wollte man die 2-dimensionale 8×8-DCT direkt gemäß obiger Formel implementieren, wären für eine Transformation unter Zuhilfenahme einer Tabelle zum Speichern der 8×8-Cosinus-Produkte 4096 Multiplikationen und 3136 Additionen nötig. Dieser Ansatz ist für eine effiziente Implementierung nicht geeignet. Aber es gibt einige Methoden, mit denen dieser Aufwand reduziert werden kann.

Die 2-dimensionale 8×8-DCT ist separierbar in 2 mal 8 eindimensionale 8-Punkt-DCTs. Dazu werden zuerst die Zeilen getrennt transformiert und danach die Spalten des Ergebnisses. Selbst wenn man die 1-dimensionale DCT direkt nach der Formel implementieren würde (wofür wir 64 Multiplikationen und 56 Additionen benötigen), reduziert sich damit der Aufwand auf 1024 Multiplikationen und 896 Additionen, was aber noch lange nicht das bestmögliche Verfahren ist.

Arai, Agui und Nakajima [AAN88] beschreiben einen Algorithmus zur Berechnung einer 1-dimensionalen 8-Punkt-DCT mit 13 Multiplikationen und 29 Additionen. Das schöne an diesem Algorithmus ist, daß 8 der Multiplikationen am Ende zur Skalierung des Ergebnisses benutzt werden. Dadurch kann die der DCT meistens folgende Quantisierungsstufe, welche noch einmal weitere Multiplikationen benötigen würden, mit der DCT kombiniert werden. Man braucht deshalb effektiv für die DCT nur 5 Multiplikationen.

Implementiert man die 2-dimensionale DCT durch Separierung in 2 mal 8 eindimensionale DCTs, dann kann man die Skalierungsmultiplikationen am Ende der ersten 8 eindimensionalen DCTs mit den Skalierungen der zweiten 8 1-dimensionalen DCTs und der Quantisierung zusammenfassen. Somit kommt man effektiv für die DCT auf $2 \cdot 8 \cdot 5 = 80$ Multiplikationen und $2 \cdot 8 \cdot 29 = 464$ Additionen.

Feig [FL90] hat durch einen direkten 2D-DCT-Ansatz die Operationen weiter auf 54 Multiplikationen, 462 Additionen und 6 arithmetische Shifts reduzieren können. Zwar benötigen Multiplikationen in den meisten Mikroprozessoren viel mehr Rechenzeit als eine Addition, aber in DSPs gibt es für gewöhnlich eine MULT-ADD-Anweisung, die eine Multiplikation und eine Addition zusammen ausführt. Speziell für diesen Prozessortyp haben Linzer und Feig [LF91] einen Algorithmus mit 416-MULT-ADD-Anweisungen entwickelt.

Eine genauere Beschreibung von schnellen DCT-Algorithmen findet sich in [PM93].

2.5.2 Bewegungskompensation

Viele Bildinhalte in einer Bildsequenz ändern sich nicht wesentlich von einem Bild zum nächsten. Aber diese Bildteile bewegen sich häufig, so daß der Inhalt nicht einfach vom letzten Bild an die selbe Stelle kopiert werden kann. Unterteilt man das Bild aber in kleine Bereiche und gibt für jeden dieser Bereiche den Vektor an, um den sich der Bereich von einem Bild zum nächsten bewegt hat, so hat man damit schon eine sehr gute Prädiktion des Folgebildes. Diese Motionvektoren sind für ein idealisiertes Beispiel in Abbildung 2.6 dargestellt, für ein reales Beispiel in Abbildung 2.7[4] .

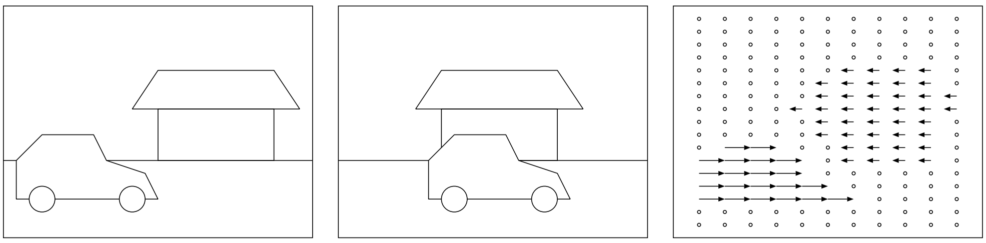


Abbildung 2.6: Prinzip von Motion Vektoren

Motionvektoren codiert man am besten differentiell zum letzten, da sich große Bildbereich meist gleichförmig bewegen (z.B. der gesamte Bildhintergrund bei einer Kamerafahrt). Bei MPEG wird das Konzept der Motionvektoren noch etwas erweitert. Dort kann ein Makroblock in B-Frames zwei Motionvektoren haben. Einer davon bezieht sich auf ein Bild in der Vergangenheit und ein

[4]Das Bild ist der Seite `http://wwwam.hhi.de/mpeg-video/papers/sikora/mpeg1_2/mpeg1_2.htm` entnommen

Abbildung 2.7: Motion-Vektoren in einer realen Videoszene

zweiter auf ein zukünftiges. Dadurch steht für die Codierung ein größerer Informationsvorrat zur Verfügung, der zur Codierung benutzt werden kann. Zur Codierung eines Makroblocks hat man dann die vier Möglichkeiten:

- Benutze die Bilddaten des vergangenen Bildes.
- Benutze die Bilddaten des zukünftigen Bildes.
- Benutze die Bilddaten des vergangenen und zukünftigen Bildes. Dabei wird der Mittelwert beider durch die Motionvektoren gegebenen Prädiktionen verwendet.
- Codiere es unabhängig von anderen Bildern.

Der durch die Motionvektoren bestimme Bildausschnitt wird in der Regel nicht unverändert in das Bild übernommen. Die Unterschiede zwischen Prädiktion und Realbild werden auch, ähnlich wie normale Bilddaten ohne Prädiktion, als DCT-Koeffizienten gespeichert, wobei jetzt noch mehr davon nach der Quantisierung 0 werden.

Meist verwendet man zur Suche von guten Motionvektoren die *mean absolute distortion* (MAD) Distanz. Sei das Originalbild durch $f(x, y)$ gegeben und das Bild, das als Prädiktion dienen soll, durch $g(x, y)$, so ergibt sich $MAD(i, j)$ an der Stelle (x, y) zu

$$MAD(x, y, i, j) = \frac{1}{256} \sum_{dx=0}^{15} \sum_{dy=0}^{15} |f(x + dx, y + dy) - g(x + dx + i, y + dy + i)|$$

Der Motionvektor $MV(x, y)$ ist dann das (i, j) Tupel, für das $MAD(x, y, i, j)$ minimal ist.

Der Ansatz, gute Motionvektoren durch eine vollständige Suche zu finden, ist sehr aufwendig. Es existieren eine ganze Reihe Algorithmen, die versuchen, in vertretbarer Zeit einen guten, aber nicht unbedingt den besten Motionvektor zu finden. Siehe dazu Kapitel 13 in [MPFL96].

Kapitel 3

MPEG Grundlagen

MPEG steht für *Motion Picture Expert Group* und bezeichnet eine Gruppe, die unter der Schirmherrschaft von ISO (International Organization for Standardization) und IEC (International Electrotechnical Commission) geführt wurde. Ziel dieser Gruppe war der Entwurf eines internationalen Standards zur digitalen Speicherung von Audio- und Videodaten auf CDROM bzw. zur Codierung mit Bitraten bis zu etwa $1,5\frac{MBit}{s}$. Dieser Standard bekam im November 1991 den Status eines Internationalen Standards und trägt heute die Bezeichnung ISO/IEC 11172 (oder geläufiger: MPEG-1). [MPFL96]

Als die Nachfrage nach einem Standard für die qualitativ hochwertige digitale Übertragung von Audio- und Videodaten für Rundfunkzwecke stieg, begann die Arbeit an dem Standard ISO/IEC 13818 (MPEG-2), denn MPEG-1 fehlte dazu die Unterstützung von Interlaced-Video (wie es heute im analogen Fernsehen benutzt wird) und war von vornherein auf niedrigere Bitraten und geringere Auflösungen ausgelegt. Für MPEG-2 wurden allerdings die grundlegenden Algorithmen von MPEG-1 beibehalten, so daß ein MPEG-2-Decoder (bis auf ein paar Feinheiten) einen MPEG-1 konformen Bitstream decodieren kann [HPN97].

Vorbemerkung Die Syntaxdiagramme zur Veranschaulichung der MPEG-Stream-Syntax sind teilweise stark vereinfacht. Einzelne Markerbits, die eine fälschliche Erkennung von Startcodes verhindern und Extension-Daten für benutzerdefinierte Daten oder für zukünftige Erweiterungen sind nicht eingetragen. Felder, die für Verwaltungsaufgaben des Decoderbuffers enthalten sind, sind nur bei Anwendungen mit konstanter Übertragungsbitrate oder fester Decoderbuffergröße notwendig und auch nicht in die Diagramme eingetragen.

3.1 Anatomie eines MPEG-System-Streams

Ein MPEG-System-Streams läßt sich in drei unabhängige syntaktische Teile zerteilen:

Video-Streams enthalten jeweils genau einen Videokanal ohne Ton.

Audio-Streams enthalten jeweils genau einen Audiokanal. Dabei werden bei einer Stereo-Quelle die beiden Stereo-Kanäle zusammen in einem MPEG-Audio-Stream codiert.

Der System-Stream dient zur Kombination von Video- und Audio-Streams und deren gegenseitigen Synchronisation. Ein System-Stream kann gleichzeitig 16 Video- und 32 Audiokanäle aufnehmen.

Video- und Audio-Streams werden als *elementary Streams* bezeichnet. Die meisten derzeit verfügbaren MPEG-Video- und -Audio-Dateien sind reine elementary Streams. Dies hat den Nachteil, daß darin keine genauen Timinginformationen enthalten sind, die für ein synchrones Abspielen von Video und Audio erforderlich sind.

3.1.1 Der System Target Decoder

MPEG definiert nicht, wie der komprimierte Bitstream zu erzeugen ist. Es wird nur definiert, wie ein Decoder den komprimierten Bitstream zu decodieren hat.

Dazu wird ein idealisiertes Modell eines Decoders eingeführt. Der *System Target Decoder* (s. Abb. 3.1) ist das idealisierte Modell eines Decoders für MPEG-System-Streams. Er besteht aus dem eigentlichen System-Decoder, den einzelnen Elementary-Stream-Decodern mit ihren Eingangs-Buffern und einer System-Uhr zur Synchronisation der Elementary-Stream-Decoder. Diese System-Uhr läuft mir einer Frequenz von 90000 Hz.[1]

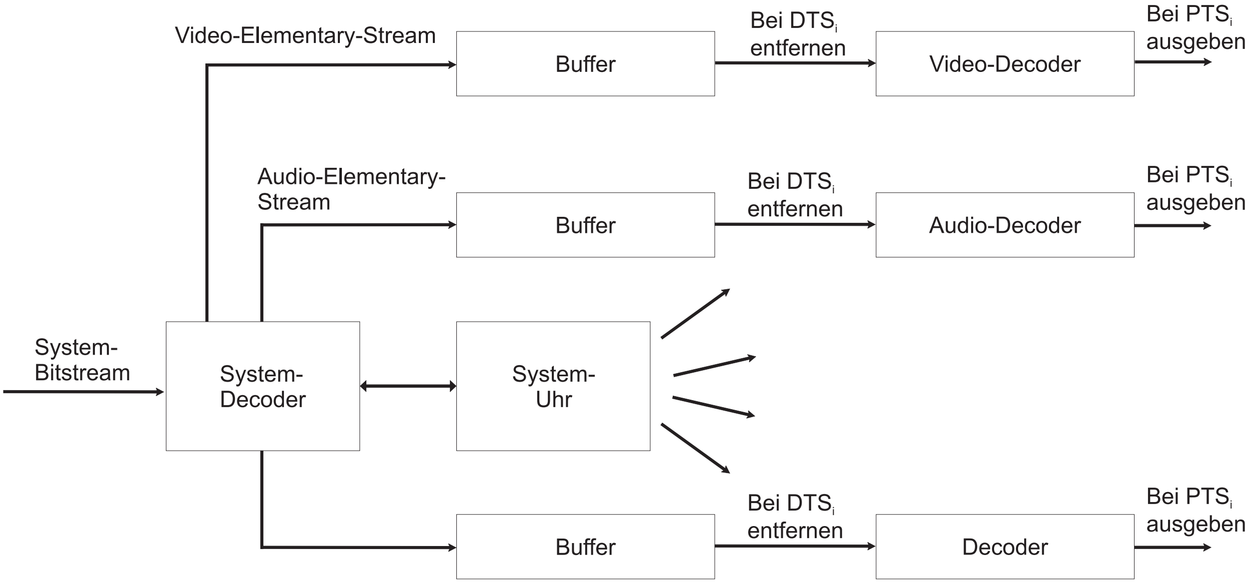


Abbildung 3.1: System Target Decoder

Der System-Stream kommt mit einer konstanten Bitrate in den System-Decoder. Mit den Informationen der System-Schicht des Streams teilt der System-Decoder diesen Stream in die Elementary-Streams auf, die er dann zu den jeweiligen Elementary-Stream-Decodern schickt.

Im System-Stream sind *Decoding Time Stamp*s (DTS) und *Presentation Time Stamp*s (PTS) codiert. Die DTS-Zeiten geben an, wann der Elementary-Stream-Decoder eine *Access Unit*[2] aus seinem Buffer entnehmen soll. Die PTS-Zeit gibt dahingegen an, wann die Access Unit darzustellen ist. Obwohl die Decodierzeiten zu 0 angenommen werden, können DTS und PTS voneinander abweichen, da zum Beispiel für die Videodecodierung zeitlich später folgende Bilder (P-Frames)

[1]90000 Hz ist geradzahliges Vielfaches der heute üblichen Bildwiederholraten 25 Hz und 30 Hz, was die Implementation von Hardware-Decodern besonders einfach macht.

[2]Eine Access Unit bezeichnet die Daten eines Elementary Streams, die jeweils zusammen aus dem Buffer entnommen und decodiert werden. In Video-Streams werden jeweils die Daten, die ab einem PictureStartCode beginnen, als Access Unit bezeichnet.

teilweise für die Decodierung zeitlich davor liegender Bilder (B-Frames) benötigt werden. Diese P-Frames müssen deshalb vor die B-Frames verschoben werden und haben somit einen frühere DTS als PTS.

3.1.2 Syntax des System-Streams

Startcodes

Syntaktische Elemente beginnen bei MPEG immer mit einem Startcode, der mit der Bitsequenz `0x000001` beginnt. Daran schließt sich ein Byte an, das den Typ des Startcodes angibt (siehe Tabelle 3.1). Diese Startcodes bieten einem Decoder auch während einer kontinuierlichen Übertragung oder nach Übertragungsfehlern Synchronisationspunkte, an denen die Decodierung (neu) beginnen kann. Es ist gewährleistet, daß in den codierten Daten niemals versehentlich Sequenzen auftreten, die als Startcode mißinterpretiert werden könnten.

Auch im Video-Stream werden Startcodes verwendet. Diese sind aber für den System-Decoder ohne Bedeutung.

System-Codes		Video-Codes	
B9	iso11172 end code	00	picture start code
BA	pack start code	01	slice 1 start code
BB	system header start code	...	...
C0	audio stream 0	AF	slice 175 start code
...	...	B3	sequence header code
DF	audio stream 31	B7	sequence end code
E0	video stream 0	B8	group start code
...	...		
EF	video stream 15		

Tabelle 3.1: StartCode-IDs (Auszug)

Packs und Packets

Ein System-Stream besteht aus einer Folge von *Packs*. Jeder Pack enthält wiederum eine Folge von Packets (siehe Abbildung 3.2 auf der nächsten Seite). Die Pack-Ebene ist für uns hier nicht weiter interessant.

In den Packets befindet sich optional ein PTS, ein PTS und DTS oder keins von beiden. Schließlich folgt eine Anzahl von Datenbytes, die direkt in den Buffer des jeweiligen Elementary-Stream-Decoders übertragen wird. Zu welchem Decoder die Daten geschickt werden, ist aus der *StreamID* (dem StartCode des Packets) ersichtlich. Die PTSs und DTSs beziehen sich jeweils auf das erste Byte einer Access Unit.

3.2 Aufbau eines Video-Streams

Abbildung 3.3 auf Seite 17 zeigt in einer TopDown-Darstellung, welche Daten in MPEG zu Syntaxeinheiten zusammengefaßt werden. Ausgehend von der ganzen Videosequenz werden jeweils einige aufeinanderfolgende Frames als *Group of Pictures* (GOP) zusammengefaßt. Einzelne Frames dieser Group of Pictures werden einfach *Pictures* genannt. Ein Picture besteht dann wiederum aus

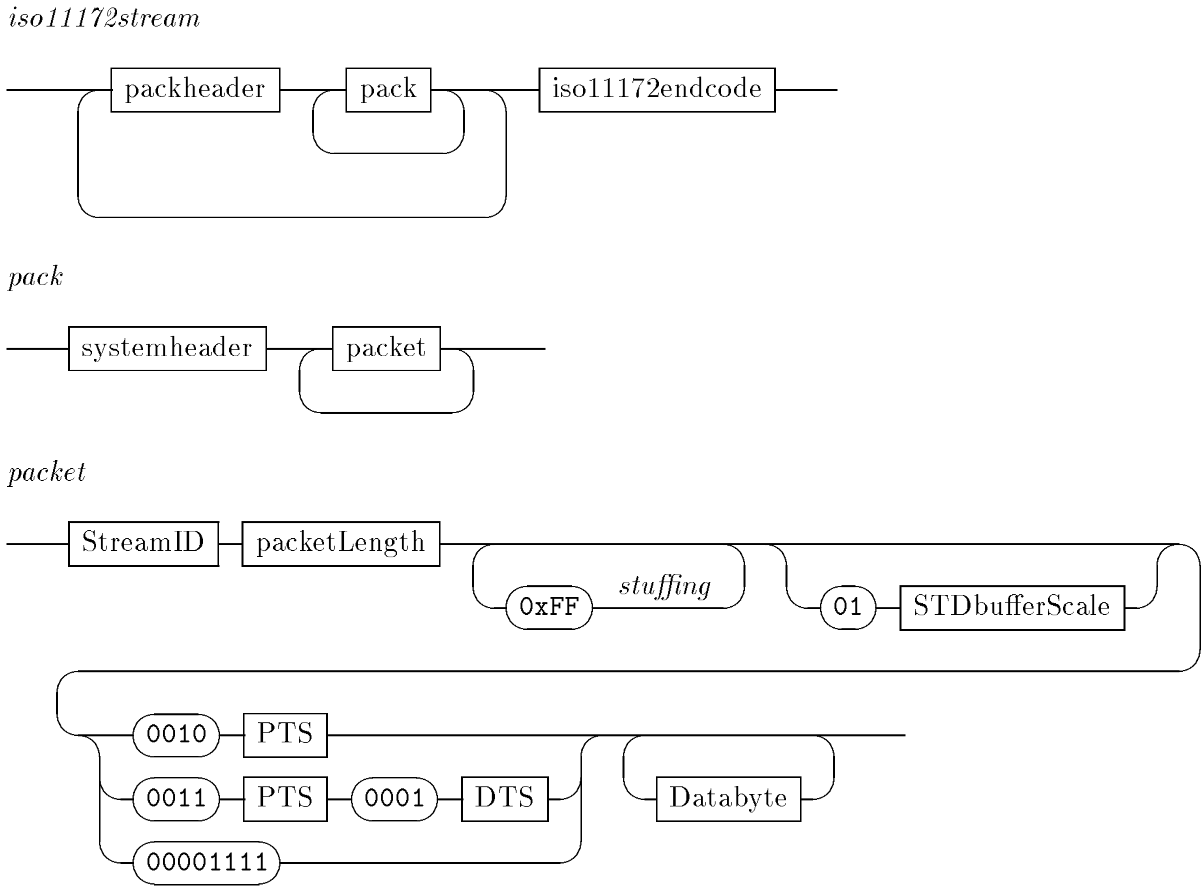


Abbildung 3.2: Syntax der System-Streams

einem oder mehreren *Slices*, deren Höhe jeweils ein ganzzahliges Vielfaches von 16 Pixel ist. Jede Slice besteht aus einer Menge von *Makroblöcken*, die jeweils die Größe 16x16 Pixel haben. Wie der Makroblock nun codiert wird, ist abhängig vom Bildtyp und Bildinhalt.

Als Eingabe dienen bei MPEG-Video ausschließlich Farbbilder im YCbCr-Farbraum. Dabei wird für die Luminanz (Y) eine doppelt so hohe horizontale und vertikale Auflösung verwendet wie für die Chrominanz-Kanäle (Cb und Cr).

3.2.1 Makroblöcke

Grundlage der Videocodierung ist bei MPEG die DCT-Transformation. Dazu werden die drei Farbkanäle des Bildes in Blöcke zu 8x8 Pixeln unterteilt, auf welche die DCT-Transformation angewendet wird. Jeweils 6 solcher Blöcke werden zu einem *Makroblock* zusammengefaßt (4 Y-Blöcke und jeweils ein Cb- und Cr-Block, siehe Abbildung 3.4 auf Seite 18).

Die DCT-Koeffizienten werden danach mit einer (vordefinierten, aber auch in jedem *Sequence-Header* umdefinierbaren) Quantisierungsmatrix (der *Intra-Matrix*, siehe Abbildung 3.5(a)) quantisiert. Diese Quantisierungsmatrix kann in einem Makroblock um einen frei codierten Faktor verändert werden, was aber zusätzliche Bits benötigt. Die Intra-Matrix basiert auf der Quantisierungsmatrix, die vom JPEG-Komitee entwickelt wurde und diese beruht wiederum auf einer Reihe von Experimenten. Die Erfahrung hat gezeigt, daß sie auch bei der Codierung von Videosequenzen

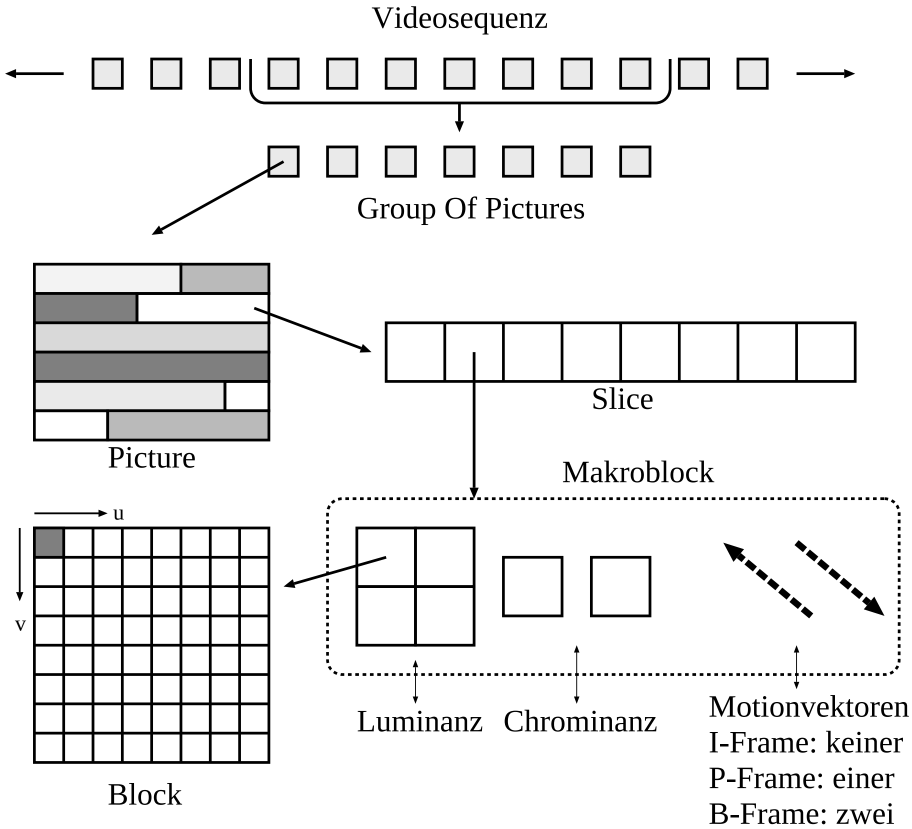


Abbildung 3.3: Hierarchie der MPEG-Codierung

gute Ergebnisse liefert.

Nachdem die Koeffizienten gemäß einem ZigZag-Schema linear angeordnet wurden (siehe Abbildung 2.5 auf Seite 10), werden die Koeffizienten huffmancodiert. Dabei wird jedesmal der Wert eines Koeffizienten und die Anzahl der danach folgenden 0-Werte zu einem Codewort zusammengefaßt, da diese stark miteinander korrelieren. Eine Ausnahme bildet der DC-Koeffizient $F(0,0)$, der differenziell zum letzten DC-Koeffizienten codiert wird. Nur der erste DC-Koeffizient in einer Slice wird absolut codiert.

3.2.2 Motion-Vektoren

Um die reichlich vorhandenen Korrelationen zwischen zeitlich aufeinanderfolgenden Bildern auszunutzen, durchsucht der Encoder Bezugsbilder nach Bereichen, die sehr „ähnlich" zu dem zu codierenden Makroblock sind. Die Verschiebungen zwischen diesen Bildbereichen wird in dem Makroblock als Motion-Vektoren gespeichert. Die Differenz aus Originalbild und Prädiktion wird wieder mittels DCT transformiert und (mit einer flachen Quantisierungsmatrix, *NonIntra-Matrix* genannt, siehe Abbildung 3.5(b)) quantisiert. Motion-Vektoren werden differentiell zum Motion-Vektor des letzten Makroblocks codiert, da sich häufig große Bereiche des Bildes gleichförmig bewegen (z.B. Verschiebung des gesamten Bildhintergrunds). Zur Codierung werden Huffmancodes benutzt, wobei kleine Richtungsänderungen kürzere Codes bekommen.

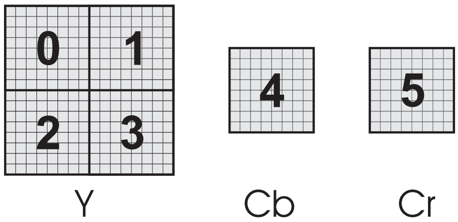


Abbildung 3.4: Ein Makroblock besteht aus 4 Y- und je einem Cb- und Cr-Block.

$$\begin{pmatrix} 8 & 16 & 19 & 22 & 26 & 27 & 29 & 34 \\ 16 & 16 & 22 & 24 & 27 & 29 & 34 & 37 \\ 19 & 22 & 26 & 27 & 29 & 34 & 34 & 38 \\ 22 & 22 & 26 & 27 & 29 & 34 & 37 & 40 \\ 22 & 26 & 27 & 29 & 32 & 35 & 40 & 48 \\ 26 & 27 & 29 & 32 & 35 & 40 & 48 & 58 \\ 26 & 27 & 29 & 34 & 38 & 46 & 56 & 69 \\ 27 & 29 & 35 & 38 & 46 & 56 & 69 & 83 \end{pmatrix}$$

(a) Intra-Matrix

$$\begin{pmatrix} 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \end{pmatrix}$$

(b) Nonintra-Matrix

Abbildung 3.5: Standard Quantisierungsmatrizen

3.2.3 Slices

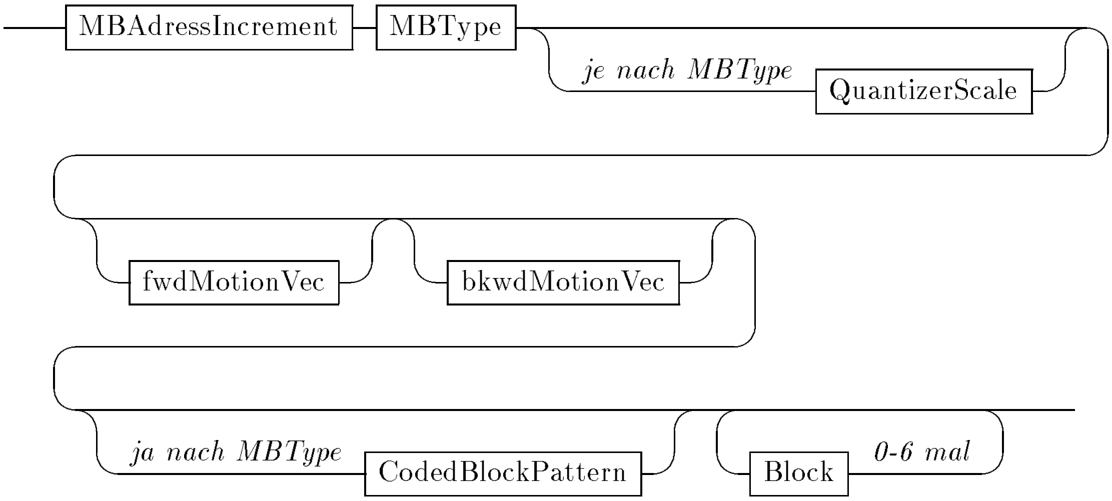
Eine Folge von Makroblöcken eines Bildes wird in *Slices* zusammengefaßt. Da Slices (im Gegensatz zu einzelnen Makroblöcken) mit einem Startcode beginnen und unabhängig von anderen Slices decodiert werden können, kann man die Fehlerrobustheit erhöhen, wenn man die Anzahl der Slices erhöht. Allerdings steigt damit auch der Verwaltungs-Overhead. Slices können in MPEG-1 an beliebiger Stelle beginnen und enden (solange sie sich nicht überlappen) und müssen das ganze Bild abdecken. In MPEG-2 dürfen sich Slices nicht über mehrere Makroblock-Zeilen erstrecken. Dafür ist es in bestimmten MPEG-2-Profiles erlaubt, daß das Bild nicht vollständig mit Slices bedeckt ist. Der unbedeckte Bereich wird unverändert vom letzten Bild übernommen.

In einer Slice kann die Quantisierungsmatrix um einen `QuantizerScale` Faktor verändert werden. Diese Position eignet sich besser zur Realisierung von Adaptiver Quantisierung, da es hier im Gegensatz zur Änderung im Makroblock keinen Overhead erzeugt.

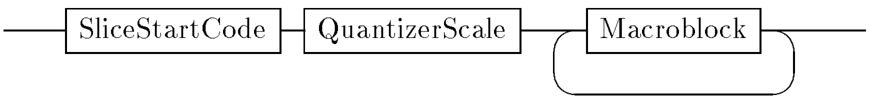
3.2.4 Bildtypen

Effiziente Videocodierung erfordert, daß Korrelationen zwischen zeitlich folgenden Bildern ausgenutzt werden, da große Teile des Bildes oft gleich bleiben und somit einen guten Ansatzpunkt für die Kompression bieten. Andererseits will man meist auch wahlfreien Zugriff auf Sequenzen im Video, ohne das ganze davorliegende Video dekomprimieren zu müssen. Auch wenn man sich in laufende Übertragungen einschalten will oder Übertragungsfehler möglich sind, werden Wiederaufsetzpunkte innerhalb des Videos benötigt, an denen die Decodierung neu beginnen kann. MPEG definiert vier verschiedene Arten der Codierung eines Bildes in einer Videosequenz:

Macroblock



Slice



Intra coded (I) Frames werden unabhängig von anderen Bildern codiert und können wieder decodiert werden ohne auf Informationen aus anderen Bildern zurückgreifen zu müssen.

Predictive coded (P) Frames können zusätzlich Bezug auf das zeitlich vorhergehende I- bzw. P-Bild nehmen. Gleiche Bildinhalte können (ggf. verschoben) in das zu codierende Bild übernommen werden.

Bidirectionally predictive coded (B) Frames können zusätzlich zu den Möglichkeiten von P-Frames auch noch auf das zeitlich *folgende* I- bzw. P-Frame zurückgreifen. B-Bilder werden niemals für die Rekonstruktion anderer Bilder verwendet.[3]

DC-coded (D) Frames sind qualitativ sehr minderwertige Bilder, die unabhängig von anderen decodiert werden können.[4] Sie sind nur für die Benutzung während Trickmodes (Schnellvorlauf, ...) gedacht, denn beim Schnellvorlauf entsteht das Problem, daß der Decoder die Daten viel schneller decodieren muß. Selbst wenn er nur die I-Frames decodieren würde (die sich ja unabhängig decodieren lassen), müßte er für höhere Datenraten ausgelegt werden, da I-Frames das größte Datenvolumen besitzen. Von der Verwendung von D-Frames wird im MPEG-2-Standard aber abgeraten und sie sollen hier auch nicht weiter beachtet werden.

Ein I-Frame hat typischerweise das größte Datenvolumen, das B-Frame das geringste. Eine typische Videosequenz besteht meistens aus der Wiederholung folgenden Musters: Ausgehend von einem I-Frame wird eine Kette von P-Frames abgeleitet. Zwischen jedem P-Frame-Paar bzw. dem I/P-Frame-Paar am Beginn und zwischen dem letzten P-Frame und dem I-Frame am Anfang des

[3] Mit Ausnahme des Temporal-Scalability-Modes in MPEG-2

[4] In D-Frames wird nur die DC-Komponente eines 8×8-DCT-Blocks (also der Gleichanteil) codiert.

folgenden Musters befinden sich nochmals einige B-Frames. Dies ist in Abbildung 3.6 dargestellt. Die Pfeile unter den Frames geben die Abhängigkeiten zwischen den Frames an.

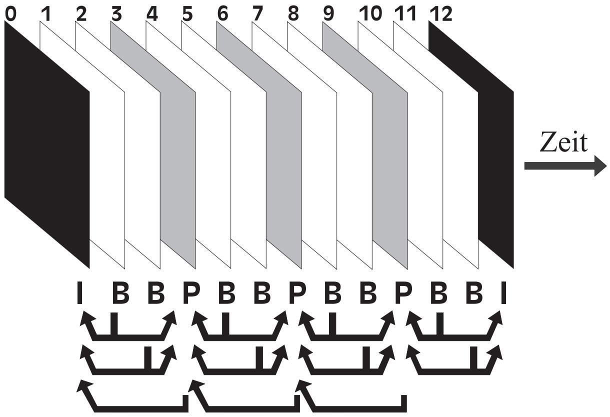


Abbildung 3.6: Videosequenz in Darstellungsreihenfolge

Video-Sequenzen mit B-Bildern können nicht mehr in der richtigen zeitlichen Reihenfolge in dem Stream codiert werden, da sie nicht decodiert werden können, bevor das folgende I- oder P-Bild decodiert wurde. Deshalb werden die I- bzw. P-Bilder vor den B-Bildern gesendet, wie in Abbildung 3.7 dargestellt ist. Diese Ordnung bezeichnet man als Stream-Ordnung (Stream-Order). Man sieht, daß sämtliche Abhängigkeiten zwischen den Bildern sich nun nur auf früher gesendete

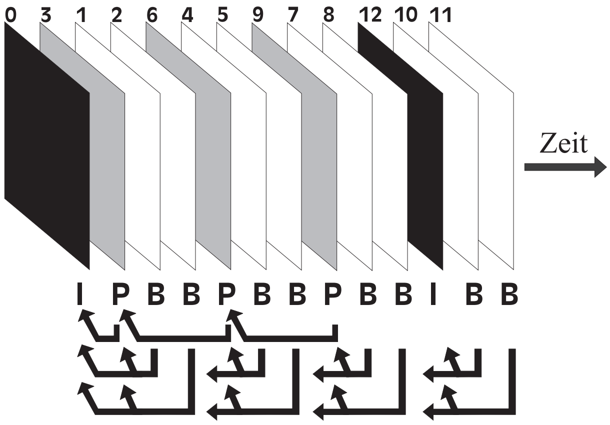


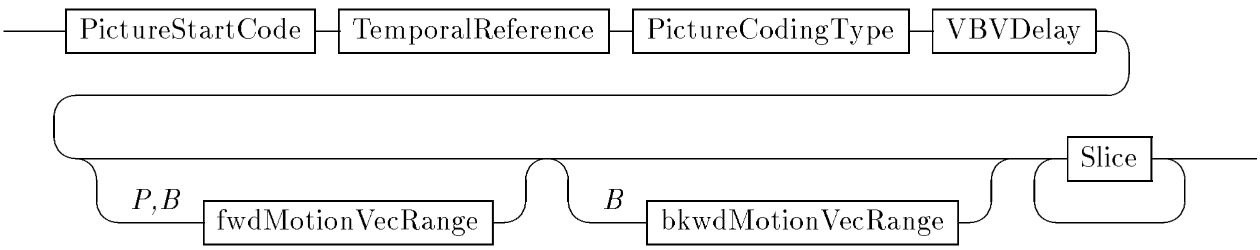
Abbildung 3.7: Videosequenz in Übertragungsreihenfolge

Bilder beziehen und somit jedes Bild sofort decodiert werden kann. Der Decoder muß nun aber jeweils die beiden zuletzt decodierten I/P-Frames zwischenspeichern, um die B-Frames decodieren zu können. Die Ausgabe der P-Frames muß bis hinter die B-Frames verzögert werden.

3.2.5 Group of Pictures

Die Grundidee, als die *Group of Pictures* (GOP) eingeführt wurde, war, eine Sequenz fortlaufender Bilder zusammenzufassen, die unabhängig von Bildern aus anderen GOPs decodiert werden können.

Picture



Im endgültigen Standard ist diese Eigenschaft aber nicht mehr vorhanden. Es gelten nun hingegen die folgenden abgeschwächten Eigenschaften:

1. Eine GOP beginnt in Stream-Ordnung immer mit einem I-Frame.
2. Eine GOP beginnt in Darstellungs-Ordnung immer mit einem I- oder B-Frame und endet mit einem I- oder P-Frame.

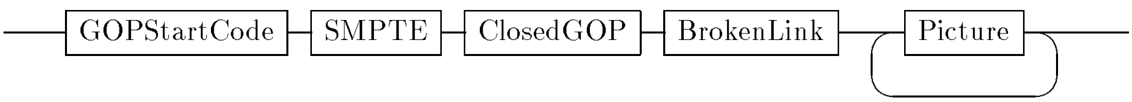
Da sich vor dem ersten I-Frame in Darstellungs-Ordnung einige B-Frames befinden können, die Informationen aus der letzten GOP benötigen, ist eine solche GOP nicht mehr unabhängig decodierbar, es sei denn, in den B-Frames am Beginn wird nur *rückwärts-Bewegungskompensation* (vom ersten I-Frame der GOP) benutzt.

Für den Videoschnitt (der sehr einfach an GOP-Grenzen durchgeführt werden kann) existieren noch zwei Flags im Header: An dem Flag *ClosedGOP* kann man erkennen, ob alle Bilder der GOP unabhängig von anderen GOPs decodiert werden können (siehe Tabelle 3.2). Das Flag *BrokenLink* wird hingegen gesetzt, wenn beim Schnitt die vorhergehende GOP entfernt wurde. Ein Decoder darf dann die ersten B-Frames nicht decodieren, da die Motion-Vektoren sich auf falsches Bildmaterial beziehen würden.

		I	B	B	P	B	B	P	B	B	P	Bsp. für eine *closed GOP*
B	B	I	B	B	P	B	B	P	B	B	P	Bsp. für eine *open GOP*

Tabelle 3.2: closed und open GOPs

GroupOfPictures



3.2.6 Syntax des Videostreams

Die oberste Hierarchieebene bildet die *VideoSequence*, deren (vereinfachte) Syntax in Abbildung 3.8 angegeben ist. Im *SequenceHeader* sind die globalen Eigenschaften des Videos enthalten. Dazu zählen die Bildauflösung und Framerate. Soll es möglich sein, daß die Decoder während einer

Übertragung mit der Decodierung beginnen können, so muß der SequenceHeader des öfteren wiederholt werden, da ohne den SequenceHeader eine Decodierung nicht möglich ist. Dabei muß in jedem SequenceHeader mit Ausnahme der Quantisierungsmatrizen, die gleichen Daten enthalten sein. Diese Quantisierungsmatrizen können benutzt werden, um die Quantisierung an die jeweilige Bildszene anzupassen. Sie ersetzen in diesem Fall die Standardmatrizen für Intra-codierte Makroblöcke.

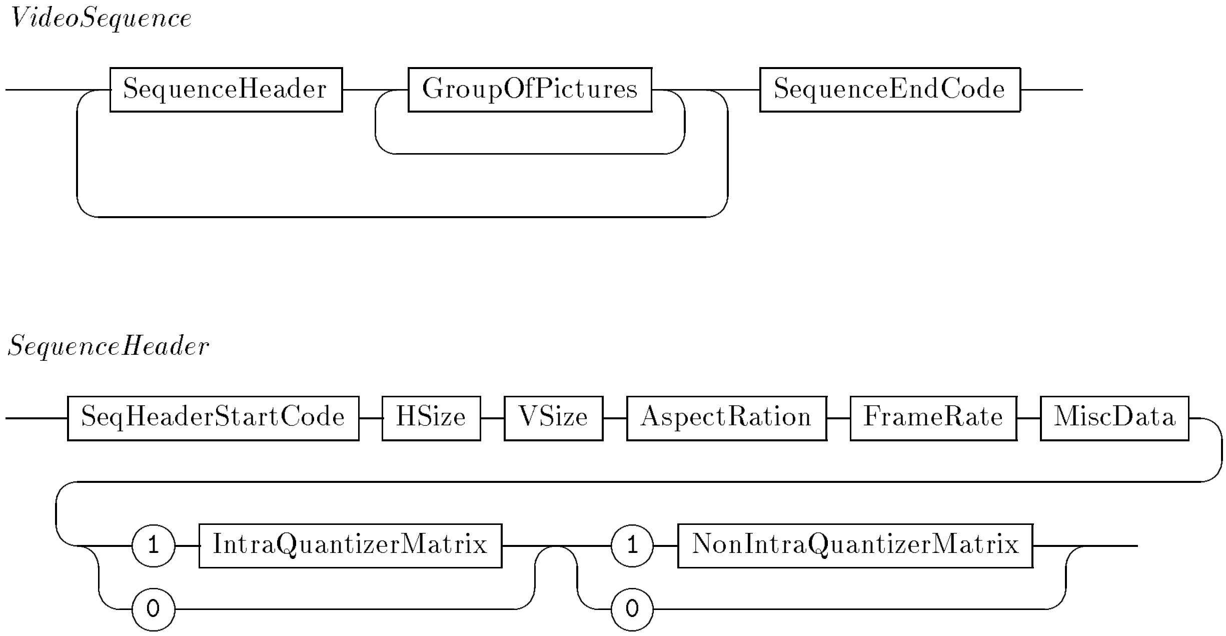


Abbildung 3.8: TopLevel-Syntax einer Videosequenz

3.3 Aufbau eines MPEG-Video-Decoders

Den prinzipiellen Aufbau des Video-Decoders zeigt Abbildung 3.9. Die Eingabedaten kommen mit konstanter Bitrate in den Eingabebuffer, von dem sie blockweise (jeweils eine Access-Unit = 1 Frame) entfernt werden. Es folgt eine Huffman- und Lauflängendecodierung um die quantisierte DCT-Matrix zu erhalten. Die Koeffizienten werden mit ihrem Quantisierungsfaktor aus der Intra-Quantizer-Matrix multipliziert und daraufhin wird die Inverse-DCT-Transformation angewendet. Das fertige Bild wird nun je nach Frametyp weiterverarbeitet. Ist es ein B-Frame, so wird es nur dargestellt. Ansonsten wandert es in den Bildspeicher „*zukünftiges Bild*" und das Bild, das darin gespeichert war, in „*vorheriges Bild*". Diese beiden Bildspeicher werden für die Decodierung der Motionvektoren von B- und P-Frames benötigt.

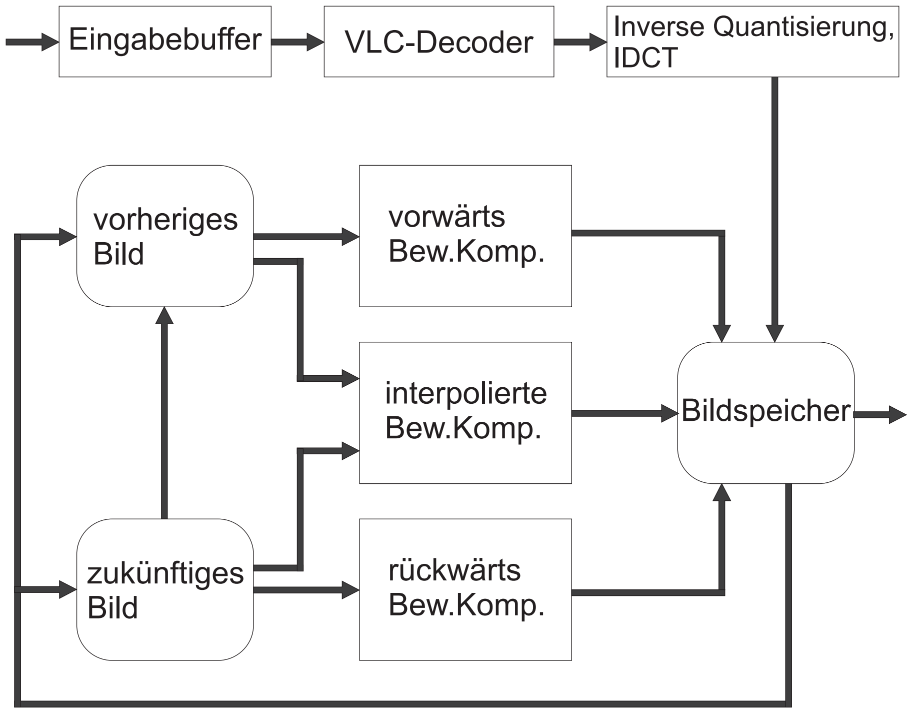


Abbildung 3.9: Prinzipaufbau eines MPEG-Video-Decodes

3.4 Typische Größen von I,P,B-Frames

```
Width: 176      Height: 112
Avg. Frame Size: 417 bytes + 0 bits

Factor of quantization [scales quantization matrix]: 30
Total Compression Rate:  0.71 % of uncompressed 24 bit images
                       =  0.17 bits per pixel

Number of Macroblocks [width * height = sum]: 11 x 7 = 77 per frame

Frame specific information:

59  I FRAMES, average is:
        Size: 2387 bytes + 1 bits
        Compression Rate:  4.04%
        Time to Decode: 0.013713 secs.

241 P FRAMES, average is:
        Size: 917 bytes + 7 bits
        Compression Rate:  1.55%
        Time to Decode: 0.009459 secs.

598 B FRAMES, average is:
        Size: 20 bytes + 6 bits
        Compression Rate:  0.03%
        Time to Decode: 0.001923 secs.
```

Man sieht, daß die Bildgrößen der einzelnen Bildtypen sehr stark voneinander abweichen. Bei diesem MPEG-Stream beträgt das Verhältnis von I:P:B-Frames etwa 120:46:1.

```
Width: 192      Height: 144
Avg. Frame Size: 7593 bytes + 2 bits

Factor of quantization [scales quantization matrix]: 5
Total Compression Rate:  9.15 % of uncompressed 24 bit images
                       =  2.20 bits per pixel

Number of Macroblocks [width * height = sum]: 12 x 9 = 108 per frame

Length of vectors in pixels:

        Horizontal forward vectors, maximum :  23       average:   3
        Vertical forward vectors, maximum   :  23       average:   3

        Horizontal backward vectors, maximum:  15       average:   3
        Vertical backward vectors, maximum  :  15       average:   2
```

```
Frame specific information:

781 I FRAMES, average is:
        Size: 13351 bytes + 5 bits
        Compression Rate: 16.10%
        Time to Decode: 0.038320 secs.

2341 P FRAMES, average is:
        Size: 7708 bytes + 4 bits
        Compression Rate:  9.29%
        Time to Decode: 0.033373 secs.

6240 B FRAMES, average is:
        Size: 6829 bytes + 2 bits
        Compression Rate:  8.23%
        67.05% interpolated Macro Blocks
        Time to Decode: 0.035602 secs.
```

Hier beträgt das Verhältnis I:P:B nur etwa 2:1,13:1.

3.5 Scalability-Modes

In MPEG-2 wurden sogenannte *Scalability-Modes* definiert, die es erlauben, Video in mehreren verschiedenen Qualitäten zu senden, ohne viel mehr Bandbreite als zum alleinigen Senden der höchsten Qualitätsstufe zu brauchen. Dabei wird der Videostream in bis zu drei Layers aufgeteilt, die als *Base*, *Enhancement* und *Higher* bezeichnet werden. Ein Video in geringer Qualität kann empfangen werden, wenn allein der *Base*-Layer decodiert wird. Die beiden anderen Layers bauen jeweils auf den darunterliegenden Layers auf und verbessern die Qualität.

Da auch wir bei der Videoübertragung mit unterschiedlichen Netzwerkbandbreiten zu leben haben, die sich auch noch im Laufe der Zeit ändern können, bietet es sich an, den Videostream ebenfalls in mehrere Layers aufzuteilen. Die Daten eines jeweils höheren Layers werden erst dann übertragen, wenn der Layer unter diesem übertragen worden ist. Somit ist es möglich, daß die Bildqualität immer optimal der Netzwerkkapazität entspricht, ohne daß das Video bei der Übertragung umcodiert werden muß. Wird auf der Decoderseite ein MPEG-2 Decoder verwendet, ist der Aufwand dabei relativ gering.

In MPEG-2 werden 4 Arten von Scalability-Modes definiert, die auch miteinander kombiniert werden können. Diese sollen nun im einzelnen erläutert werden.

3.5.1 Data-Partitioning

Data-Partitioning ist die einfachste Scalability-Variante. Bei ihr wird ein Bitstream so in zwei getrennte aufgeteilt, daß der erste die gesamte Kontrollinformationen und die niederfrequenten DCT-Koeffizienten enthält. Der zweite enthält zur Synchronisation mit der Base-Layer ebenfalls die Kontrollinformationen, aber nur die höherfrequenten DCT-Koeffizienten. Der genaue Punkt, an dem die Koeffizienten aufgeteilt werden, ist variabel.

Die Umcodierung eines Bitstreams in zwei mit Data-Partitioning ist recht einfach möglich. Dazu muß der Bitstream nur bis inclusive der Huffmancodes decodiert und dann wieder neu Huffman-codiert werden. Es sind keine mathematischen Transformationen oder Suchen notwendig. Werden beide Layers decodiert, ist gegenüber dem ursprünglichen Bitstream kein Qualitätsverlust vorhanden. Wird aber im Regelfall nur die Base-Layer decodiert, so erreicht man mit den anderen Scalability-Modes bessere Ergebnisse.

3.5.2 SNR-Scalability

SNR-Scalability hat eine gewisse Ähnlichkeit mit Data-Partitioning. Auch hier werden alleinig die DCT-Koeffizienten verändert. Anstatt höherfrequente Koeffizienten einfach aus dem Base-Layer zu entfernen, werden hier jedoch alle Koeffizienten lediglich stärker quantisiert und die Differenz dann im höheren Layer übertragen. Dadurch ist die Bildqualität des Base-Layers besser als bei Data-Partitioning, da die Schärfe nicht so stark abnimmt, sondern nur das Rauschen entlang von Kanten verstärkt wird.

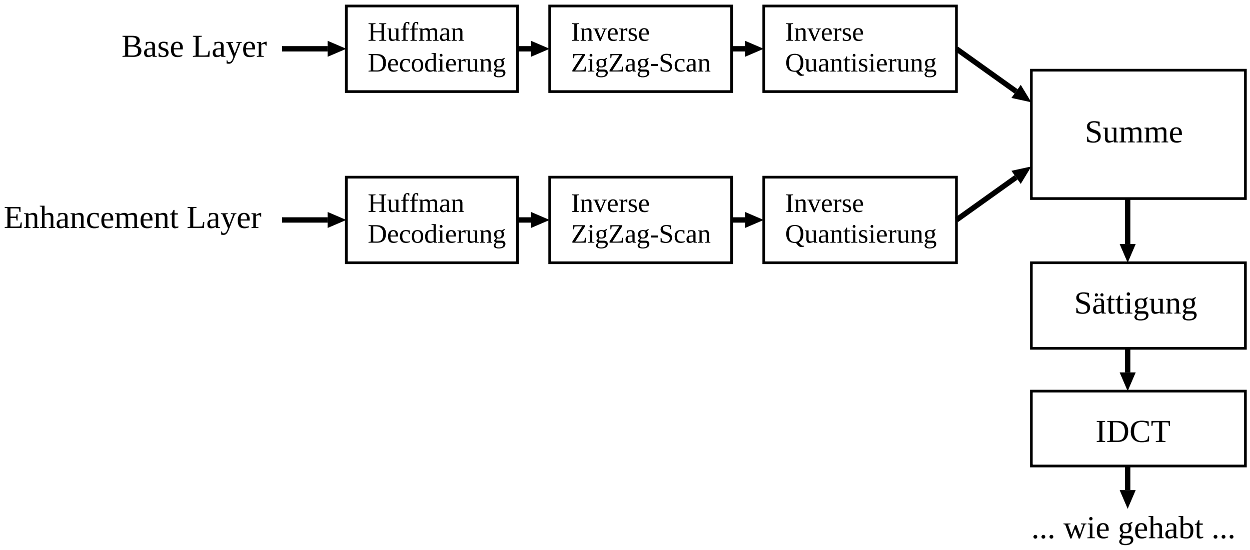


Abbildung 3.10: Decoderaufbau für SNR-Scalability

Auch diese Umcodierung ist noch mit sehr geringem Aufwand möglich. Nach der Huffmandecodierung müssen die Koeffizienten lediglich neu quantisiert werden, eine DCT ist nicht erforderlich. Allerdings muß beachtet werden, daß durch die höherquantisierten Koeffizienten ein gewisser Fehlerdrift erzeugt werden kann, wenn das höhere Layer nicht decodiert wird. Ein Encoder, der diesen Drift verhindert, ist aber nur unwesentlich umfangreicher und das Problem kann durch eine Verkürzung des Abstandes von I-Frames auch verringert werden.

3.5.3 Spatial-Scalability

Bei *Spatial-Scalability* wird im Base-Layer das Video mit einer verringerten Auflösung übertragen. Das höhere Layer enthält eine Mischung aus neuen Motion-Vektoren, DCT-Koeffizienten und Gewichtungsfaktoren. Zur Decodierung muß zunächst der Base-Layer komplett decodiert werden.

Dieses Bild wird dann hochskaliert und ggf. als Ausschnitt eines größeren betrachtet. Bei der Decodierung des höheren Layers wird das decodierte Bild des Base-Layers als Prädiktion benutzt. Das endgültige Bild entsteht dann durch Benutzung der neuen Motion-Vektoren und dem Differenzsignal, das aus den DCT-Koeffizienten gewonnen wird. Das ganze Verfahren ist sehr kompliziert und aufwendig. Eine Realisierung in Software scheint zur Zeit wenig vernünftig, da auf Decoderseite die Komplexität von zwei völlig getrennten Decodern und weiterer zusätzlicher Logik notwendig ist.

Die Bildqualität ist hierbei jedoch auch bei alleiniger Decodierung des Base-Layers sehr gut. Gedacht ist dieser Scalability-Mode für den einfachen Umstieg von Videosystemen im PAL-Format zu Systemen mit HDTV-Auflösungen.

3.5.4 Temporal-Scalability

Hierbei wird die Anzahl der Frames pro Sekunde verringert. Hat das Ausgangsvideo z.B. 50 Bilder/s, so kann es so codiert werden, daß der Base-Layer beispielsweise nur 10 Bilder/s enthält und der Enhancement-Layer die übrigen. Die Codierung ist wesentlich komplexer, als man vielleicht im ersten Moment vermuten würde, da es z.B. bei Temporal-Scalability abweichend von der Regel ausnahmsweise erlaubt ist, B-Frames als Referenzframes zu benutzen. Eine Umcodierung einer vorhandenen Videosequenz in Temporal-Scalability-Bitstreams erscheint nicht sinnvoll, da dafür etliche Motionvektoren neu berechnet werden müßten, was nicht zuletzt auch eine Reduktion der Qualität nach sich ziehen würde. Dieser Scalability-Mode macht nur Sinn, wenn er direkt bei der Erstcodierung eingesetzt wird. Dabei erhöht sich der Codieraufwand bezüglich einer Codierung ohne Scalability nicht wesentlich.

Der Aufwand einer Umcodierung und des Decoders ist auch hier recht hoch, so daß sich eine Softwareimplementation nicht lohnen wird. Die Komplexität dieses Scalability-Modes ist nicht zuletzt deswegen so hoch, da dort auch die Konvertierung von Interlaced-Videos in Progressive-Video und umgekehrt definiert wird.

3.5.5 Hybrid Scalability

Hybrid Scalability bezeichnet keinen eigenen Scalability-Mode, sondern man versteht darunter die Verknüpfung mehrerer Scalability-Modes in einer Übertragung. So kann z.B. zwischen dem Base- und Enhancement-Layer SNR-scalability benutzt werden und daraufhin nochmals Spatial-Scalability zwischen dem Enhancement-Layer und dem Higher-Layer. Allerdings sind nicht sämtliche Kombinationen erlaubt.

3.5.6 Zusammenfassung

Für unsere Anwendung eignen sich nur Data-Partitioning und SNR-Scalability. Die beiden anderen Verfahren scheiden aufgrund der zu hohen Decoderkomplexität von vornherein aus. Sowohl Data-Partitioning, wie auch SNR-Scalability ist auf Encoderseite so einfach zu codieren, daß eine Umcodierung in Echtzeit möglich ist. Der Base-Layer könnte somit so viel Daten enthalten, daß eine Übertragung in beispielsweise 98% der Zeit möglich ist und der Enhancement-Layer weitere Daten, die dann z.B. nur in 50% der Zeit den Decoder erreichen. Da der Server eine Statistik über die Netzwerkbandbreite zur Verfügung hat und das Video in Echtzeit umcodieren kann, ist solch eine Implementation machbar.

Kapitel 4

Videoübertragung

4.1 Die TCP/IP Protokollfamilie

Das Internet ist auf der TCP/IP-Protokollfamilie aufgebaut. Die unterste Schicht von TCP/IP bildet das IP-Protokoll, auf dem alle höheren Protokolle aufsetzen. IP bietet einen unzuverlässigen, verbindungslosen Paketversand. *Unzuverlässig* bedeutet in diesem Zusammenhang, daß Pakete, die aus irgendwelchen Gründen nicht weitergeschickt werden können, einfach gelöscht werden. *Verbindungslos* bezeichnet die Eigenschaft, daß keine Informationen über den Versand von Paketen gesammelt werden. Jedes Paket wird unabhängig von den anderen verschickt. Das bedeutet auch, daß die Pakete in einer anderen Reihenfolge beim Empfänger ankommen können, als der Sender sie geschickt hat. IP-Pakete dürfen auf der Übertragungsstrecke in mehrere kleinere aufgeteilt werden. Sie werden dann auf der Empfängerseite für die höheren Protokolle transparent wieder zusammengesetzt und nur die vollständigen Pakete werden an die Anwendungssoftware weitergegeben. Die Paketverlustrate steigt deshalb mit der Größe des Pakets an.

Die maximale Paketgröße beträgt 65536 Bytes, wobei Hosts nur Pakete bis zu 576 Bytes annehmen müssen. Allerdings verarbeiten die allermeisten heutigen Implementationen auch Pakete mit über 8192 Bytes.

Im IP-Headerfeld *type of service* kann der Paketversand den speziellen Anforderungen angepaßt werden. Folgende vier Möglichkeiten sind vorhanden:

1. Minimize delay für interaktive Anwendungen wie Telnet.
2. Maximize throughput für FTP-Daten, SMTP-Daten (EMail) usw.
3. Maximize reliability
4. Minimize monetary cost für NNTP (Usenet News), ...

Allerdings wird dieses Feld von den meisten Implementationen derzeit noch nicht unterstützt. Auch ist die Aufteilung recht grob, eine feinere Zuweisung von Paketprioritäten ist nicht möglich.

Für den Datentransport stehen vor allem zwei Protokolle zur Auswahl: TCP und UDP. Davon ist TCP ein verbindungsorientiertes Protokoll, das eine feste Verbindung zwischen zwei Rechnern aufbaut (aber ohne eine feste Bandbreite zu garantieren) und eine verlustlose Übertragung gewährleistet. Außerdem kommen die Pakete auch in genau der Reihenfolge beim Empfänger an, in der sie der Sender geschickt hat. Gehen unterwegs Pakete verloren, werden diese nochmals geschickt, bis sie schließlich den Empfänger erreichen.[1] Im Extremfall kann dadurch die Übertragung

[1] Genaugenommen wird ein Sliding-Window Verfahren benutzt.

sehr langsam werden, eine Echtzeitübertragung ist nicht garantierbar. Dieses Protokoll ist für Anwendungen wie EMail gedacht, bei der die Übertragungsgeschwindigkeit zweitrangig ist, dafür aber eine sichere Übertragung gefordert ist.

UDP ist dagegen ein verbindungsloses Protokoll, das keine sichere Paketübertragung realisiert. Pakete können aus diversen Gründen bei der Übertragung ohne Benachrichtigung des Senders oder Empfängers entfernt werden. Es wird lediglich garantiert, daß empfangene Pakete stets komplett sind und der Inhalt korrekt ist. UDP eignet sich für Anwendungen, die eine schnelle Übertragung benötigen und Paketverluste tolerieren können. Da auch die Videoübertragung in diese Kategorie fällt, beschränken wir uns im folgenden auf UDP, falls nicht ausdrücklich anders angegeben.

4.2 Videoübertragung in Netzwerken

4.2.1 VBR- vs. CBR-Übertragung

Der MPEG-Standard läßt eine Codierung einer Sequenz als CBR (constant bitrate) und als VBR (variable bitrate) Bitstream zu. CBR wird vorwiegend in den Fällen benutzt, in denen für die Videoübertragung ein Kanal exklusiv zur Verfügung steht. Da bei der MPEG-Codierung für die verschiedenen Bildtypen stark abweichende Datenvolumina erzeugt werden können, ist eine CBR Übertragung nur durch einen Sende- und Empfangspuffer möglich, der diese schwankende Bitrate glättet. Um ein Bufferüberlauf und -unterlauf zu verhindern, wird die Bildqualität z.B. durch Veränderung des QuantizationScale-Faktors je nach Bufferfüllstand reguliert. Dies kann aber dazu führen, daß ein drohender Bufferüberlauf mit hohem Detailaufkommen im Bild zusammenfällt, was dann zu einer starken Verschlechterung der Bildqualität führen kann, da in diesem Fall der QuantizationScale-Faktor sehr hoch gewählt werden muß. Umgekehrt muß der Encoder den Bitstream bei drohendem Bufferunterlauf notfalls mit Stuffingbits auffüllen, um ihn in einen CBR Bitstream umzuformen. Eine Übertragung eines CBR Bitstreams im Multiplex mit anderen Bitstreams oder im öffentlichen Internet ist somit nicht sehr effizient.

Wird eine VBR Übertragung benutzt, so wird zwar in der Regel auch ein Sendepuffer benutzt, um auf Schwankungen der Netzwerkauslastung reagieren zu können, aber die Bildqualität muß nicht so aggressiv geregelt werden und sehr hohe Maximalbitraten sind möglich. Bei niedrigem Detailgehalt und damit kleiner benötigter Bitrate kann die überschüssige Bandbreite anderen Anwendern zur Verfügung gestellt werden. Durch den geringeren Regelbedarf schwankt der QuantizationScale-Faktor nicht so stark und die resultierende Videoqualität ist gleichmäßiger und höher als bei CBR Übertragung.

4.2.2 Statistical Multiplexing Gain

Schwankt die benötigte Datenrate für einen Datenstream in einem großen Bereich, so ist es ineffizient, einen Kanal mit der maximale Datenrate zu reservieren, da dieser in einem Großteil der Zeit nicht ausgelastet wird. Überträgt man dagegen mehrere solcher Datenstreams im Zeitmultiplex, ist dazu im statistischen Mittel nur ein Kanal mit einer geringeren Bandbreite erforderlich, als die Summe der maximalen Datenraten aller Datenstreams. Dieser Effekt wird als *SMG (Statistical Multiplexing Gain)* bezeichnet. Läßt man Verzögerungen zu, reicht theoretisch ein Kanal mit der Summe der mittleren Datenraten. Begrenzt man die Verzögerung (und damit den Sendepuffer), so muß man allerdings mit Paketverlusten rechnen.

4.3 Messungen

In diesem Abschnitt soll untersucht werden, wie die Paketverlustraten und die Übertragungszeiten von der Paketgröße abhängen. Dazu wurden mehrere Rechner mit unterschiedlicher geographischer Entfernung beliebig ausgewählt und mit dem UNIX-Standardkommando `ping` die Verbindungsqualität gemessen. In jeder Messung wurden 25 Pakete verschickt. Die Messung wurde mehrmals pro Tag (gleichverteilt über den ganzen Tag) durchgeführt und an verschiedenen Tagen wiederholt. Da neuere Versionen des Programms `ping` auf eine Paketgröße von 4096 Bytes beschränkt sind, konnte die Messung nur auf einem Rechner durchgeführt werden, auf dem sich eine Version befand, die Pakete bis zu einer Größe von 65536 Bytes schicken konnte. Deswegen fanden alle Messungen auf dem gleichen Rechner mit dem Namen `rembrandt.informatik.uni-stuttgart.de` statt.

Die Diagramme der Meßergebnisse folgen auf den nächsten Seiten. Auf jeder Seite sind jeweils die Daten für einen Rechner zu finden. Im jeweils oberen Diagramm sind die Paketverlustraten in Prozent $loss(n)$ über der Paketgröße n eingetragen. Die durchgezogene Linie stellt die mittlere Verlustrate dar, die Fehlerbalken die minimale und maximale Verlustrate von allen Messungen.

Das Diagramm in der Mitte gibt davon abgeleitet die *Effektive Paketgröße* an. Dieser Wert zeigt, wieviele Bytes in einem empfangenen Paket durchschnittlich empfangen werden, wenn man über alle zu sendenden Pakete mittelt. Bei der grünen (unteren) Linie wird davon ausgegangen, daß ein Paket nur einmal geschickt wird. Bei Verlust sind seine Daten komplett verloren. Bei der blauen (oberen) Linie wird hingegen erlaubt, daß das Paket einmal nachgeschickt wird. Im ersten Fall berechnet sich die Effektive Paketgröße n_{eff} einfach durch

$$n_{eff} = n \cdot (1 - loss(n))$$

Im zweiten Fall hingegen zu

$$n'_{eff} = n \cdot loss(n) \cdot (1 - loss(n)) \cdot (1 - loss_{min}) \cdot (1 - loss(n))$$

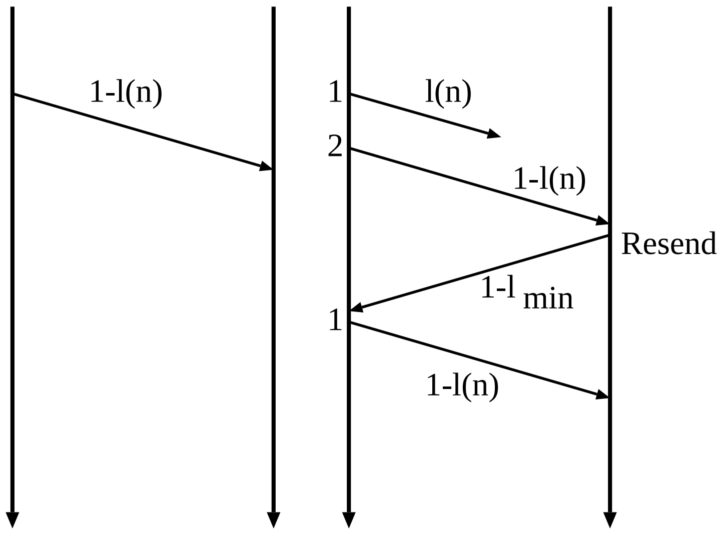
In Abbildung 4.1(b) wird verdeutlicht, wie dieser Ausdruck zustande kommt. Es ist der Ablauf eines Packet-Resendings dargestellt, wie es in dieser Arbeit implementiert wurde. Genauere Informationen dazu stehen in Abschnitt 6.5.2 auf Seite 58. $loss_{min}$ gibt die Größe der ReceptionReport-Message an. Diese ist aber mit sehr großer Wahrscheinlichkeit kleiner als 1480 Bytes, so daß die Verlustrate gering sein wird. Bei der Berechnung wurde $loss_{min}$ einfach als 0 angenommen, was durch die Daten hinreichend begründet wird.

Das letzte Diagramm gibt schließlich noch die mittlere (sowie minimale und maximale) Übertragungszeiten bei verschiedenen Paketgrößen an. Die Angaben sind hier in Millisekunden. Sehr schön ist zu sehen, wie die minimale Übertragungszeit hinreichend genau durch eine lineare Funktion $y = a + b \cdot n$ darstellbar ist.

4.3.1 Interpretation

Alle gemessenen Verbindungen lassen sich anhand der Effektiven Paketgröße recht gut in 4 Klassen einteilen:

- **lineare** Verbindungen, bei denen wenig Verlust auftritt und die Pakete aller Größen befördern, wobei die Verlustrate bei großen Paketen leicht steigen kann (Seiten 32 bis 36).
- **weiche best-Größe** Verbindungen, bei denen es eine maximale Effektive Paketgröße gibt, nach der die Effektive Paketgröße aber nicht allzu abrupt abnimmt. Die Verlustrate steigt dann meist bis 100% an, wo sie auch konstant bleibt (Seiten 37 bis 40).



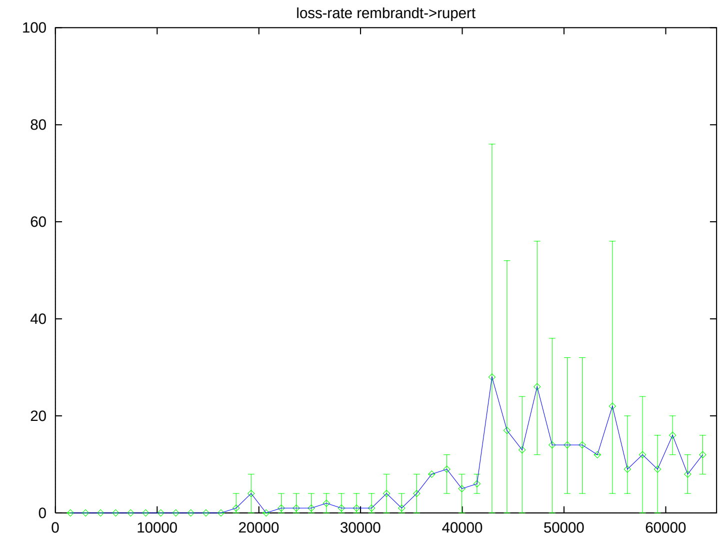
(a) direkter Empfang (b) Empfang nach 1. Wiederholung

Abbildung 4.1: Wahrscheinlichkeiten für den Empfang

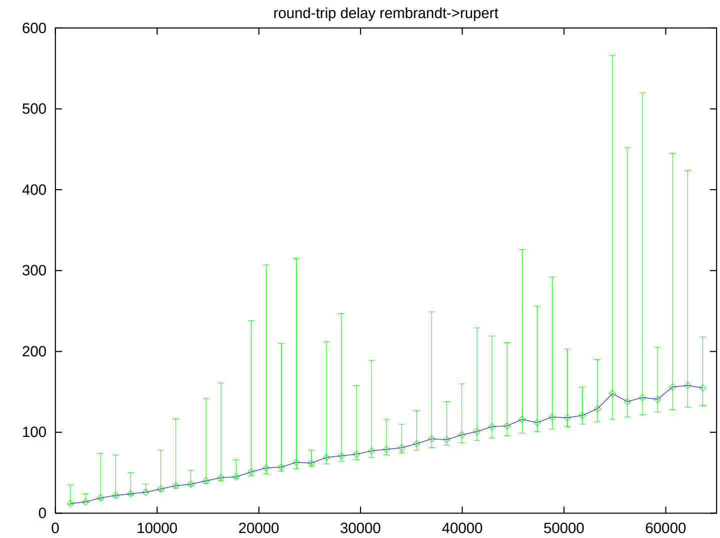
- **harte best-Größe** entspricht der Klasse *weiche best-Größe*, allerdings steigt hier die Effektive Paketgröße ziemlich linear an, um dann ganz plötzlich komplett auf 0 zu sinken, da die Verlustrate auf 100% springt (Seiten 41 bis 44).
- **big-is-beautiful** Pakete größer als 1480 Bytes haben stets eine Verlustrate von 100%, diese fällt aber ganz plötzlich ab einer Paketgröße von etwa 45000 Bytes sehr weit auf wenige Prozent (bis unter 20%). Mir ist unklar, für welchen Dienst es vorteilhaft sein könnte, die Übertragung in dieser Weise zu beschränken. (Seiten 45 und 46).

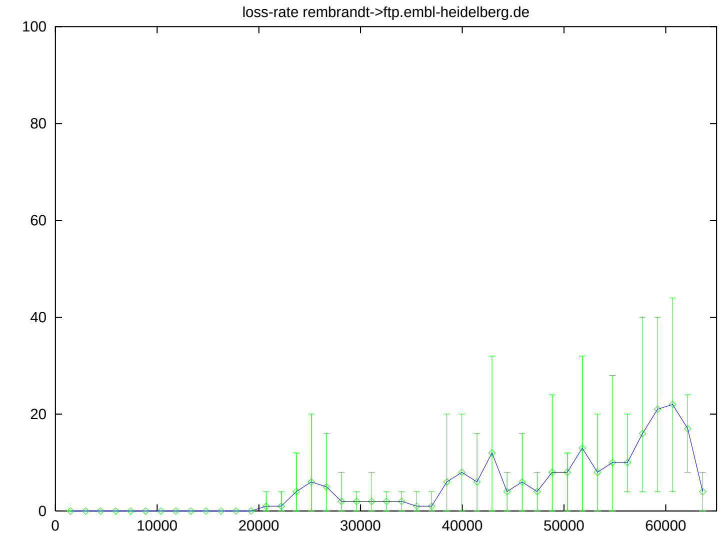
Es zeigte sich, daß manche Grenzen noch schärfer sind, als in den Diagrammen dargestellt. So war die Verlustrate bei `ftp.doc.ic.ac.uk` und `wuarchive.wustl.edu` bis 1480 Bytes fast 0%, stieg bei 1481 Bytes aber sprunghaft auf 100%. Dies läßt sich dadurch erklären, daß ab 1481 Bytes die Message in zwei Ethernet-Frames aufgeteilt werden muß. Einige Router scheinen aber ganz streng nur Nachrichten weiterzuleiten, die in ein einzelnes Ethernet-Frame passen.

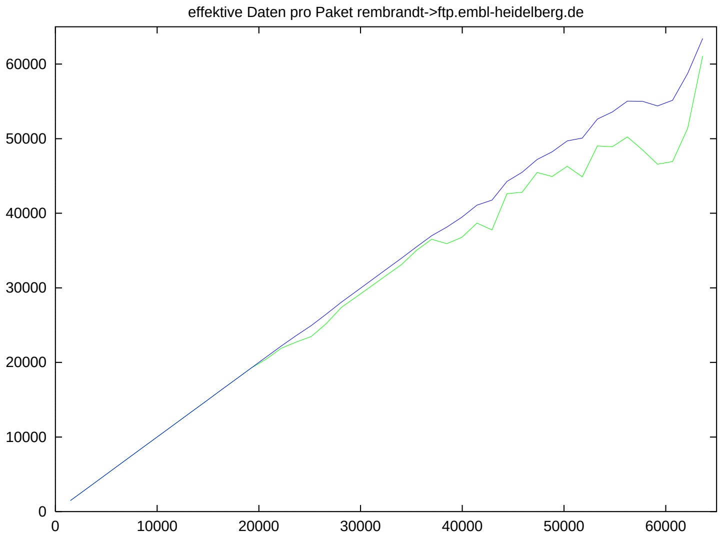
```
ftp.doc.ic.ac.uk
================
Groesse    Laufzeiten     Verlustrate   Effektive Paketgroesse
         min  avg  max    min avg max  kein resend            ein resend
-----------------------------------------------------------------------
 1478     43   44   50      0   0   0  1478.000000 0.000000  1478.000000
 1479     43   44   46      0   0   0  1479.000000 0.000000  1479.000000
 1480     43   44   47      0   0   0  1480.000000 0.000000  1480.000000
 1481   (273)(273)(273)   100 100 100     0.000000 0.000000     0.000000
 1482   (273)(273)(273)   100 100 100     0.000000 0.000000     0.000000
 1483   (273)(273)(273)   100 100 100     0.000000 0.000000     0.000000
```

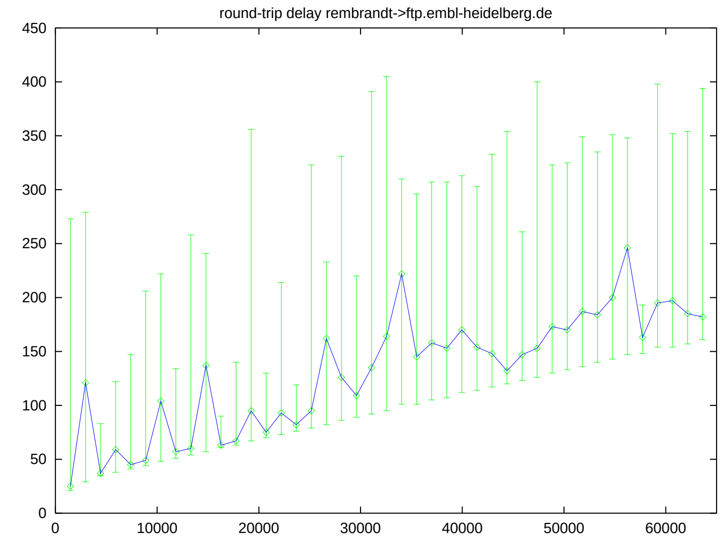

loss-rate rembrandt->rupert
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

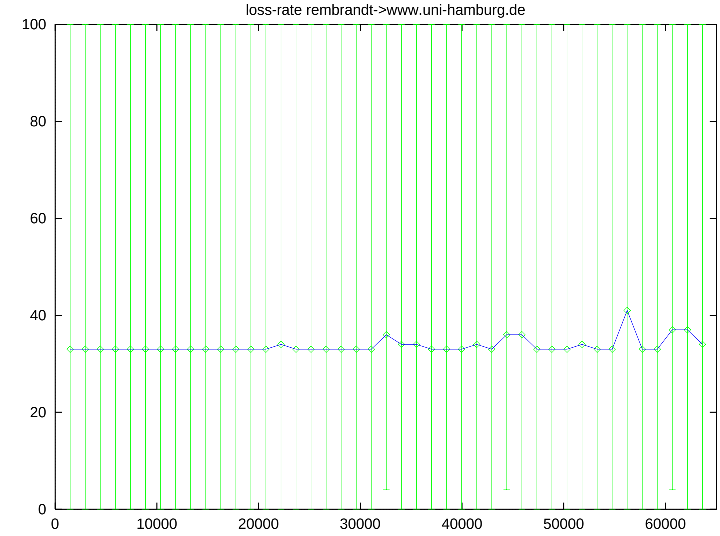

effektive Daten pro Paket rembrandt->rupert
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

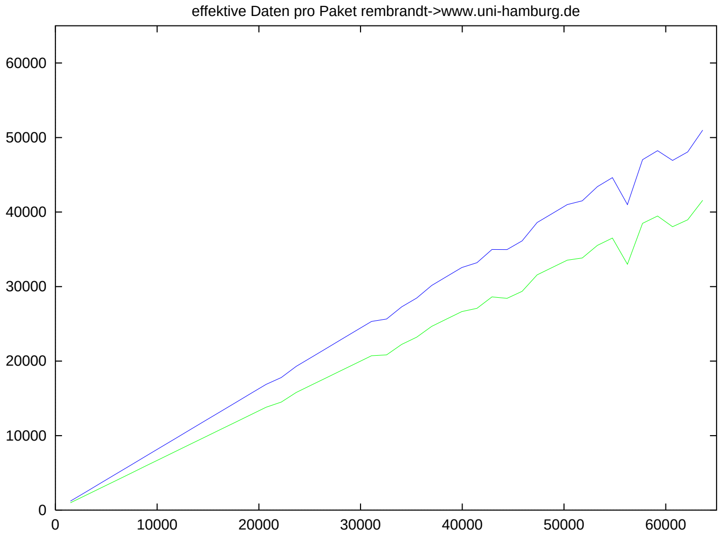

round-trip delay rembrandt->rupert
600
500
400
300
200
100
0
0
10000
20000
30000
40000
50000
60000

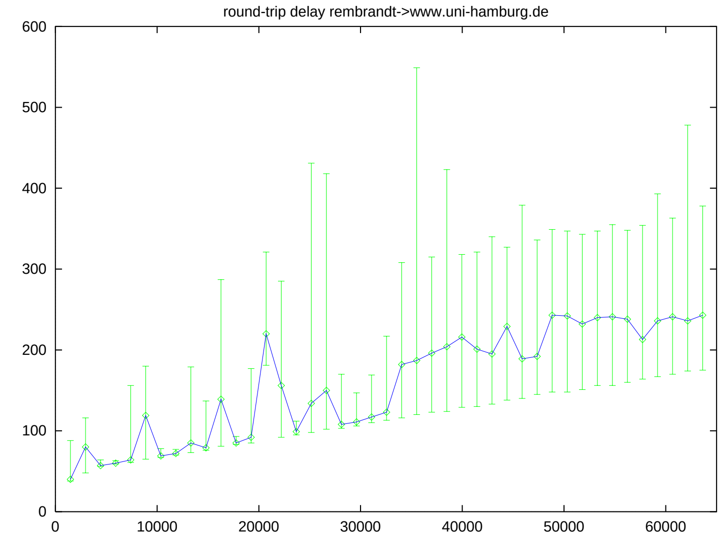

loss-rate rembrandt->ftp.embl-heidelberg.de
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

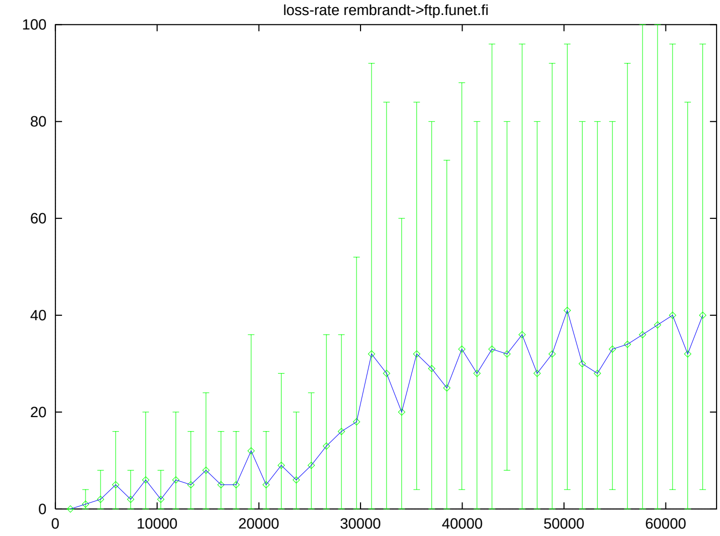

effektive Daten pro Paket rembrandt->ftp.embl-heidelberg.de
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

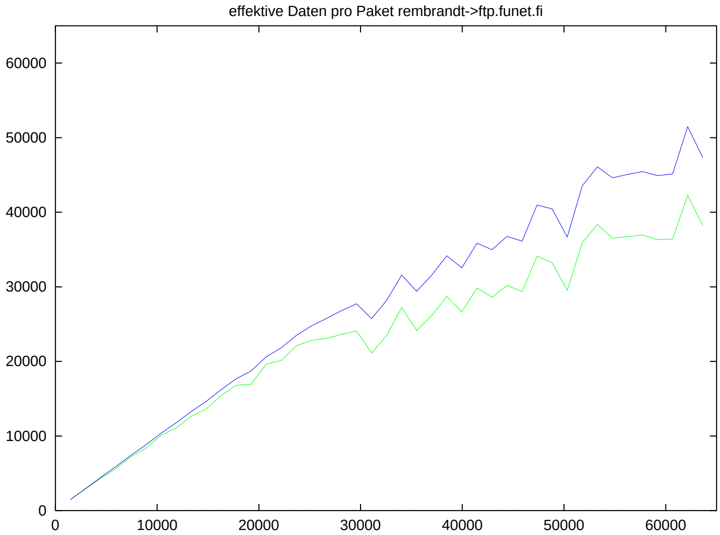

round-trip delay rembrandt->ftp.embl-heidelberg.de
450
400
350
300
250
200
150
100
50
0
0
10000
20000
30000
40000
50000
60000

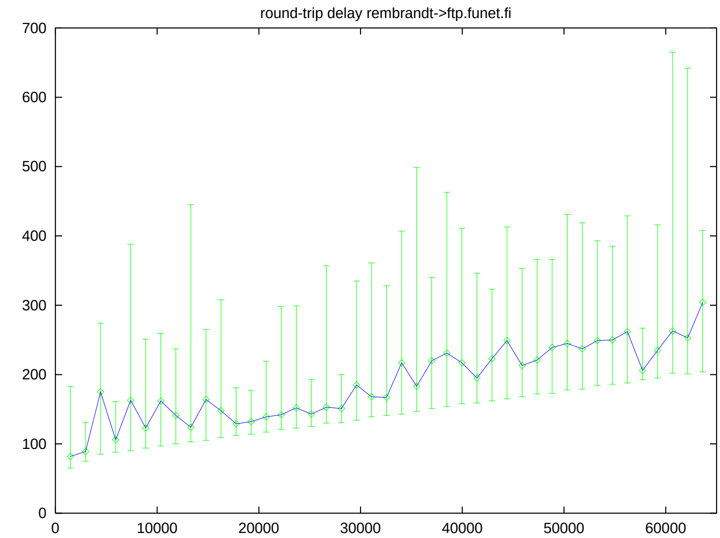

loss-rate rembrandt->www.uni-hamburg.de
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

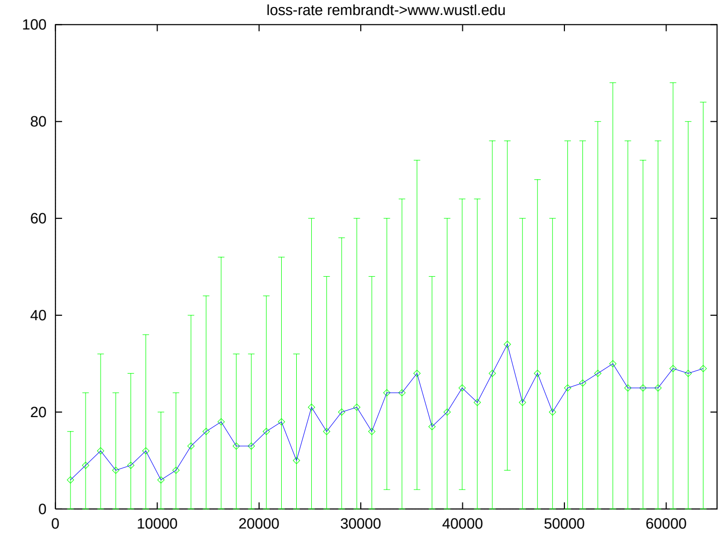

effektive Daten pro Paket rembrandt->www.uni-hamburg.de
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

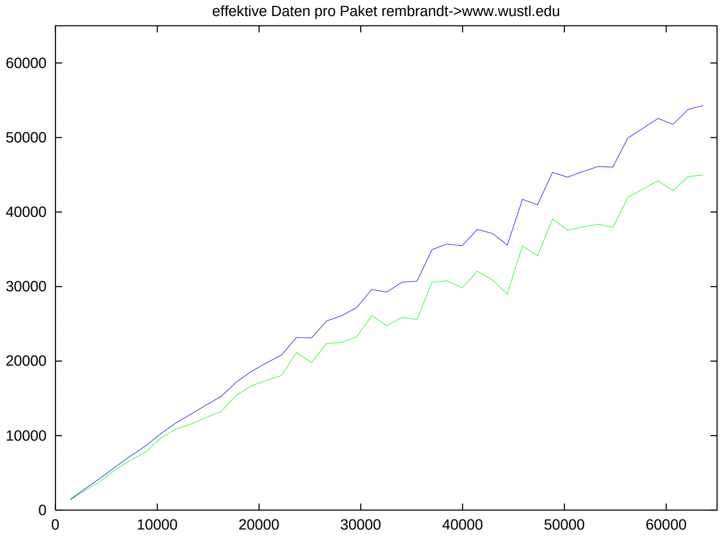

round-trip delay rembrandt->www.uni-hamburg.de
600
500
400
300
200
100
0
0
10000
20000
30000
40000
50000
60000

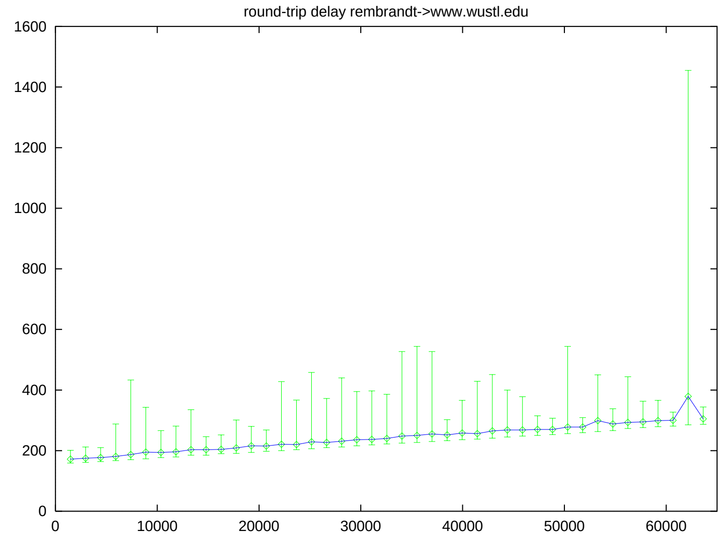

loss-rate rembrandt->ftp.funet.fi
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

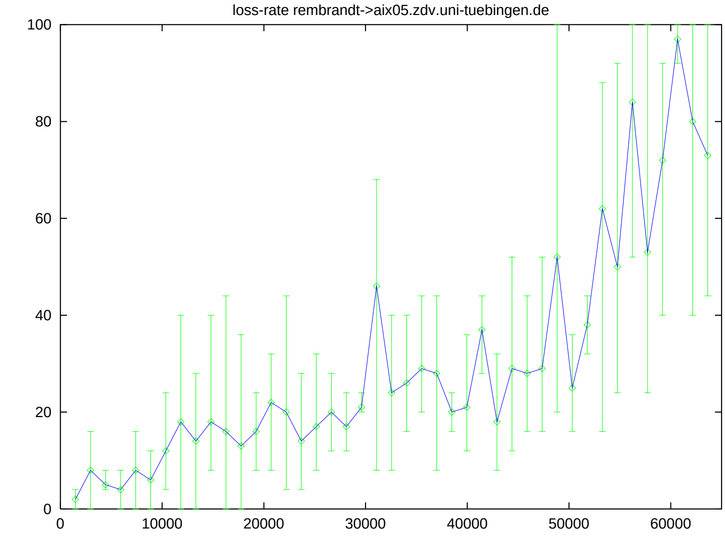

effektive Daten pro Paket rembrandt->ftp.funet.fi
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

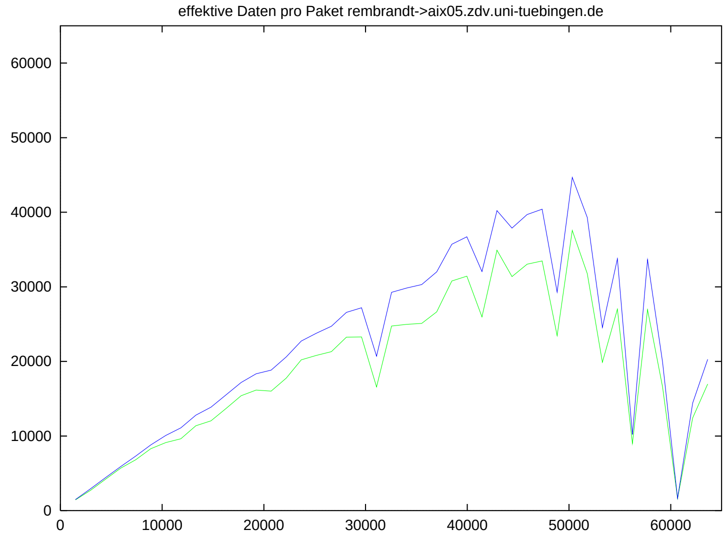

round-trip delay rembrandt->ftp.funet.fi
700
600
500
400
300
200
100
0
0
10000
20000
30000
40000
50000
60000

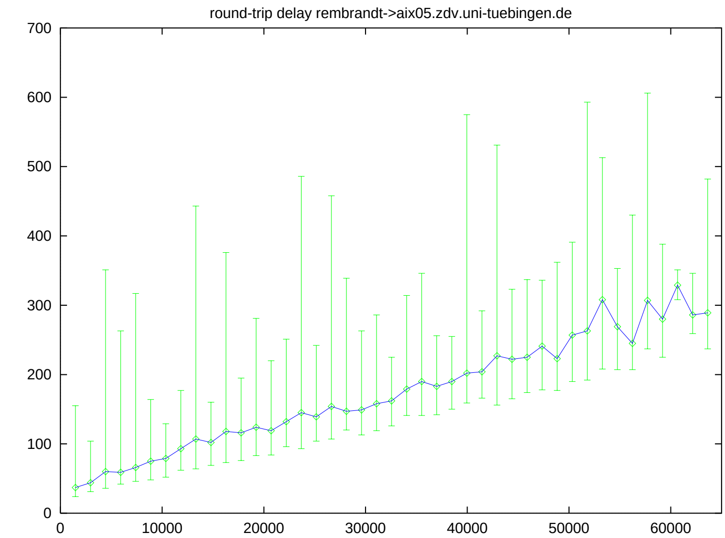

loss-rate rembrandt->www.wustl.edu
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

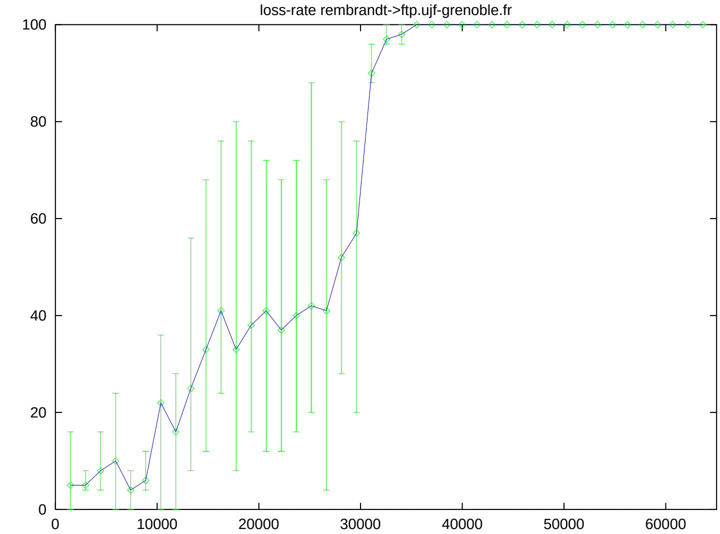

effektive Daten pro Paket rembrandt->www.wustl.edu
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

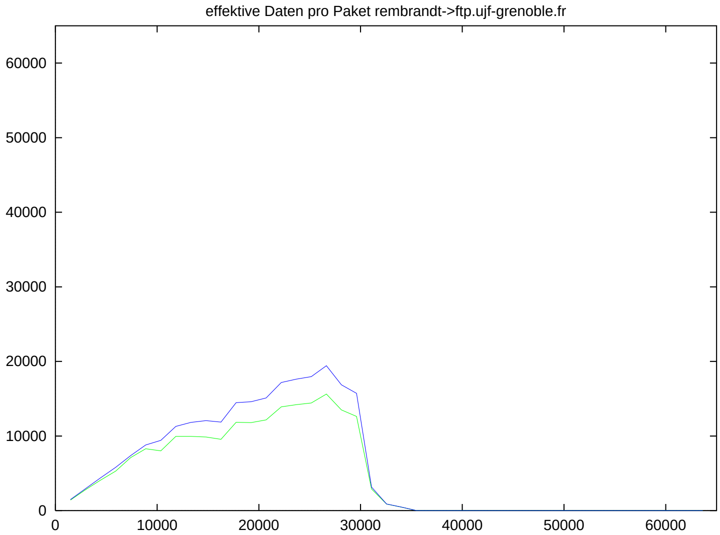

round-trip delay rembrandt->www.wustl.edu
1600
1400
1200
1000
800
600
400
200
0
0
10000
20000
30000
40000
50000
60000

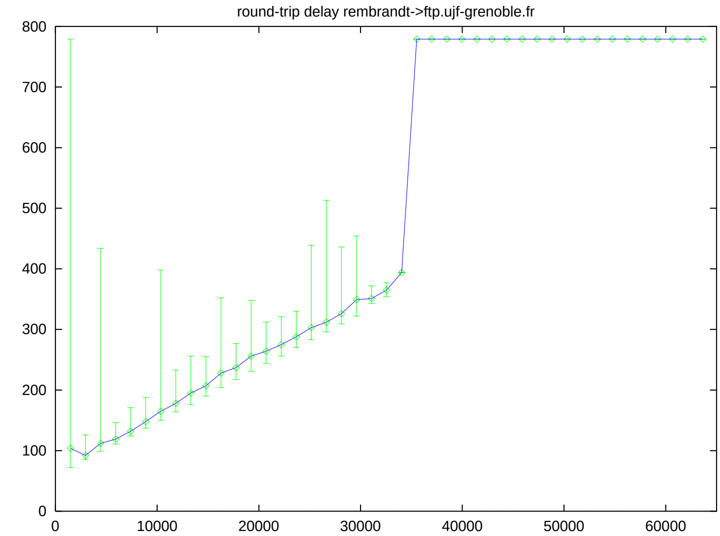

loss-rate rembrandt->aix05.zdv.uni-tuebingen.de
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

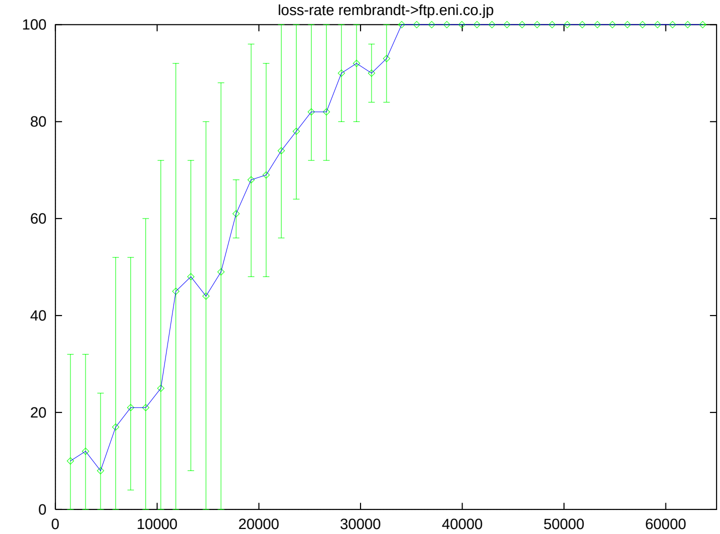

effektive Daten pro Paket rembrandt->aix05.zdv.uni-tuebingen.de
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

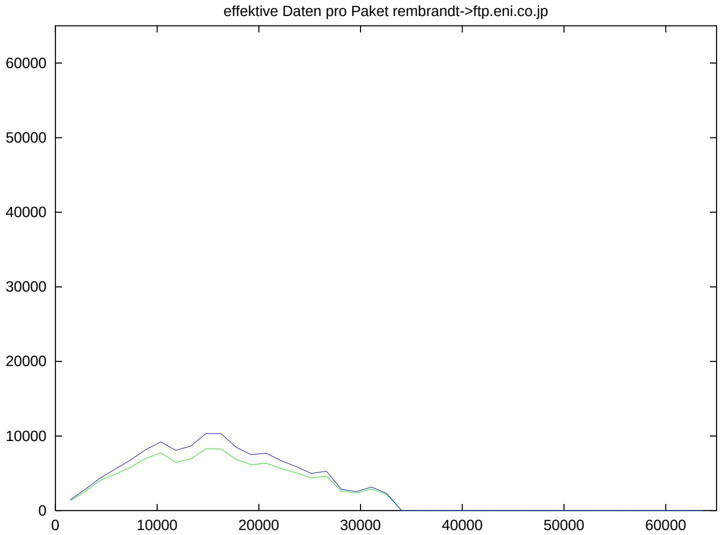

round-trip delay rembrandt->aix05.zdv.uni-tuebingen.de
700
600
500
400
300
200
100
0
0
10000
20000
30000
40000
50000
60000

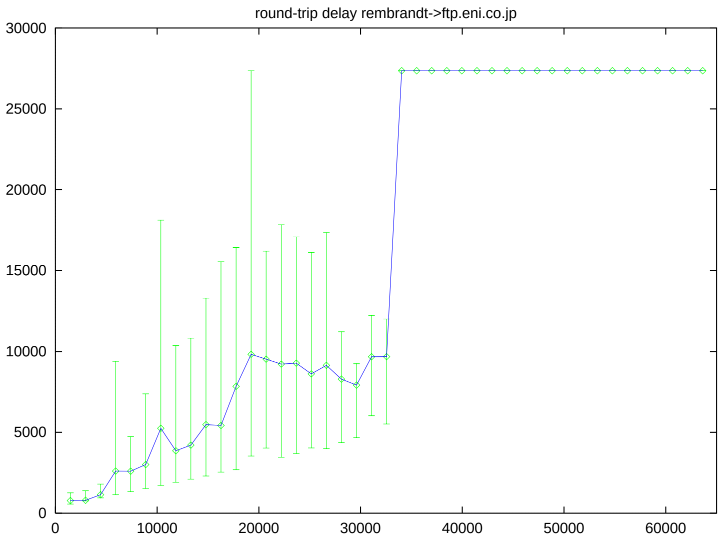

loss-rate rembrandt->ftp.ujf-grenoble.fr
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000

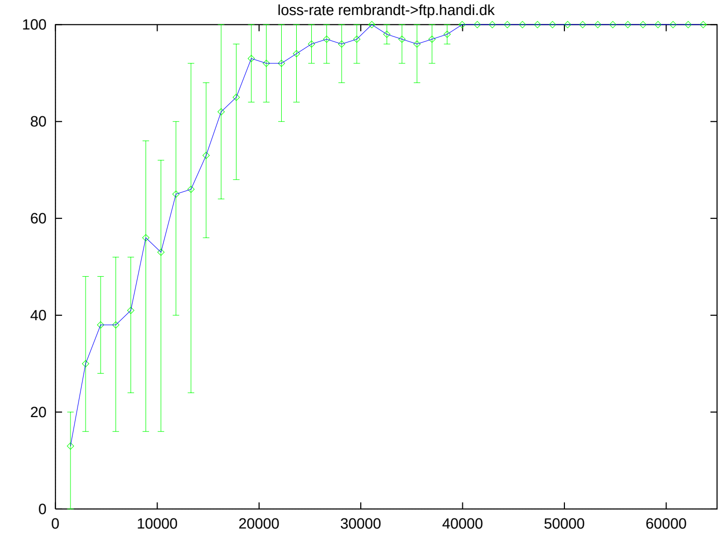

effektive Daten pro Paket rembrandt->ftp.ujf-grenoble.fr
0
10000
20000
30000
40000
50000
60000
0
10000
20000
30000
40000
50000
60000

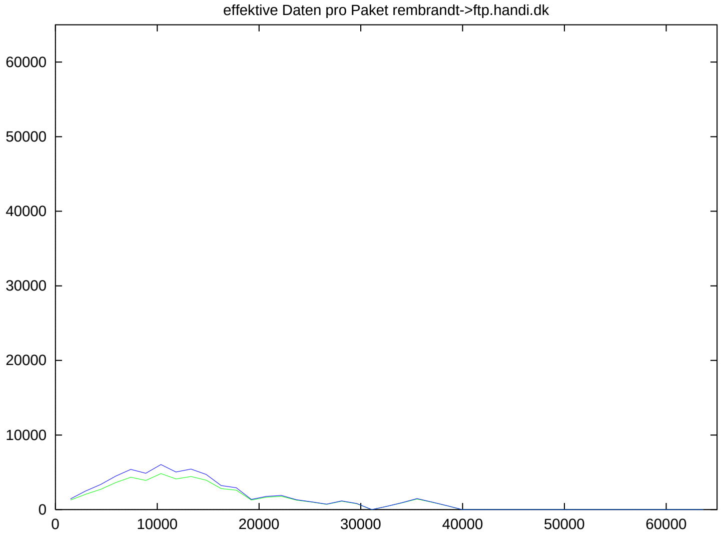

round-trip delay rembrandt->ftp.ujf-grenoble.fr
0
100
200
300
400
500
600
700
800
0
10000
20000
30000
40000
50000
60000

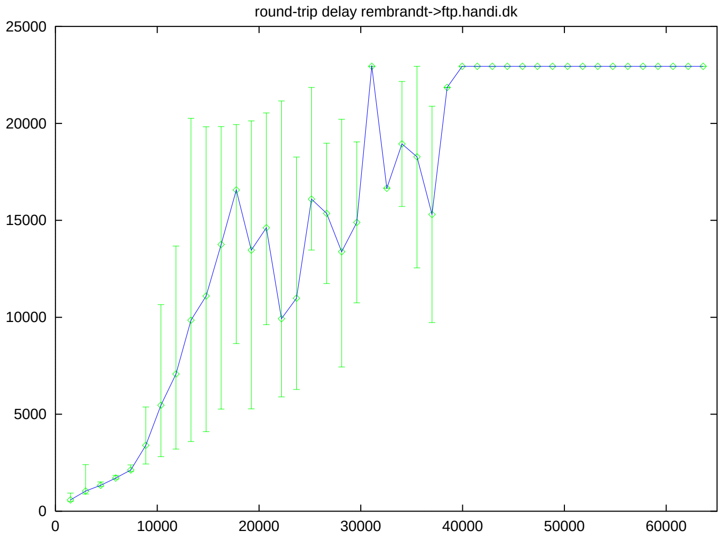

loss-rate rembrandt->ftp.eni.co.jp
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

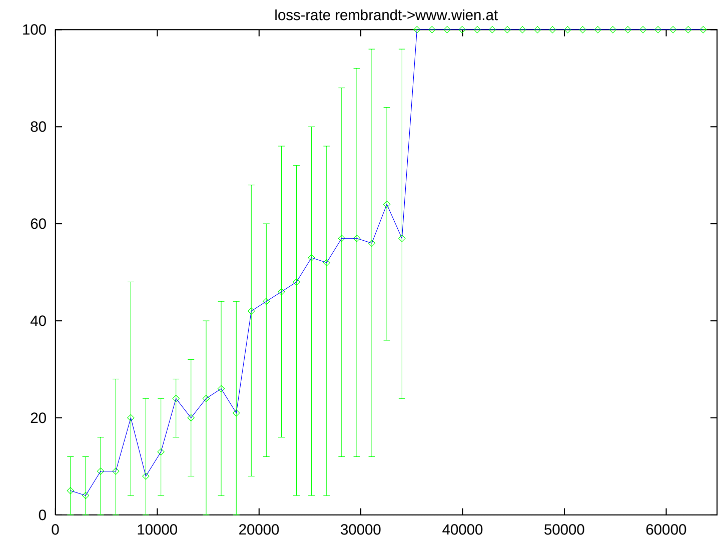

effektive Daten pro Paket rembrandt->ftp.eni.co.jp
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

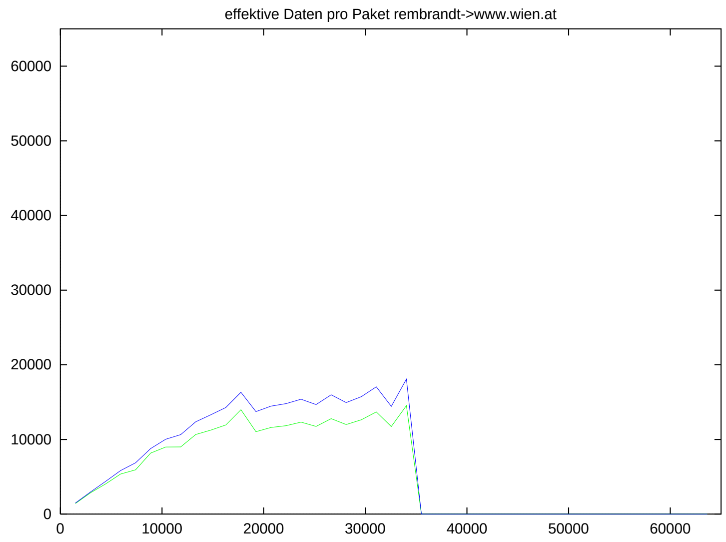

round-trip delay rembrandt->ftp.eni.co.jp
30000
25000
20000
15000
10000
5000
0
0
10000
20000
30000
40000
50000
60000

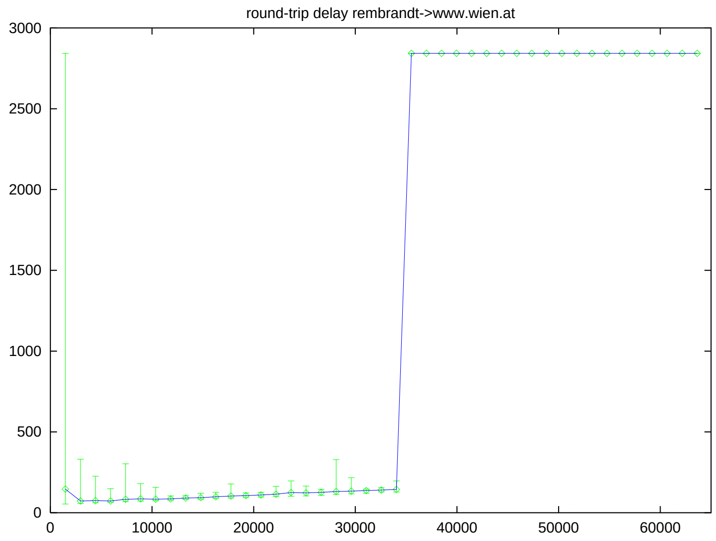

loss-rate rembrandt->ftp.handi.dk
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

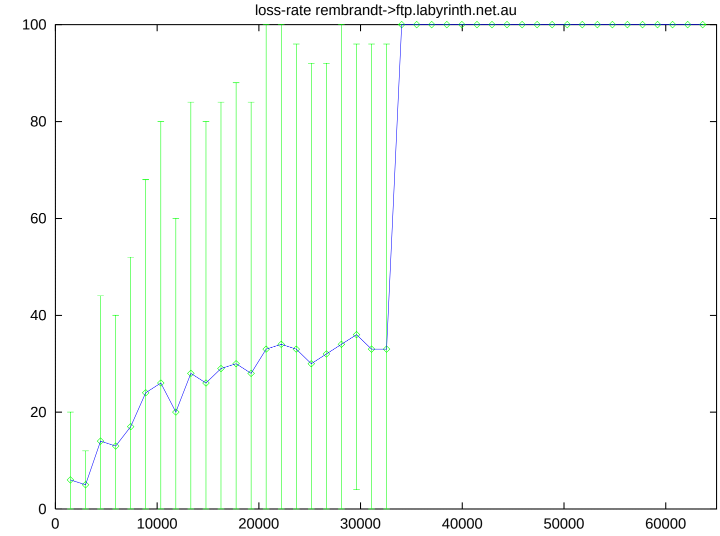

effektive Daten pro Paket rembrandt->ftp.handi.dk
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

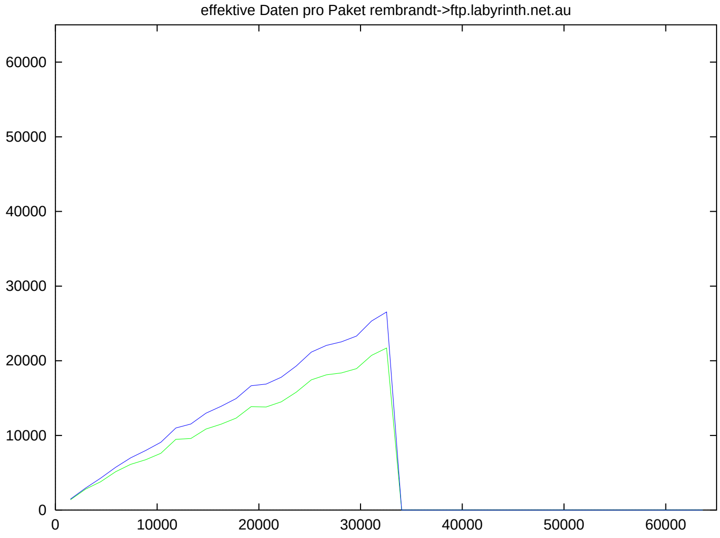

round-trip delay rembrandt->ftp.handi.dk
25000
20000
15000
10000
5000
0
0
10000
20000
30000
40000
50000
60000

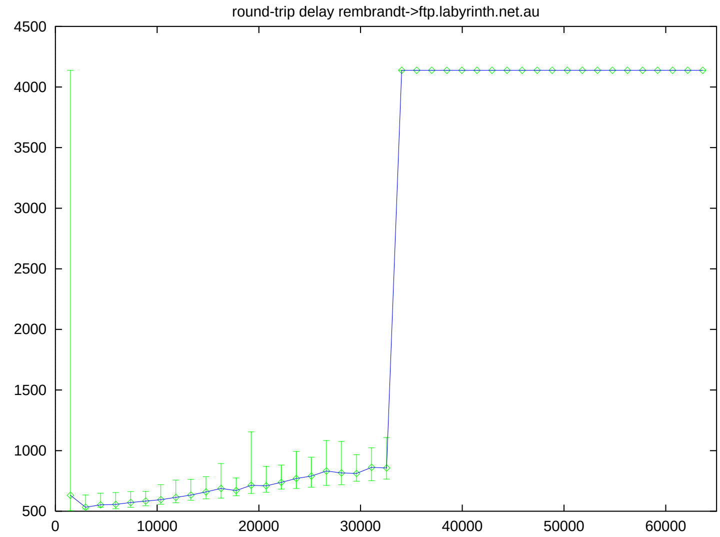

loss-rate rembrandt->www.wien.at
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

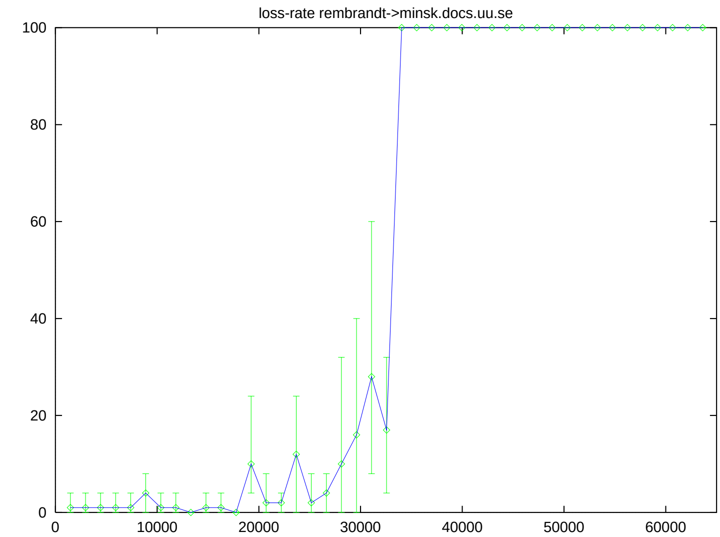

effektive Daten pro Paket rembrandt->www.wien.at
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

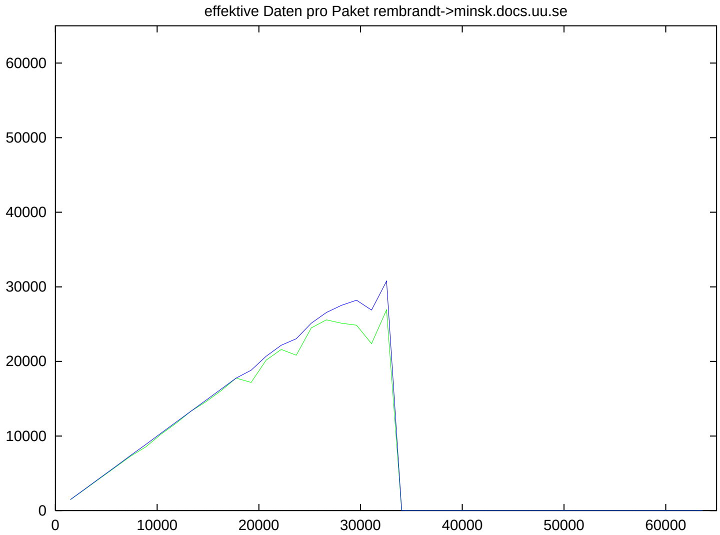

round-trip delay rembrandt->www.wien.at
3000
2500
2000
1500
1000
500
0
0
10000
20000
30000
40000
50000
60000

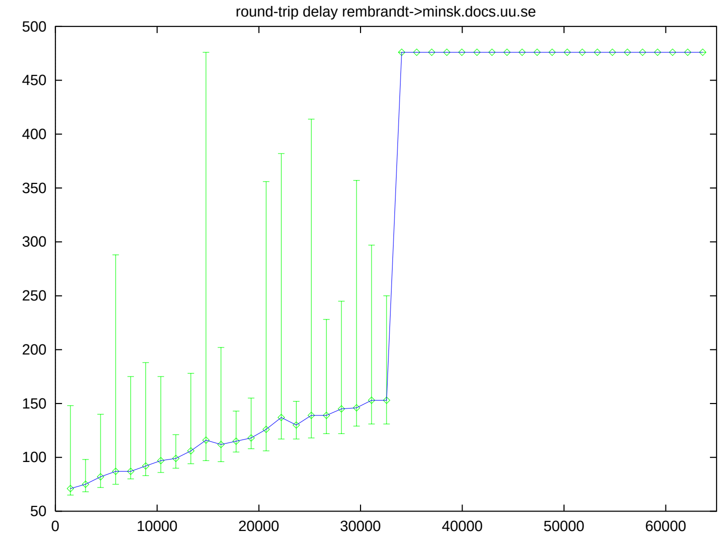

loss-rate rembrandt->ftp.labyrinth.net.au
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000

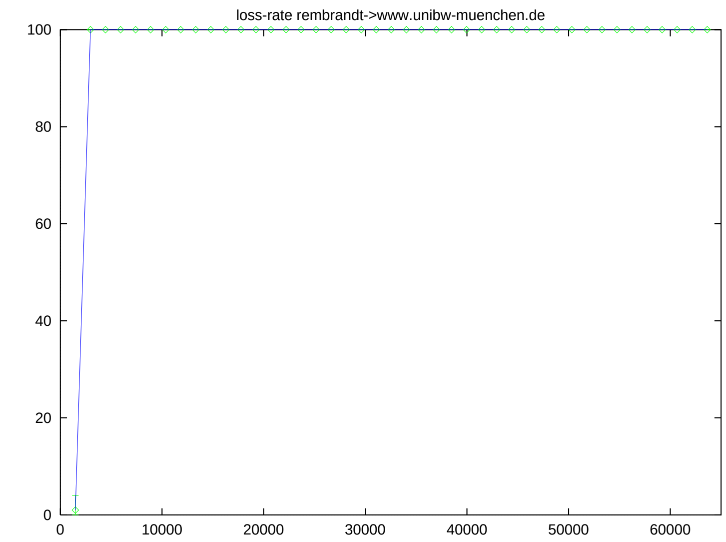

effektive Daten pro Paket rembrandt->ftp.labyrinth.net.au
0
10000
20000
30000
40000
50000
60000
0
10000
20000
30000
40000
50000
60000

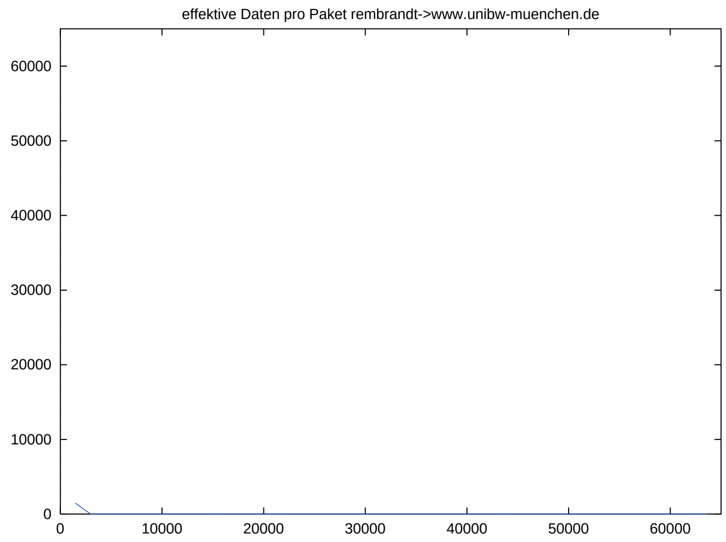

round-trip delay rembrandt->ftp.labyrinth.net.au
500
1000
1500
2000
2500
3000
3500
4000
4500
0
10000
20000
30000
40000
50000
60000

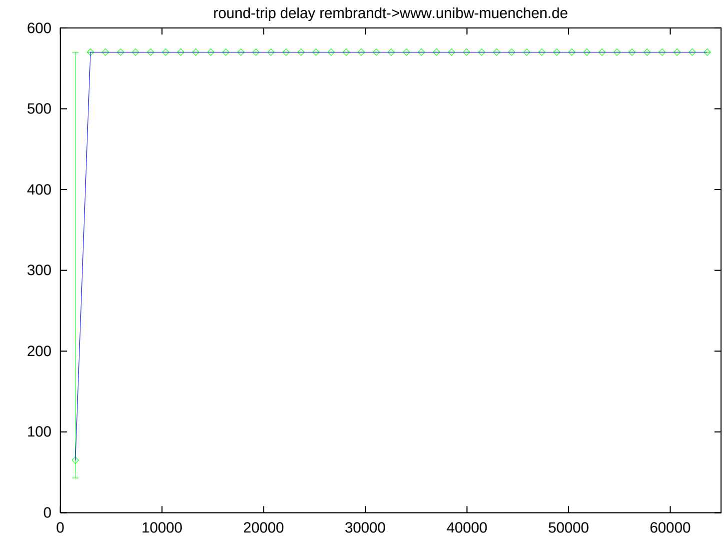

loss-rate rembrandt->minsk.docs.uu.se
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

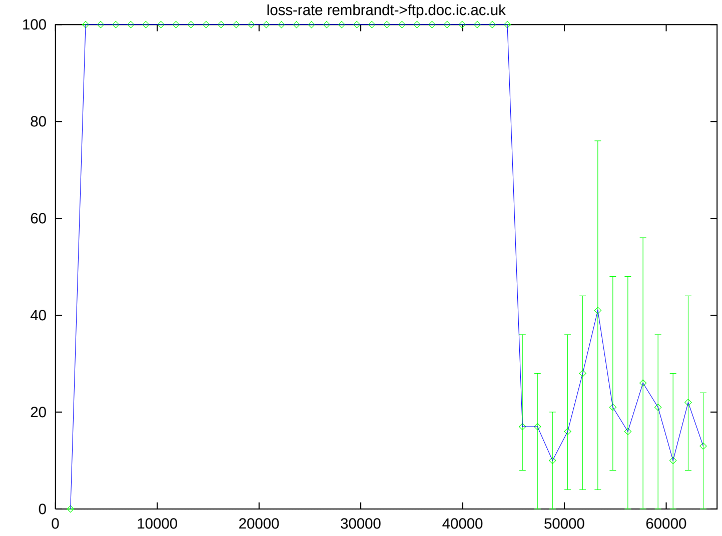

effektive Daten pro Paket rembrandt->minsk.docs.uu.se
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000

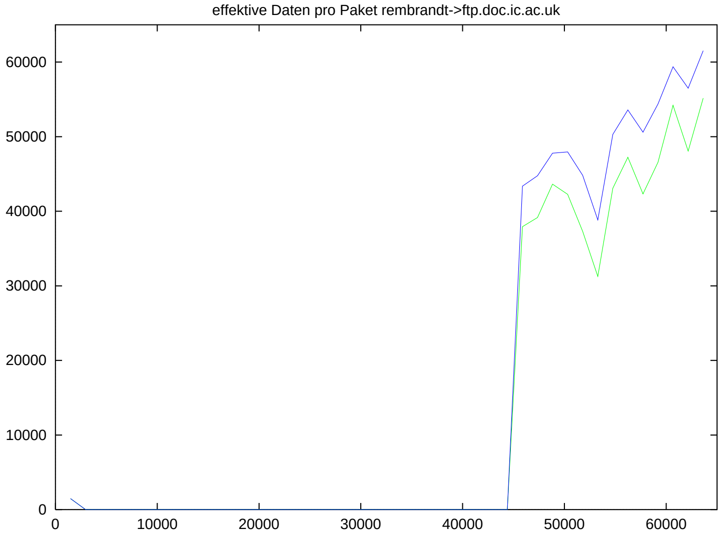

round-trip delay rembrandt->minsk.docs.uu.se
500
450
400
350
300
250
200
150
100
50
0
10000
20000
30000
40000
50000
60000

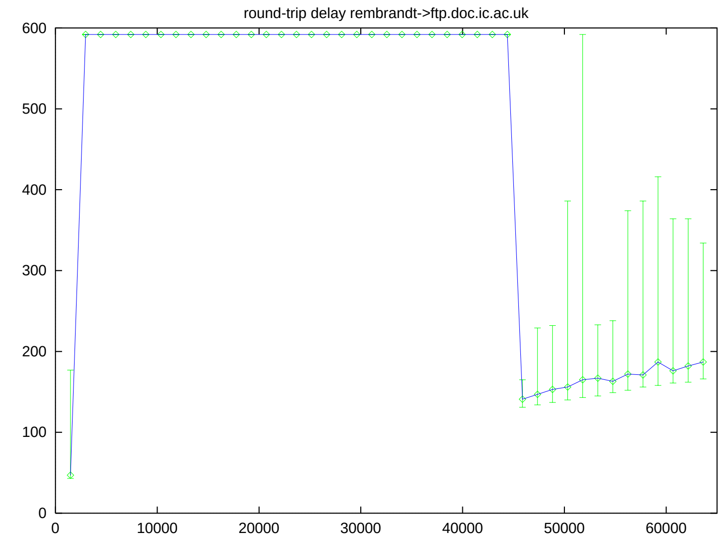

loss-rate rembrandt->www.unibw-muenchen.de
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000

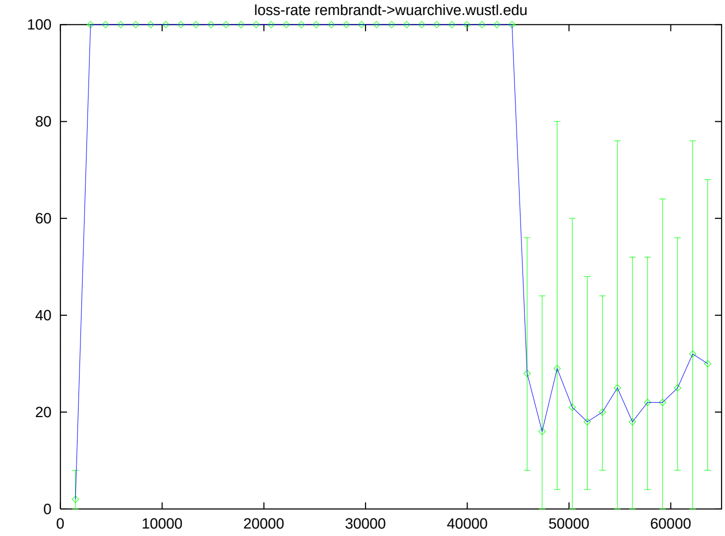

effektive Daten pro Paket rembrandt->www.unibw-muenchen.de
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000


round-trip delay rembrandt->www.unibw-muenchen.de
600
500
400
300
200
100
0
0
10000
20000
30000
40000
50000
60000

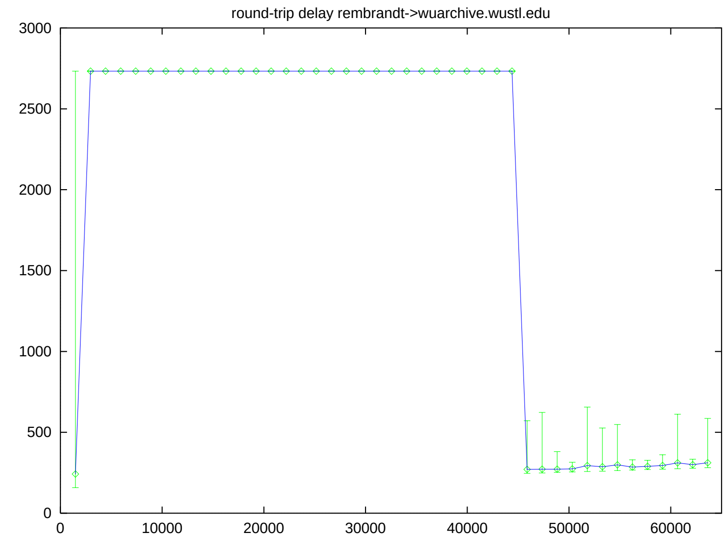

loss-rate rembrandt->ftp.doc.ic.ac.uk
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000


effektive Daten pro Paket rembrandt->ftp.doc.ic.ac.uk
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000


round-trip delay rembrandt->ftp.doc.ic.ac.uk
600
500
400
300
200
100
0
0
10000
20000
30000
40000
50000
60000


loss-rate rembrandt->wuarchive.wustl.edu
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000


effektive Daten pro Paket rembrandt->wuarchive.wustl.edu
60000
50000
40000
30000
20000
10000
0
0
10000
20000
30000
40000
50000
60000


round-trip delay rembrandt->wuarchive.wustl.edu
3000
2500
2000
1500
1000
500
0
0
10000
20000
30000
40000
50000
60000

Kapitel 5

Bestehende Arbeiten und Software

5.1 Arbeiten

Die meisten Arbeiten gehen davon aus, daß das ganze Video vollständig zum Client übertragen wird und beschäftigen sich mit einer Regelung der Codierung um die Bitrate der Netzwerksituation anzupassen. Auch sollen meist mehrere Clients gleichzeitig durch einen Server bedient werden. Eine individuelle Anpassung kann dann nicht mehr erfolgen. Als Netzwerktyp kommen fast ausschließlich ATM-Netzwerke zur Sprache.

Ein Mittelweg wird in [TB] beschrieben. In Netzwerken mit unterschiedlichen Übertragungskapazitäten auf verschiedenen Zweigen kann an jedem Router eine Umcodierung erfolgen, um die Bitrate zu reduzieren.

Der Ansatz, der am nächsten an dieser Arbeit ist, wird recht ausführlich in [Net] beschrieben. Anstelle eines Priority-Heaps verwendet der Autor aber einen Algorithmus, den er als *Cyclic-UDP* bezeichnet. Die zu sendenden Datenpakete werden in einen Ringbuffer geschrieben und in Bursts gesendet. Ein Resend wird immer dann durchgeführt, wenn das Paket wieder auf diesem Ring vorbeikommt und noch kein Acknowledge vom Client empfangen wurde. Auf dieser Arbeit basiert auch der in Abschnitt 7.4 leicht abgewandelte Antijitter-Algorithmus.

Der Artikel [ACR96] befaßt sich ausführlich mit dem Zusammenwirken von MPEG-Scalability-Modes und Paketverlusten. Ihr Ergebnis ist, daß Data-Partitioning für niedrige Bitraten im Base-Layer keine guten Resultate liefert und bei SNR-Scalability ein gewisser Fehlerdrift erkennbar ist, wenn nur der Base-Layer decodiert wird. Spatial-Scalability erbrachte die besten Resultate, bedeutet aber auch einen viel höheren Rechenaufwand.

In [SKZ96] wird beschrieben, wie ein Transcoder zur Echtzeitumwandlung von normalen MPEG-Streams in zwei mit Scalability-Modes codierte Streams implementiert werden kann.

Eine Zusammenfassung der Synchronisation mehrerer Elementary-Streams in einem MPEG-System-Stream und ein Vorschlag der Implementation von Server- und Clientbuffer findet sich in [RKR96].

5.2 Andere Codierungsverfahren und Protokolle

Ein sehr ähnliches Codierungsverfahren wie MPEG ist H.261, das auch als $p \times 64$ bezeichnet wird, da es vorrangig für Bildtelefonie über ISDN-Leitungen (mit jeweils 64kbit/s) gedacht war. Der Hauptunterschied zu MPEG ist, daß hier keine B-Frames verwendet werden. Dadurch ergibt sich keine Zeitverzögerung zur Codierung, denn I- und P-Frames können immer direkt nach der Codierung verschickt werden. Die einzige Verzögerung ergibt sich durch den Ausgabebuffer, der wie in MPEG aufgebaut ist. Geringe Verzögerungen sind wichtig für interaktive Anwendungen wie eben die Bildtelefonie.

In RFC 1889 [RFCa] wird ein Übertragungsprotokoll für Video- und Audiodaten beschrieben, das den Schwerpunkt auf Videokonferenzen legt. So ist ausführlich geregelt, wie Anmeldungen erfolgen und mehrere Videostreams von einem *Mixer* in ein größeres Bild zusammengefaßt werden sollen. Nicht definiert ist hingegen, welche Codierungsverfahren für Video und Audio benutzt werden sollen und wie die Datenpakete auszusehen haben.

Dies wird in RFC 2038 [RFCc] für MPEG-Daten und in RFC 2032 [RFCb] für H.261 definiert. Die Regel, wie MPEG-Streams in Paket unterteilt werden sollen, ist dabei für MPEG-Elementary-Streams:

- Ein *Sequence-Header* befindet sich immer am Beginn eines Pakets.
- Ein *GroupOfPictures-Header* befindet sich entweder am Beginn oder direkt hinter dem Sequence-Header.
- Ein *Picture-Header* befindet sich entweder am Beginn des Pakets oder hinter dem GroupOfPictures-Header

Die Bilddaten können dann (wenn gewünscht) auch auf mehrere Pakete verteilt werden.

5.3 Software

Wärend der Bearbeitungszeit dieser Arbeit hat die Anzahl der Angebote von Videoübertragungen im Internet erheblich zugenommen. Dazu werden im Regelfall die kommerziellen Softwarepakete *RealVideo* und *VDOLive* benutzt. Der Player ist in beiden Fällen kostenlos erhältlich.

5.3.1 RealAudio/RealVideo

Die sehr bekannte, kommerzielle Software *RealAudio*, wurde jetzt zu *RealVideo* ergänzt. Die Server-Software erlaubt den gleichzeitigen, unabhängigen Zugriff von vielen Clients. Die Eingabedaten müssen in ein spezielles Format konvertiert werden, bevor sie gesendet werden können. Die verwendeten Codecs sind nicht dokumentiert, aber anhand der Artefakte im Video scheint eine MPEG-ähnliche Codierung benutzt zu werden. Zusätzlich kann man auch noch eine fraktale Videocodierung wählen, die bei gleicher Datenrate etwas bessere Bilder liefert.

Die Daten werden so umcodiert, daß ihre Datenraten innerhalb der üblichen Übertragungsraten liegen. Bei RealAudio sind das:

Verbindung/kbps	Datenrate/kbps	Kanalauslastung
14.4	8	56%
28.8	16	56%
64	40	62%
2·64	80	62%

und bei RealVideo:

Verbindung/kbps	Audio/kbps	Video/kbps	Summe/kbps	Kanalauslastung
28.8	6.5 - 8	11 - 12	19	66%
56	8.5 - 16	29 - 36.5	45	80%
112	16	74	90	80%

Wärend der Übertragung wird diese Datei einfach so geschickt, wie sie ist. Der Client scheint fehlende Pakete nachfordern zu können. Wie dies im einzelnen abläuft, ist aber nicht erkennbar. Die Anzahl der gesendeten Pakete scheint in Abhängigkeit vom jeweiligen Stream etwa zwischen 2 und 10 pro Sekunde zu schwanken.

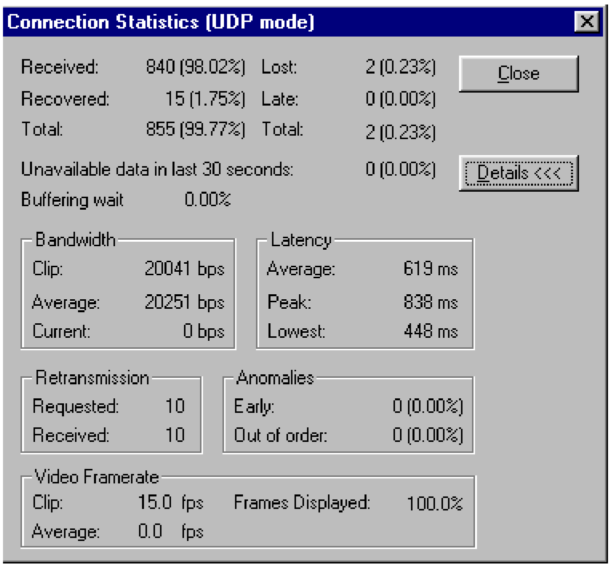
In der subjektiven Betrachtung hat man das Gefühl, daß das Video entweder bei guten Bedingungen fast vollständig übertragen wird oder bei schlechten Bedingungen so schlecht, daß es unbrauchbar wird. Die Qualität der Einzelbilder nimmt nicht mit den Paketverlusten ab, teilweise werden die Bilder dann aber nur unvollständig dargestellt.

In einem Statistikfenster (Abbildung 5.1(a)) kann verfolgt werden, wie die Übertragung abläuft. Allerdings bin ich mir nicht sicher, ob man diesen Daten blind vertrauen kann, da das Video bedeutend schlechter aussieht, als man laut der Statistik meinen könnte. Speziell die Framerate kann nach dem subjektiven Empfinden nicht mehr als 5 fps betragen.

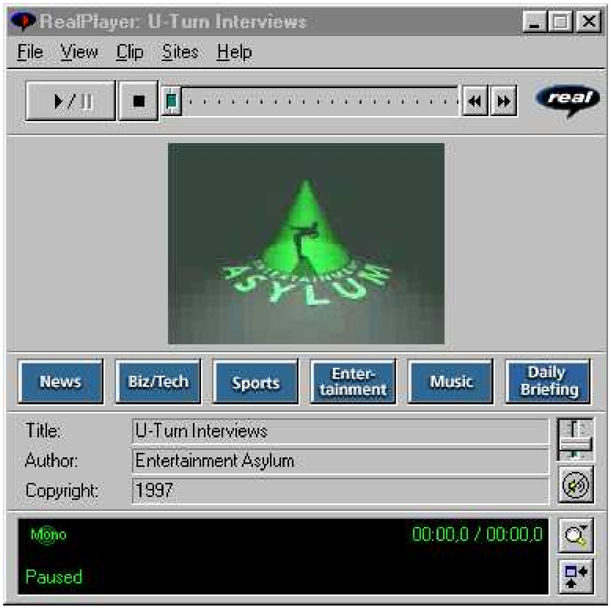
Kurz vor Bearbeitungsende dieser Arbeit wurde die Software noch um einen weiteren Codec ergänzt, der speziell für Trickfilme gedacht ist. Die Ergebnisse sind dabei in puncto Bildqualität und Framerate bedeutend besser. Es ist nicht bekannt, wie das Codierungsverfahren arbeitet, aber ich vermute, daß die Animationen schon in diesem speziellen Format editiert werden müssen. Das Verfahren scheint eine Mischung aus Bitmapgrafik (verlustlos komprimiert) und Vektorgrafik zu sein (siehe Abbildung 5.3.1).

5.3.2 VDOLive

Hier waren keine Informationen über das Übertragungsverfahren zu finden. Es scheint auch hier ein DCT-basiertes Verfahren benutzt zu werden. Im Unterschied zu RealVideo hat man das Gefühl, daß hier die Qualität bei abnehmender Übertragungsbandbreite kontinuierlicher abnimmt.



(a) Statistikfenster



(b) Darstellungsfester

Abbildung 5.1: RealVideo

Abbildung 5.2: RealVideo, spezieller Zeichentrick-Codec

Kapitel 6

Protokollbeschreibung und Programmaufbau

Die im Rahmen dieser Arbeit entwickelte Software soll es ermöglichen, eine Videoübertragung über die bestehende Infrastruktur des Internets von einem Server zu einem (oder nicht allzu vielen) Clients herzustellen. Durch diese Beschränkung auf wenige Clients ist es vertretbar, einen höheren Verwaltungsaufwand beim Server zur Verbesserung der Übertragungsqualität in Kauf zu nehmen. Es wurde aber darauf geachtet, daß der Programmaufbau für eine Erweiterung auf Multicasting geeignet ist. Der vorgestellte Serveralgorithmus ist allerdings nicht für Multicasting gedacht.

6.1 Grundlegender Aufbau

Die Übertragung des Videostreams basiert auf dem UDP-Protokoll der TCP/IP-Protokollfamilie. Nur zur Initialisierung und zum Senden von Kommandos vom Client zum Server (Start, Stop, Pause) werden vereinzelt TCP-Pakete verschickt. Es wurde versucht, die grundlegenden Transportmechanismen möglichst unabhängig von einem speziellen Videocodierungsstandard zu halten.

In der Implementierung wurde das MPEG-Codierungsverfahren benutzt, da dafür zahlreiche Encoder und Decoder verfügbar sind und viel Videomaterial in diesem Format vorliegt.

6.1.1 Grundlagen des Übertragungsprotokolls

Bei Übertragung über UDP gibt es keine Möglichkeit, den Paketen Prioritäten in der Weise zu geben, daß der Paketverlust mit steigender Priorität abnimmt. Dieses Hilfsmittel ist aber für die Übertragung von Videodaten sehr nützlich. Würden zum Beispiel die Bilder einer Bildsequenz jeweils in einem Paket geschickt werden und hätten diese immer abwechselnd hohe und niedrigere Prioritäten, so wäre es wahrscheinlich, daß beim Client bei Paketverlusten wenigstens noch ein Video mit der halben Framerate ankommt, was besser ist, als wenn z.B. immer abwechselnd eine ganze Sekunde komplett übertragen wird und daraufhin immer eine ganze Sekunde fehlt.

Das in dieser Arbeit beschriebene Protokoll versucht, eine prioritätsgesteuerte Übertragung zu simulieren, indem der Server Pakete mit höherer Priorität früher sendet. Nur wenn noch genügend Netzwerkbandbreite vorhanden ist, werden die Pakete mit niedrigerer Priorität gesendet.

Dieses macht nur Sinn, wenn der Server weiß, welche Pakete der Client schon empfangen hat und welche verlorengegangen sind. Deshalb schickt der Client dem Server regelmäßig, welche Pakete er nicht bekommen hat. Der Server entscheidet dann, ob er die Pakete nochmals nachsendet, oder ob schon neue Pakete mit höherer Priorität gesendet werden müssen, denen er dann den Vorzug gibt.

6.1.2 Der Verbindungsaufbau

Der Server erzeugt nach dem Start ein TCP-Socket an einer bekannten Portadresse oder wählt sich selbst einen freien Port. An diese Adresse kann nun ein Client eine Anfrage auf Verbindung schicken. In dieser Anfrage können weitere Informationen, wie z.B. der Name des gewünschten Videostreams übergeben werden.

Der Server startet nun einen neuen Serverprozeß, der alleinig für die Kommunikation mit diesem Client zuständig ist. Dieser Prozess startet daraufhin wieder je nach Art des Streams (Video allein, Audio allein, beides oder unterschiedliche Codierungsverfahren) entsprechende Encoder-Threads und erzeugt für jeden Encoder sowohl einen Daten-, wie auch einen Kontrollkanal in Form eines UDP-Sockets. Zusätzlich wird noch ein weiterer UDP-Socket für globale Kontrolldaten erstellt. Die Adressen dieser Sockets, sowie die Information, welcher Encoder daran angekoppelt ist, wird dem Client über die TCP-Verbindung geschickt.

Dieser wiederum erzeugt nun ebenfalls für jeden Encoder ein zugehöriges Paar UDP-Sockets und startet die jeweiligen Decoder als Threads. Die Adressen dieser Sockets werden dem Server über die TCP-Verbindung geschickt. Damit kann die eigentliche Übertragung beginnen (s. Abbildung 6.1). Tatsächlich werden noch etwas mehr Informationen ausgetauscht. So teilt der Client dem Server seine Eingabebuffergrößen und die maximal zu erwartende Übertragungsrate mit und die Server- und Clientuhr werden synchronisiert.

Aus der maximal zu erwartenden Übertragungsrate kann der Server z.B. noch vor der Übertragung aus mehreren codierten Videos sich dasjenige aussuchen, dessen Datenrate am nächsten an dieser Übertragungsrate liegt. Dies ist sicher einfacher, als zu versuchen, in Echtzeit drastisch die Datenrate eines qualitativ hochwertigen Videos zu reduzieren.

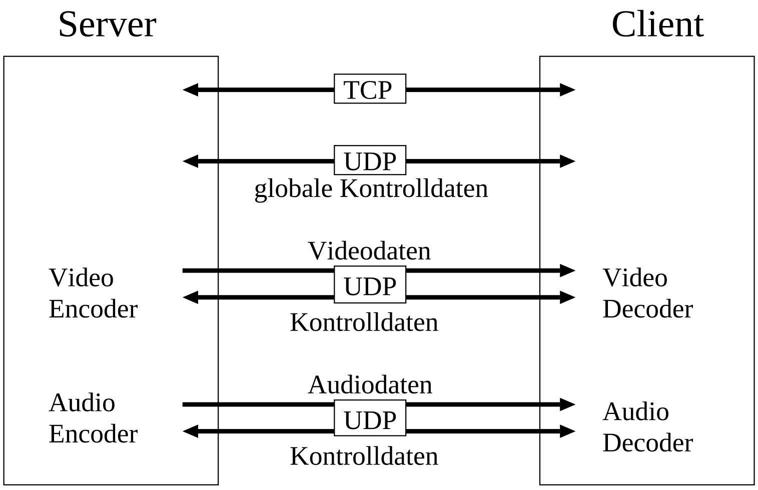


Abbildung 6.1: Netzwerkverbindungen für eine Übertragung von Audio- und Videodaten

Dieses Schema lehnt sich an die Definition des Internet-Standards RFC 1889 an. Die Vorteile, für jeden Streamtyp einen eigenen Port zu benutzen, treten dann hervor, wenn man ein Multicasting betreibt, also derselbe Stream an mehrere Clients geschickt werden soll. Kann ein Client die erforderliche Datenrate für eine Audio/Video-Verbindung nicht empfangen, so kann er sich dafür entscheiden, nur die Audio-Daten zu empfangen. Würden alle Daten über einen Port geschickt, müßte er stets den gesamten Datenstrom empfangen, um dann die Audiodaten herauszufiltern. Wird hingegen jeder Client individuell vom Server beliefert, so lohnt sich der Mehraufwand nicht.

6.2 Der Server

In Abbildung 6.2 ist eine grobe Übersicht des Serveraufbaus dargestellt. Zu sehen ist der Server-Main-Prozess, der, wie oben geschildert, die Verbindung aufbaut, aber während der eigentlichen Übertragung keine Funktion hat.

Die Daten werden von den Encodern erzeugt, von denen es beliebig viele geben kann. Die Encoder liefern ihre Daten an den Server-Algorithmus, der den Serverbuffer verwaltet, die Pakete schließlich verschickt und nicht empfangene Pakete gegebenenfalls nachschickt.

Gestrichelt eingezeichnet ist noch der Statistik-Teil, der anhand der Menge der gesendeten und (beim Decoder) empfangenen Daten ermittelt, mit welcher Datenrate gesendet werden soll und welche Größe ein Paket nicht überschreiten sollte, um mit hoher Wahrscheinlichkeit anzukommen. Diese Information wird auch an die Encoder weitergeleitet, die den Eingabestream dann gleich mit einer passenden Bitrate codieren können.

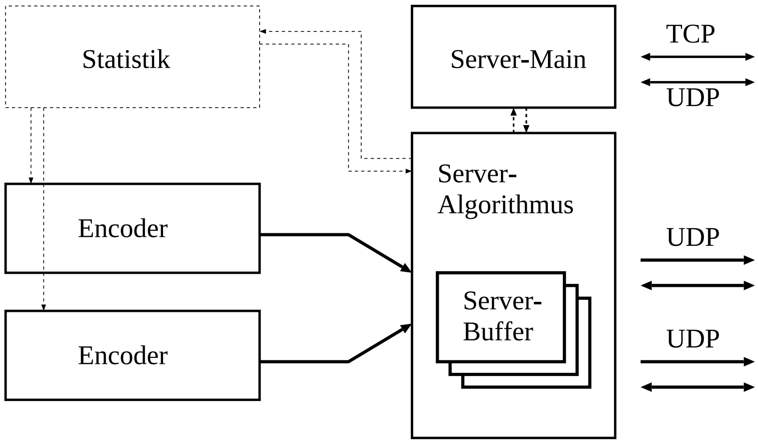


Abbildung 6.2: Grober Aufbau des Servers

Die Organisation ist bewußt modular aufgebaut. Es ist problemlos möglich, neue Encodertypen für andere Codierungsverfahren oder andere Server-Algorithmen z.B. für andere Netzwerktypen zu definieren und einzufügen oder gegen die alten auszutauschen. Diese Modularität setzt sich auch in den feineren Strukturen fort.

6.2.1 Die Encoder

Die oben dargestellten Encoder können in weitere Einzelteile zerlegt werden (Abbildung 6.3). Der Teil des Encoders, der vom Server erzeugt wird, hieß aus technischen Gründen in der ersten Programmversion *EncoderControl*. Dieser Name ist wegen der Kompatibilität beibehalten worden. Als *Encoder* wird im Programm das Objekt bezeichnet, das die Daten erzeugt (oder einliest) und zum richtigen Zeitpunkt (d.h. in Echtzeit) an den EncoderControl weitergibt.

Eine Datenquelle muß nicht immer explizit vorhanden sein, sie kann auch in den Encoder integriert werden. Werden MPEG-System-Streams als Eingabe verwendet, bietet es sich aber an, die Datenquelle (die in diesem Fall das Eingabefile zerlegt) vom Encoder zu trennen. Sonst müßten nämlich der Video- und der Audioencoder das Eingabefile jeweils getrennt voneinander lesen und den Audio- bzw. Videoteil überspringen. Effizienter ist dagegen eine Datenquelle, die sowohl Audio wie auch Video einliest und die Daten für den jeweiligen Encoder bereitstellt.

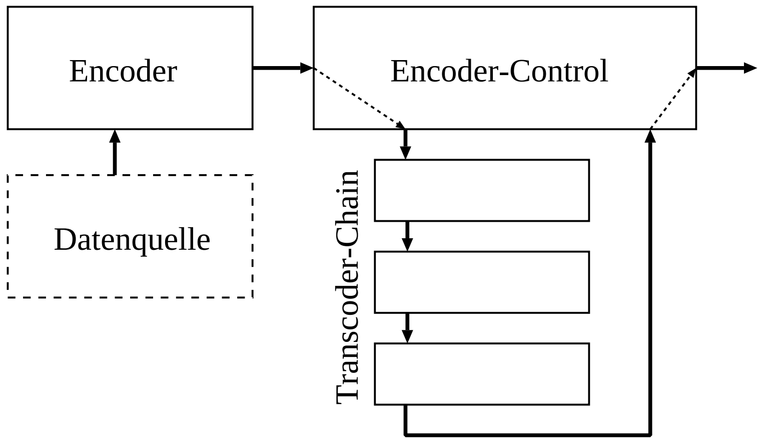


Abbildung 6.3: Encoderaufbau

Die Daten gelangen von der Datenquelle nicht sofort zum Server, sondern werden zuerst durch eine Schlange von *Transcodern* geleitet. Jeder Transcoder kann den Stream in einer bestimmen Weise verändern. So existieren Transcoder, welche die Pakete mit Prioritäten versehen können. Andere erzeugen Zusatzinformationen für den Decoder. Auch ganz komplexe Vorgänge, wie das Umcodieren auf eine andere Bitrate oder die Umcodierung für einen Scalability-Mode von MPEG ist möglich. Weitere Transcoder können auch als Module eingefügt und durch andere ausgetauscht werden.

6.3 Der Client

Abbildung 6.4 zeigt den Aufbau des Clients. Der *Receiver*-Block kümmert sich um alle Netzwerkanbindungen und leitet ankommende Pakete an die zugehörigen *Receiver-Algorithmen* weiter. Außerdem übernimmt der Receiver noch Aufgaben wie die Uhrensynchronisation und zentrale Steuerungen.

Für jeden Server-Encoder wird auf Clientseite ein zugehöriger *Receiver-Algorithmus* und ein zugehöriger *Decoder* erzeugt. Anstatt getrennter Decoder kann es auch sinnvoll sein, nur einen Decoder für mehrere Streamtypen zu verwenden. Ein MPEG-System-Decoder kann so z.B. die Daten von dem Video- und Audiostream gemeinsam decodieren, was den Vorteil hat, daß damit eine synchrone Decodierung einfacher realisierbar ist. Weiterhin gibt es noch einen Statistik-Teil, der unter anderem die empfangene Datenrate mißt und diese Meßwerte dem Server mitteilt.

Empfangene Transmissionpakete[1] leitet der Receiver zum einen an den Statistikteil weiter, der damit seine Statistik erweitert, nach dem Aufteilen der Daten in Encoderpakete aber natürlich auch zum Receiver-Algorithmus, der das direkte Gegenstück zum Serveralgorithmus darstellen muß. Der Receiver-Algorithmus liefert dann die Daten (ggf. nach einem kurzen Aufenthalt im Empfangsbuffer) an den Decoder, der die Daten darstellt. Die Kontrolldatenpakete werden auch zum Receiver-Algorithmus weitergeleitet, erreichen aber nicht den Decoder.

An den Receiver-Algorithmus können weitere Teil-Algorithmen angehängt werden, die ihm einige Arbeit abnehmen. So kann z.B. ein *Reception-Log*-Objekt eingehängt werden, das verlorengegangene Pakete nachfordert. Dieser Aufbau ist ähnlich der Transcoder-Chain im Encoder, ist aber nicht das entsprechende Gegenstück auf Clientseite.

[1]Siehe Abschnitt 6.5.1 für eine genaue Erklärung der verschiedenen Pakettypen

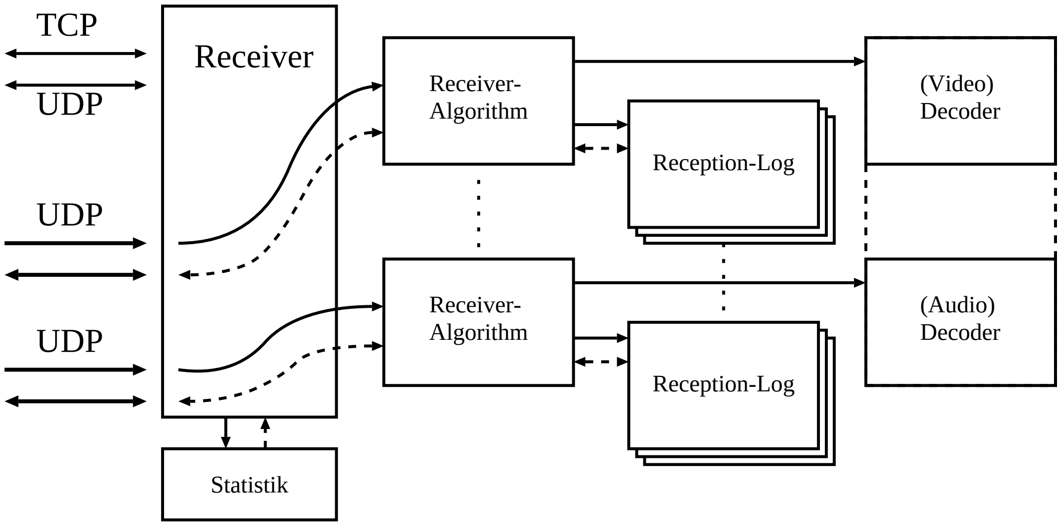


Abbildung 6.4: Grober Aufbau des Clients. Durchgezogene Pfeile stellen den Fluß von Daten dar, gestrichelte Pfeile den Fluß von Kontrolldaten.

6.3.1 Erzeugen eines gültigen Bitstreams

Da im empfangenen Bitstream Pakete fehlen werden, muß sich aus den empfangenen Daten nicht immer ein gültiger Bitstream ergeben. Manche Daten werden sich z.B. durch Interpolation ersetzen lassen, andere sind aber unwiderruflich verloren. Da von diesen Daten andere Daten abhängen können, die schon empfangen wurden, sind auch diese Daten nutzlos. Diese dürfen somit nicht an den Decoder geschickt werden.

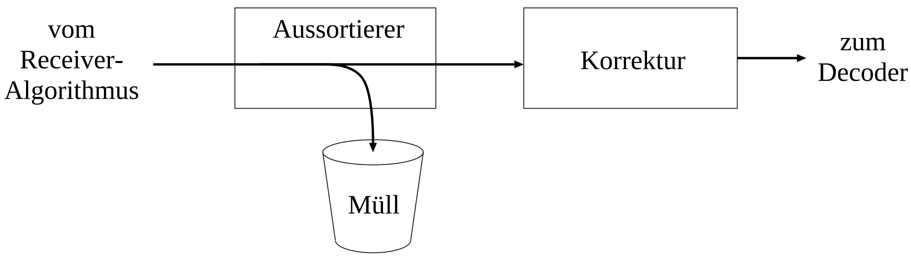


Abbildung 6.5: Korrektur des empfangenen Bitstreams

Abbildung 6.5 zeigt die beiden benötigten Funktionsblöcke, die zwischen Receiver-Algorithmus und Decoder angesiedelt sind. Oft ist es jedoch praktischer, diese Teile mit dem Receiver-Algorithmus oder Decoder zusammenzufassen. In der vorhandenen Implementation ist der *Aussortierer* in den Receiver-Algorithmus integriert. Die Ausgabe ist dann ein gültiger Videostream, in dem aber Frames oder auch nur Slices fehlen können. Fehlende Slices versucht dahingegen der Decoder durch sinnvolle Daten zu ersetzen.

6.4 Allgemeines

6.4.1 Synchronisation von Server und Client

Sowohl Server als auch Client verarbeiten die Daten in Echtzeit. Dazu besitzen Server und Client jeweils eine Echtzeituhr, die das pünktliche Absenden von Paketen und Decodieren der Daten ermöglichen. Das Übertragungsprotokoll setzt an mehreren Stellen voraus, daß der Server die Clientuhrzeit kennt. Es wird also ein Algorithmus zur Synchronisation beider Uhren benötigt. Dabei kommt es nicht auf extreme Genauigkeit an (eine von der Buffergröße anhängige Abweichung von etwa 500ms ist durchaus akzeptabel). Da jeweils die Echtzeituhr verwendet wird, kann von einer ausreichenden Genauigkeit ausgegangen werden, allerdings muß man die absolute Zeit dennoch ggf. anpassen, wenn sich die absolute Uhrzeiten wesentlich unterscheiden.

Angepaßt wird grundsätzlich die Client-Uhr, da der Client immer nur mit *einem* Server in Verbindung steht. Ein Server kann jedoch (zumindest in Erweiterungen dieser Implementation) mehrere Clients zugleich bedienen.

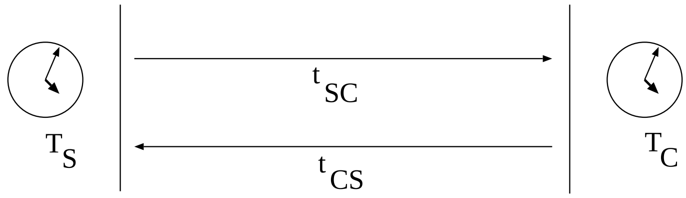


Abbildung 6.6: Uhrzeiten und Übertragungszeiten

In Abbildung 6.6 sind die bei der Übertragung auftretenden Zeiten eingetragen. Dabei ist T_S die absolute Serverzeit, T_C entsprechend die absolute Clientzeit, t_{SC} die mittlere Übertragungszeit vom Server zum Client und t_{CS} die mittlere Übertragungszeit vom Client zum Server. Oftmals kann eine symmetrische Verbindung angenommen werden, bei der $t_{SC} \approx t_{CS}$ gilt. Unter der *Round-Trip-Time* t_{RT} verstehen wir die mittlere Zeit, die benötigt wird, um ein Paket vom Server zum Client und von dort sofort wieder zum Server zu schicken: $t_{RT} = t_{SC} + t_{CS}$.

Man beachte, daß die angebenen Übertragungszeiten t_{RT}, t_{SC}, t_{CS} i.A. abhängig von der übertragenen Paketgröße sind, was dann immer durch $t_i(n)$ dargestellt werden soll. Wird keine Paketgröße angegeben, so soll im folgenden immer die minimale Übertragungszeit gemeint sein.

Die Uhren werden so eingestellt, daß immer $T_S \approx T_C + t_{SC}$ gilt. Damit kann man dann das Senden vom Server zum Client als Vorgang betrachten, der keine Zeit (bzw. nur die Differenz von wahrer Übertragungszeit zu minimaler Übertragungszeit) benötigt.

Implementierung

Noch bevor die Übertragung des Videostreams beginnt, schickt der Server seine augenblickliche Uhrzeit an den Client. Dieser setzt daraufhin seine Uhr auf diesen Zeitwert und schickt dem Server die Rückmeldung, daß seine Uhr gesetzt wurde. Erst wenn der Server diese Rückmeldung empfängt, startet er mit der Übertragung des Videostreams.

Sind sehr lange Übertragungen erforderlich, so kann man den Gleichlauf verbessern, indem jedesmal, wenn der Client mit einem Transmissionpaket die Serverzeit geliefert bekommt, die Differenz von Serverzeit zu Clientzeit gebildet wird und man die Clientzeit um einen Bruchteil dieser Zeit korrigiert. Dabei sollte noch ein Maximalwert von wenigen Millisekunden angegeben werden, um den in einem Schritt korrigiert wird. Dies ist ausreichend, da die Echtzeituhren als sehr genau betrachtet werden können. Der Bereich Δt, in dem die Paketlaufzeiten schwanken, kann nämlich so groß sein, daß die Decodierung nicht mehr flüssig wäre, würde man die Clientuhr einfach um Δt umstellen.

6.5 Die Serveralgorithmen

Der Serveralgorithmus ist für das Versenden der Daten, sowie weiterer Kommunikation mit dem Server zuständig, zu dem auch das Nachsenden von verlorenen Paketen zählt. Die zu sendenden Daten erhält er von den Encoder-Threads in Echtzeit. Wann er diese Daten schickt und wie lange er sie im Sendebuffer hält, kann der Algorithmus selbst entscheiden.

Wie oben erwähnt, können verschiedene Arten von Serveralgorithmen ausgewählt werden. So existiert für Testzwecke ein *SendOnce*-Algorithmus, der alle Pakete, die er von einem Encoder erhält, direkt ohne Zwischenspeicherung an den Decoder schickt. Dieser Algorithmus soll nicht weiter beschrieben werden.

6.5.1 Theorie des Priority-Queue Serveralgorithmus

Ein entwickelterer Serveralgorithmus ist der *Priority-Queue Algorithmus*. In diesem sind die Konzepte des prioritisierten Versands und des Nachsendens verlorener Pakete enthalten. Er geht davon aus, daß den Encoderpaketen schon Prioritäten zugeordnet wurden, was z.B. in einem Encoder-Transcoder geschehen kann. Diese Encoderpakete werden in einen Sendebuffer eingetragen. In regelmäßigen Abständen werden Encoderpakete aus diesem Sendebuffer gelesen und in ein größeres *Transmissionpaket* zusammengefaßt, das dann an den Client verschickt wird. Ein Transmissionpaket ist also größer als ein Encoderpaket. Encoderpakete im Sendebuffer, die nicht mehr verschickt werden können, so daß der Decoder sie noch rechtzeitig erhält, werden aus dem Sendebuffer entfernt. Die Sendepakete werden fortlaufend durchnumeriert. Merkt der Client an fehlenden Nummern, daß Pakete verlorengegangen sind, so schickt er eine Nachricht, welche Pakete er nicht bekommen hat. Diese schickt dann der Server sofort noch einmal.

Es sei noch einmal ausdrücklich darauf hingewiesen, daß im Server vier verschiedene Arten von Paketen unterschieden werden.

- **Datenpakete** sind die Rohdaten, die dem Encoder als Eingabe dienen. Sie bestehen genau aus den Daten der Eingabe, die nur an ausgezeichneten Stellen aufgetrennt wurden.

- **Encoderpakete** enthalten die Datenpakete, zusätzlich aber weitere Informationen, wie eine fortlaufende Sequenznummer, einen Presentation-Time-Stamp (PTS), der nicht unbedingt dem MPEG-PTS entsprechen muß und hier auch in der Einheit *Millisekunden* angegeben wird, und eine Priorität. Diese Pakete werden, wie der Name schon sagt, vom Encoder aus den Datenpaketen erzeugt. Der Encoder muß nicht alle zusätzlichen Felder ausfüllen. Er kann sie auch nur mit Defaultwerten initialisieren. So wird die Sequenznummer z.B. *immer* erst kurz vor der Übergabe an den Server eingetragen. Es bietet sich an, die Prioritäten von einem Transcoder verteilen zu lassen, da der Datenstrom durch andere Transcoder umcodiert werden könnte.

- **Kontrollpakete** entsprechen eigentlich den Encoderpaketen, enthalten aber keine Daten des Eingabestreams, sondern zusätzliche Informationen, die dem Decoder bei der Decodierung helfen sollen, wie z.B. Informationen zur Fehlerkorrektur.

- **Transmissionpakete** werden vom Serveralgorithmus erstellt. Sie enthalten i.A. mehrere Encoder- und Kontrollpakete. Dies geschieht aus Effizienzgründen, da Encoder- und Kontrollpakete unter Umständen sehr klein sein können. Auch Sendepakete enthalten eine fortlaufende Numerierung in ihrer *TransmissionNr.* Es besteht aber kein Zusammenhang zwischen den Transmissionnummern und den Sequenznummern der Encoderpakete.

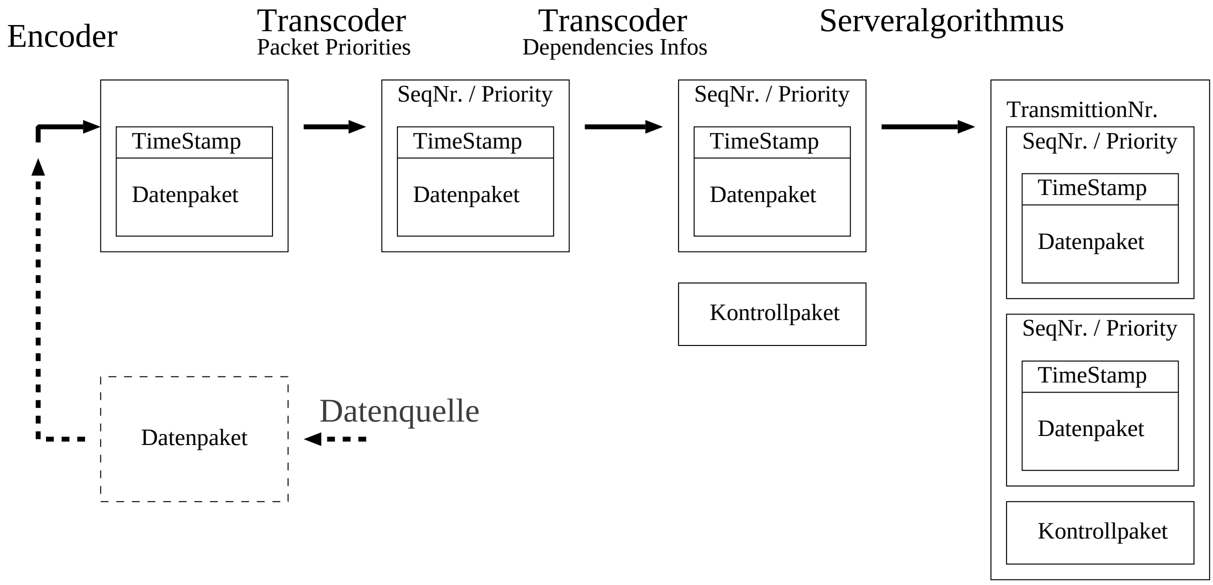


Abbildung 6.7: Datenpipeline

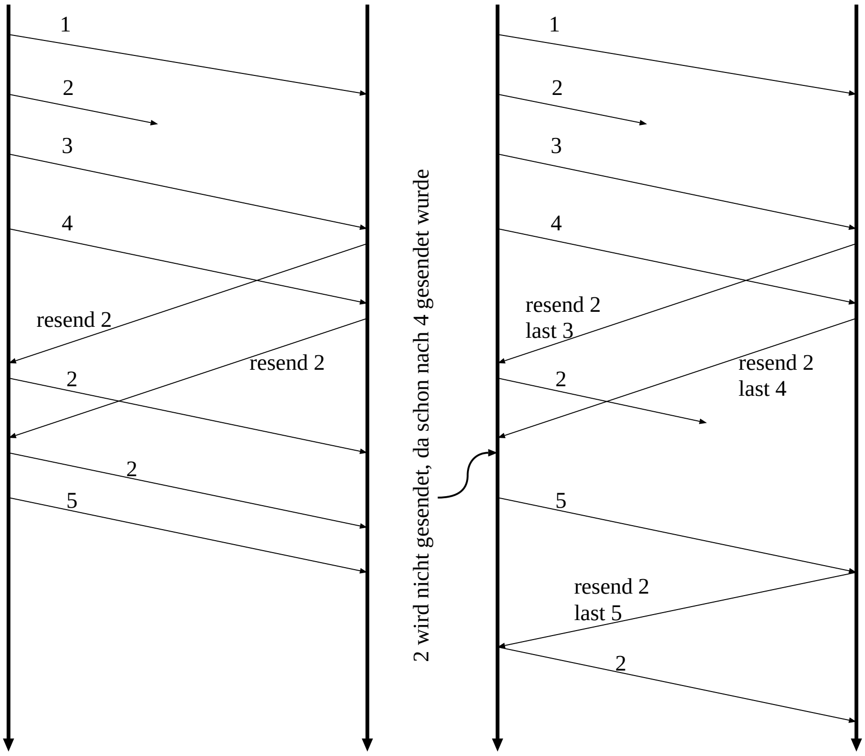
6.5.2 Packet-Resending

Der Server schickt die Sendepakete mit lückenloser, aufsteigender Numerierung. Da UDP-Pakete nur sehr selten in einer anderen Reihenfolge als der Sendereihenfolge beim Empfänger ankommen, kann der Client somit annehmen, daß Pakete fehlen, wenn eine oder mehrere Nummern in den ankommenden Paketen übersprungen wurden. Diese fehlenden Paketnummern werden an den Server geschickt, um diesem mitzuteilen, daß er diese nochmals schicken soll. Um dem Client die Arbeit zu erleichtern, ist in jedem Transmissionpaket ein Flag enthalten, das gesetzt ist, wenn das Paket schon zum wiederholten Male geschickt wurde.

Der Client schickt nach Empfang eines Paketes jedesmal eine *ReceptionReport*-Message an den Server. In dieser ist vermerkt, welche Pakete er bekommen hat und welche ihm noch fehlen. Der genaue Aufbau dieser Message ist in Abschnitt 6.10.4 beschrieben.

In der ReceptionReport-Message wird außer den Nummern der fehlenden Pakete auch noch die Nummer des letzten empfangenen Pakets angegeben. Mit dieser Information kann man verhindern, daß der Server fehlende Pakete doppelt nachschickt, indem der Server ein fehlendes Paket nur dann nochmals schickt, wenn er es nach dem Verschicken von *LastNr* noch nicht gesendet hat. Würde der Server es jedesmal nachschicken, könnte es vorkommen, daß Pakete, die nur einmal verlorengingen, zweimal nachgeschickt werden. Siehe dazu Abbildung 6.8.

Wenn das Paket so lange nicht beim Client ankommt, daß zwischen dem Erkennen des Verlustes und dem aktuellen Zeitpunkt mindestens so viel Zeit vergangen ist, wie der Decoder im Empfangsbuffer hält, so wird das Paket als endgültig verloren betrachtet.



(a) ungünstig (b) besser

Abbildung 6.8: Ablauf eines wiederholten Paketversands

6.5.3 Server- und Clientbuffer

Im MPEG-Standard wird ein Bufferprinzip definiert, das dazu dient, die variable Bitrate des MPEG-Streams, wenn er den Encoder verläßt, zu glätten und damit in einen Bitstream mit konstanter Bitrate umzuwandeln. Das Prinzip ist in Abbildung 6.9 dargestellt. Die Daten werden mit einer nicht konstanten Bitrate in den Server-Buffer geschrieben, von dort in der gleichen Reihenfolge, aber mit einer konstanten Bitrate an den Client geschickt, dort wieder gepuffert und mit einer nicht konstanten Bitrate ausgelesen. Bei einer Zwischenspeicherzeit von t_B in jedem Buffer hat man somit eine Gesamtverzögerung von $2t_B$.

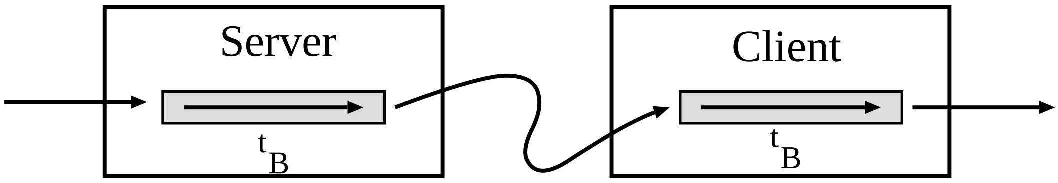


Abbildung 6.9: Buffergebrauch bei MPEG

Durch dieses Prinzip kann ein Bitstream prinzipiell auch an ein Netzwerk mit variabler Bitrate angepaßt werden. Dann wird auch zur Übertragung zwischen Server und Client eine variable Bitrate benutzt. Das Verfahren hat aber den Nachteil, daß bei einer Übertragungskapazität unter der benötigten Bitrate nach einer Weile die übertragene Datenmenge von der Soll-Menge so weit abweicht, daß eine Echtzeitdarstellung nicht mehr möglich ist. Das gleiche gilt für unzuverlässige Netzwerke. Entweder man schickt verlorene Pakete so lange, bis sie ankommen, was die Geschwindigkeit unter Echtzeit drücken kann, oder man muß Paketverluste in Kauf nehmen.

In dieser Arbeit wurde ein anderes Bufferprinzip verwendet, das zwar dem oberen sehr ähnelt, sich aber doch grundlegend anders verhält (s. Abbildung 6.10). Auch hier werden die Daten mit variabler Bitrate in den Serverbuffer geschrieben und aus dem Clientbuffer ausgelesen. Ebenso wird die Server-Client Übertragung mit variabler Bitrate realisiert. Der grundlegende Unterschied liegt aber darin, daß die Daten nicht mehr sequentiell in der gleichen Reihenfolge geschickt werden, wie sie in den Buffer eingefügt wurden. Es ist also möglich, daß eingefügte Daten sofort oder spätestens nach Ablauf von t_B (oder auch gar nicht) geschickt werden.

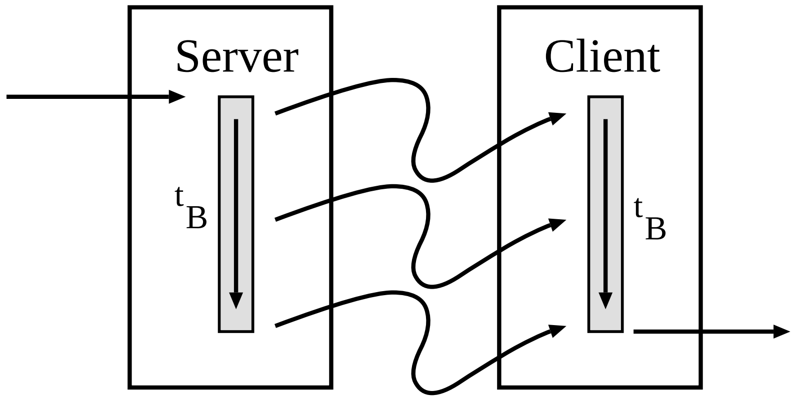


Abbildung 6.10: Zusammenspiel des Server- und des Clientbuffers

Dadurch ist es nun möglich, daß der Client verlorengegangene Pakete nachfordert. Wird ein Paket zur Zeit T_0 empfangen und soll es zur Zeit T_D decodiert werden, so kann es der Client im Falle eines Verlustes noch maximal nach Ablauf von $t_{RS} = T_D - T_0 - t_{RT}$ nach Bemerken des Verlustes nachfordern. Der letzte Term beschreibt die Zeitdauer, die der ResendRequest vom Client zum Server und das nachgeforderte Paket zurück zum Client mindestens benötigt.

Da zwischen der Empfangszeit, zu der das Paket eigentlich ankommen sollte und dem Bemerken des Verlustes noch einige Zeit vergehen kann, werden gelegentlich auch dann noch ResendRequests ausgesendet, wenn die Pakete nicht mehr rechtzeitig empfangen werden können. Dies ist aber nicht schlimm, da der Server nur Pakete schickt, für die der spätestmögliche Sendezeitpunkt T_D noch nicht überschritten ist. Der korrekte Ausdruck lautet *nicht* $T_D - t_{SC}$, da die Uhren von Server und Client ja um den Betrag t_{SC} gegeneinander verschoben sind. Dahingegen ist auch der „ganz“ korrekte Ausdruck $T_D - (t_{SC}(n) - t_{SC})$ nicht geeignet, da $t_{SC}(n)$ nur die mittlere Übertragungszeit angibt. Es kann also durchaus sein, daß das Paket schneller geschickt wird. Auch ist eine exakte Bestimmung von $t_{SC}(n)$ schwierig. Die einfachste Lösung von T_D als spätestem Sendezeitpunkt ist somit auch am besten geeignet. Man beachte, daß die Bufferverzögerung nun nicht mehr $2t_B$, sondern nur noch t_B beträgt.

6.5.4 Prioritätsgesteuerter Versand

Ein Beispiel soll verdeutlichen, was wir unter einem *prioritätsgesteuertem Versand* verstehen wollen. Dazu betrachten wir ein stark vereinfachtes Beispiel einer Übertragung. Abbildung 6.11 soll den Sendebuffer des Servers darstellen. Jede Spalte des Gitters repräsentiert die Daten eines Frames.

Alle Frames haben also genau die gleiche Datenmenge. Die Frames sind in weitere Teile unterteilt (horizontale Linien), die den jeweiligen Anteil der Daten mit unterschiedlichen Prioritäten darstellen soll. Ein Frame ist hier also in 5 Teile unterteilt, wobei der Teil mit der höchsten Priorität doppelt soviel Daten umfaßt.

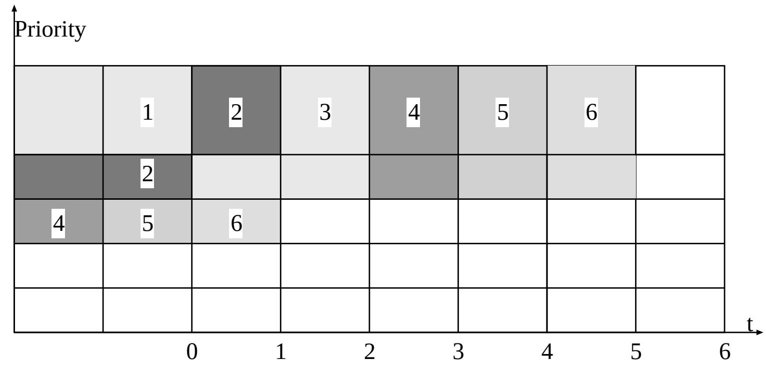


Abbildung 6.11: Übertragung mit 2/3 der benötigten Bitrate

Zu Anfang befinden sich im Buffer nur die Daten bis zur Marke $t = 0$. Der Ablauf sei nun in Schritte unterteilt. In jedem Schritt wird eine bestimmte Teilmenge der Daten verschickt und es wird das nächste Frame in den Sendebuffer eingefüllt. In der wirklichen Implementation sind diese beiden Tätigkeiten allerdings nicht so fest gekoppelt, sondern laufen asynchron. Im Beispiel der Abbildung 6.11 werden in jedem Schritt die zwei Quadraten entsprechenden Daten (also 2/3 eines Frames) geschickt. Allerdings werden die Daten nicht Frameweise, sondern in der Reihenfolge absteigender Prioritäten verschickt. Die in jedem Schritt verschickten Daten sind jeweils in einem anderen Grauton hinterlegt und durchnummeriert.

Man erkennt, daß mit etwas Verzögerung genau die Datenmenge eines Bildes geschickt wird, die der Übertragungsrate entspricht. Von keinem Bild wird jemals mehr als 2/3 seines Inhalts geschickt werden. In Gedanken stellen wir uns nun vor, was passiert, wenn die Datenrate z.B. nach Schritt 2 aufgrund von Netzwerküberlastungen kurzzeitig auf 1/3 der Framedaten sinken sollte: Es wird nur noch der Teil der Daten mit den höchsten Prioritäten übertragen. Sollte die Datenrate dann aber wieder steigen, können die Daten mit den niedrigeren Prioritäten wieder nachgeliefert werden.

Im Mittel wird also immer genau der Teil der Frames gesendet werden, der der gegenwärtig möglichen Übertragungsrate entspricht. Im Beispiel der Abbildung 6.12 sei die Übertragungsrate entsprechend des gesamten Umfangs eines Frames pro Framedauer. Auch hier werden schließlich alle Daten übertragen. Wenn wir das Beispiel in Gedanken fortführen, wird im 11. Schritt das erste Frame komplett übertragen sein.

6.5.5 Realisierung des Priority-Queue Algorithmus

Hauptbestandteil des Algorithmus ist ein Heap, in den die Encoderpakete *aller* Encoder so einsortiert werden, daß sich die Pakete mit höchster Priorität oben befinden. In regelmäßigen Zeitabständen werden einige Pakete von der Spitze des Heaps in ein Transmissionpaket verpackt, diesem eine fortlaufende Transmissionnummer zugewiesen und das ganze Paket dann abgeschickt (s. Abbildung 6.13). Wie oft dies geschieht und wie groß die Pakete jeweils sind, wird von der Statistik geregelt. Außerdem wird das Transmissionpaket in einen *Historybuffer* eingefügt, um es im Bedarfsfall bei einem *ResendRequest* nochmals nachschicken zu können. Im Historybuffer liegen diese Pakete mit streng aufsteigender Transmissionnummer. Deshalb ist ein sehr schnelles Auffinden des gesuchten

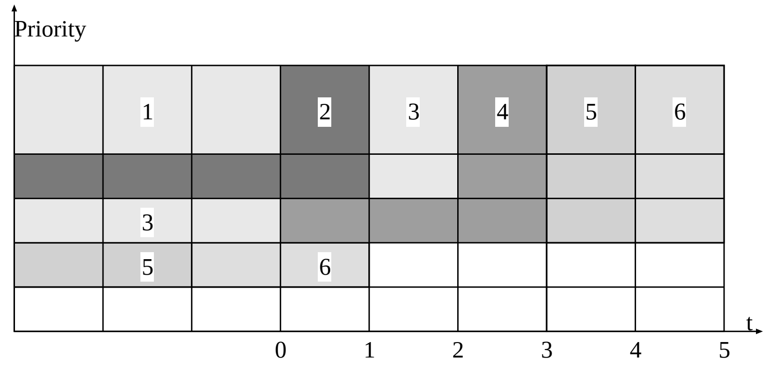


Abbildung 6.12: Übertragung mit der benötigten Bitrate, das 1. Frame würde im 11. Schritt vollständig übertragen

Paketes möglich. Die Transmissionpakete können wieder aus dem Historybuffer entfernt werden, wenn das Paket bei einem ResendRequest nicht mehr vor seiner angegebenen Decodierungszeit den Decoder erreichen kann.

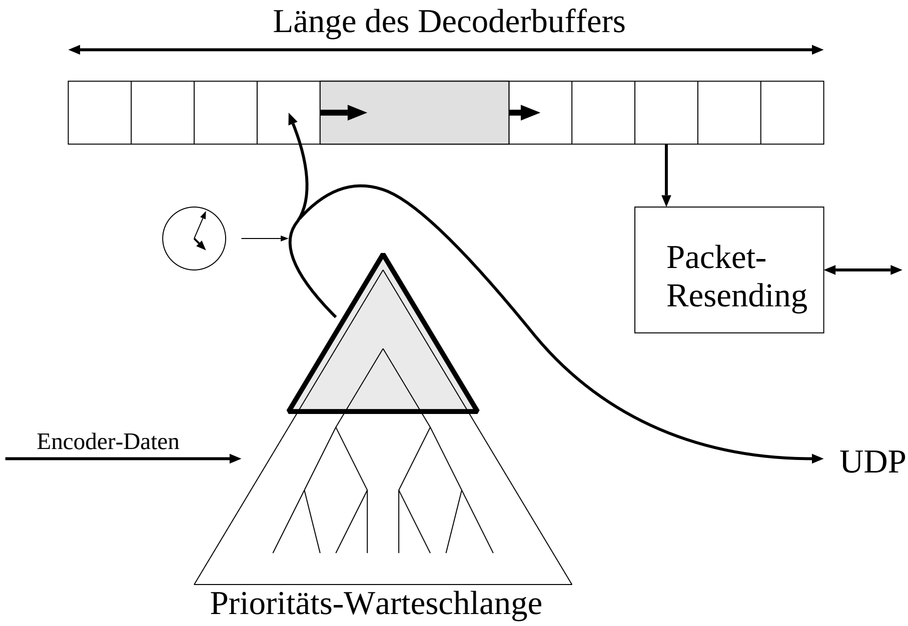


Abbildung 6.13: Einpacken der Encoderpakete in Transmissionpakete und Einsortieren dieser in den Historybuffer

Das in Abschnitt 6.5.3 beschriebene neue Bufferverfahren wird hier also insofern noch etwas verfeinert, daß der Sendebuffer in zwei Teile geteilt ist. Neue Daten kommen in den ersten Teil (den Prioritätsheap) und werden nach dem ersten Senden in den zweiten Teil (den Historybuffer) verschoben.

Dem aufmerksamen Leser wird aufgefallen sein, daß obige Beschreibung nicht beachtet, daß die Daten von verschiedenen Encodern auf verschiedene UDP-Ports verteilt werden müssen. Der dafür erweiterte Algorithmus ist in Abbildung 6.14 dargestellt.

Auch jetzt werden wieder in regelmäßigen Abständen die Daten mit den höchsten Prioritäten

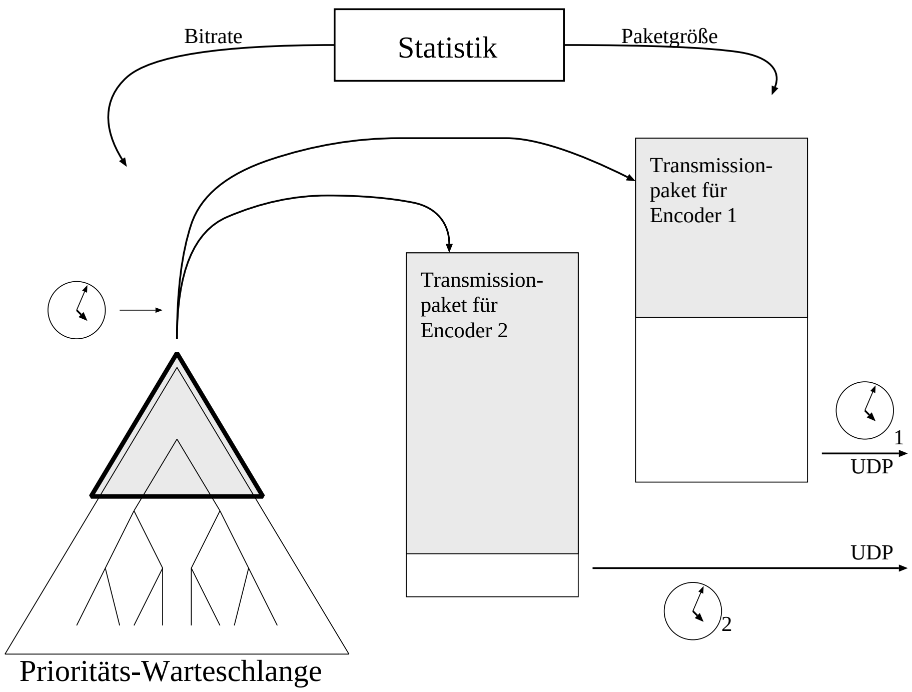


Abbildung 6.14: Erweitertes Einpacken der Encoderpakete in Transmissionpakete

aus der Prioritätswarteschlange genommen, diesmal werden die Daten aber auf mehrere Transmissionpakete verteilt. Für jeden Encoder werden getrennte Transmissionpakete angelegt. Das Auslesen aus der Prioritätswarteschlange kann jedoch nicht Encoder-weise geschehen, da sonst ein Encoder den anderen die verfügbare Datenrate wegschnappen könnte. Erst, wenn die Encoderpakete aus der Warteschlange entnommen wurden, wird nachgeschaut, in welches Transmissionpaket es verschoben werden muß. Aus der zu sendenden Bitrate b und dem Zeitabstand vom letzten Entnehmen von Encoderpaketen T kann ganz einfach gemäß $N = b * T$ der Gesamtumfang der zu entnehmenden Pakete in Bytes N berechnet werden. Die Bitrate wird durch die Statistik bestimmt, die Zeitabstände sind frei wählbar. Größere Zeitabstände bedeutet dabei größere oder mehr Pakete pro Zeiteinheit.

Die Transmissionpakete werden nicht in jedem Schritt abgeschickt. Vielmehr wird gewartet, bis sie die maximale Paketgröße, die von der Statistik bestimmt wurde, erreicht haben. Damit Pakete bei sehr geringem Datenaufkommen aber nicht Gefahr laufen, zu spät abgeschickt zu werden, kann jeder Encoder angeben, in welchen Zeitabständen der Paketversand erzwungen werden soll. Zwischen dem ersten Einfügen in ein Transmissionpaket und dem Absenden vergeht also maximal diese vom Encoder angegebene Zeit.

Aber auch mit dieser Erweiterung ist der Algorithmus noch nicht einsatzfähig. Ist es nämlich wegen einer zu kleinen Übertragungskapazität nicht möglich, den gesamten Bitstream zu schicken, dann sammelt sich die Prioritätswarteschlange mit altem Datenmüll an, der den Decoder nicht mehr rechtzeitig erreichen kann, ja sogar zu Videoteilen gehört, die schon längst decodiert sind. Der Heap muß also gelegentlich von zu alten Daten befreit werden. Da aber der Heap nicht nach Präsentationszeiten sortiert ist, ist das nicht ohne weiteres effizient möglich. Die Lösung besteht darin, einen zweiten Heap zu benutzen, der nach Präsentationszeiten sortiert ist. In beiden Heaps wird in jedem Element eingetragen, wo das entsprechende Element im anderen Heap zu finden ist. So kann auf jedem der beiden Heaps effizient gearbeitet werden. Natürlich enthalten die

Heapeinträge nicht direkt die Encoderpakete, sondern indirekte Verweise auf ein Feld, das nun die Verweise auf die Encoderpakete und zusätzliche Informationen, wie die Encoder-ID enthält. Die beiden Heaps und das Feld mit den Zusatzinformationen ist in Abbildung 6.15 dargestellt. Um Pakete schnell in das Feld mit Zusatzinformationen einfügen und entfernen zu können, sind die freien Einträge durch eine einfach verkettete Liste verknüpft. Beim Einfügen wird einfach das erste Element dieser Liste benutzt und beim Entfernen wird das nun freigewordene Feld wieder vorne an die Liste gehängt.

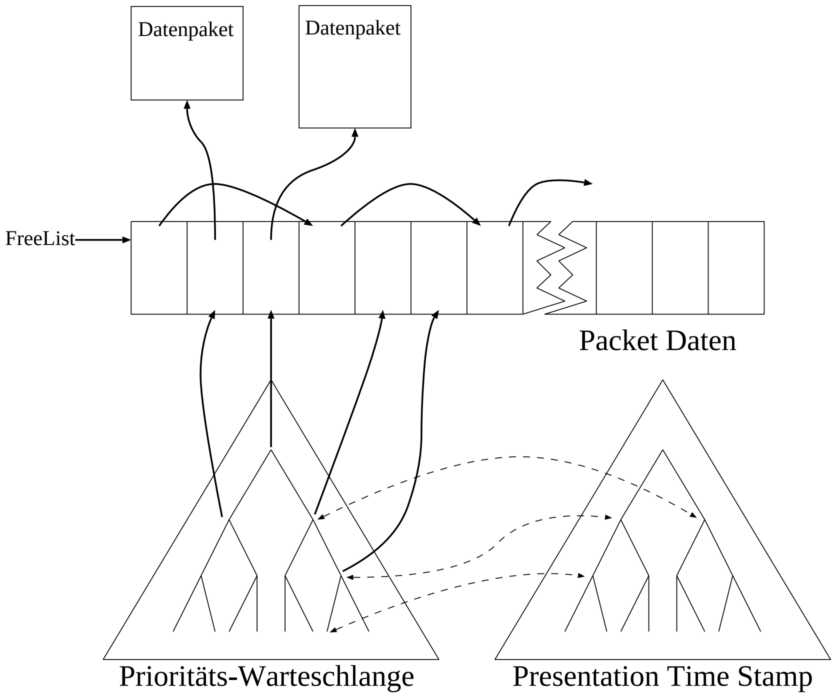


Abbildung 6.15: Effizientes Aussortieren alter Pakete

6.6 Die Transcoder

Jede Codierungsmethode benötigt einen eigenen Satz von Transcodern. Hier werden Transcoder beschrieben, die alleinig auf MPEG codierten Daten arbeiten, eine Anpassung auf andere Codierungsmethoden dürfte aber nicht schwerfallen.

6.6.1 Priority-Transcoder

Der MPEG-Bitstream wird vom Encoder an allen Startcodes in Encoderpakete unterteilt. Diesen Paketen werden in einem Transcoder die Prioritäten anhand zweier Kriterien vergeben. Zum einen direkt nach Typ des Startcodes, zum anderen werden die Frames nochmals weiter unterteilt, um den Playbackjitter bei unvollständigem Versand so minimieren (s. hierzu Kapitel 7.1).

Zuerst einmal werden einfach anhand der Startcodes folgende Prioritäten verteilt:

Sequence-Header	220
GroupOfPictures-Header	210
Picture-Header	200
Slice-Header + Daten	190

Jeweils alle Pakete, die zu einer GOP gehören, werden gesammelt und sobald die GOP komplett ist, wird ein Antijitter-Algorithmus benutzt, der eine bevorzugte Sendereihenfolge berechnet. Die jeweilige Position in der Sendereihenfolge wird nun von den oben verteilten Prioritäten abgezogen. Für den Picture-Header des wichtigsten Bildes (das erste I-Frame der GOP) ergibt sich somit die Priorität 199 und für dessen Daten die Priorität 189. Das nächstwichtige Bild (wahrscheinlich das darauffolgende P-Frame) hat dann die Prioritäten 198 für den Header und 188 für die Bilddaten.

6.6.2 Dependency-Transcoder

Fehlen dem Decoder Pakete, so kann er nicht immer selbst feststellen, ob eine sinnvolle Decodierung möglich ist. Senden wir beispielsweise die Sequenz $IBBPBBP$ und geht uns dabei das mittlere P-Bild verloren, so ist es für ihn unmöglich, zu entscheiden, ob ein P-Bild oder nur ein B-Bild fehlt. Würde ein B-Bild fehlen, könnte er alle anderen Frames ohne Problem decodieren, da aber ein P-Bild fehlt, von dem alle B-Bilder und das folgenden P-Bild abhängen, ist überhaupt keine sinnvolle Decodierung möglich (außer dem I-Frame natürlich). Werden die Daten dennoch decodiert, wären Artefakte, wie Bewegungen von Bildteilen, die sich nicht bewegen sollen, möglich, da in diesem Falle die Motionvektoren auf falsche Referenzbilder angewendet würden.

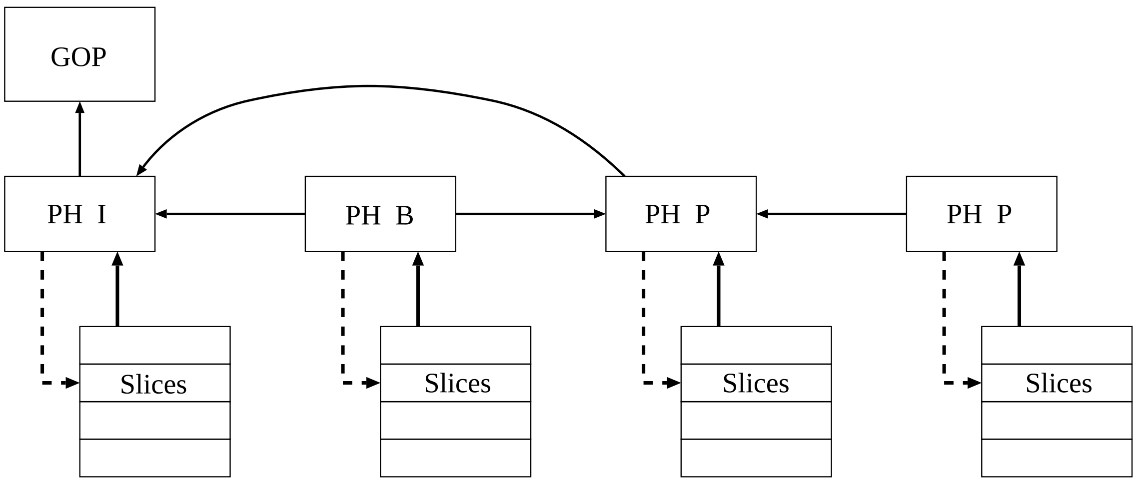


Abbildung 6.16: Datenabhängigkeiten zwischen den Paketen bei MPEG-Codierung

In Abbildung 6.16 sind die Syntaxelemente eines MPEG-Streams dargestellt, die in eigene Encoderpakete verpackt werden. Die durchgezogenen Pfeile (*Require-Abhängigkeit*) zeigen dabei, welche Pakete zur Decodierung eines Pakets unbedingt erforderlich sind. Zwar ist es möglich, z.B. den Picture-Header des P-Frames ohne den Picture-Header des I-Frames zu decodieren, aber es hat überhaupt keinen Sinn, dies zu tun. Eine etwas andere Bedeutung haben die gestrichelten Pfeile (*Require-Some-Abhängigkeit*): Pakete von denen solch ein Pfeil ausgeht werden nur dann decodiert, wenn mindestens ein Paket der Paketmenge, auf die der Pfeil zeigt, vorhanden ist. Das soll hier heißen, daß der Picture-Header eines Frames nur dann decodiert wird, wenn wenigstens ein Slice des Bildes empfangen wurde.

Diese Datenabhängigkeiten werden jeweils für eine GOP in ein eigenes Kontrollpaket geschrieben und an den Client geschickt. Dieses Kontrollpaket besteht aus mehreren gleichartigen Einträgen, im folgenden *Dependency-Slot* genannt. Jeder Slot definiert für mehrere Pakete mit fortlaufender Sequencenummer die Abhängigkeiten. Der Grund dafür, daß immer mehrere Pakete in einem Dependency-Slot zusammengefaßt werden liegt darin, daß ein Frame zuweilen aus recht vielen Slices bestehen kann, die alle die gleichen Abhängigkeiten haben. Werden diese zusammengefaßt, verkürzt sich das Kontrollpaket wesentlich. Ein weiterer Vorteil liegt darin, daß die Require-Some-Abhängigkeiten sich immer auf eine Menge von Paketen beziehen. Da die Pakete dieser Menge immer aufeinanderfolgende Pakete sind, lassen sich die Paketmengen sehr effizient durch einen einzigen Dependency-Slot-Eintrag darstellen. Aus Gründen der Übersichtlichkeit werden alle Abhängigkeiten als Verweise auf Dependency-Slot-Nummern angegeben. Die Nummern der Slots ergeben sich direkt aus der Reihenfolge, in der sie in einem Paket gespeichert sind und müssen somit nicht mitgespeichert werden. Da die Slots über mehrere Pakete hinweg durchnumeriert werden, wird jedoch die Startnummer des ersten Slots im Paket gespeichert. Dies ist notwendig, damit bei einem Paketverlust die Numerierung nicht bei der falschen Nummer beginnt.

(Dependency-Slot-Nr.)	Fortlaufende Nummer des Dependency-Slots
SeqNr.-Start	Erstes Paket, das durch diesen Dep.Slot repräsentiert wird
SeqNr.-End	Letztes Paket, das durch diesen Dep.Slot repräsentiert wird
Require 1	Dep.Slot, dessen Pakete decodiert werden müssen, um diesen hier zu decodieren
Require 2	Weiterer unbedingt benötigter Dep.Slot
Require some	Dep.Slot, von dessen Paketen mindestens eins decodiert sein muß

Betrachten wir wieder ein Beispiel. Es seien die Dependency-Slots für eine GOP zu erstellen, die aus der Framesequenz $I_1B_2P_3B_4P_5$ besteht. In den vom Encoder ankommenden Paketen sind diese Syntaxelemente gespeichert:

SeqNr.	Inhalt
21	GOP-Header
22	Picture Header von I_1
23-28	Slicedaten von I_1
29	Picture Header von P_3
30-35	Slicedaten von P_3
36	Picture Header von B_2
37-43	Slicedaten von B_2
44	Picture Header von P_5
45-51	Slicedaten von P_5
52	Picture Header von B_4
53-58	Slicedaten von B_4

Die zugehörigen Dependency-Slots lauten dann:

Dep.Slot.Nr	SeqNr-Start	SeqNr-End	Require-1	Require-2	Require-Some
1	21	21	-	-	-
2	22	22	1	-	3
3	23	28	2	-	-
4	29	29	2	-	5
5	30	35	4	-	-
6	36	36	2	4	7
7	37	43	6	-	-
8	44	44	4	-	9
9	45	51	8	-	-
10	52	52	4	8	11
11	53	58	10	-	-

Der Grund, daß wir die Dependency-Slot-Nummern nicht bei jeder GOP wieder bei 1 beginnen liegt darin, daß es auch Abhängigkeiten zwischen GOPs geben kann, wie z.B. in der Sequenz $BBIBBPBBP$. Die ersten beiden B-Frames benötigen zur Decodierung das letzte P-Frame der vorhergehenden GOP. Um Mehrdeutigkeiten zu verhindern, muß also fortlaufend durchnumeriert werden.

6.7 Der Receiver-Algorithmus

Der Receiver-Algorithmus im Client hat die Aufgabe, die empfangenen Daten wieder in die richtige Reihenfolge zu bringen und rechtzeitig an den Decoder weiterzureichen. Das Problem ist dabei, den richtigen Zeitpunkt zu finden, an dem die Daten an den Decoder geschickt werden müssen, da die Daten bei MPEG nicht in der zeitlichen Reihenfolge an den Decoder übergeben werden können. Wir erinnern uns daran, daß der Receiver-Algorithmus die Encoderpakete als Eingabe erhält. Diese enthalten einerseits eine Sequenznummer, welche die Reihenfolge angibt, an der die Daten zum Decoder geschickt werden müssen und andererseits einen Presentation-Time-Stamp, der den Zeitpunkt angibt, zu dem die Daten angezeigt werden müssen.

Der Algorithmus muß also, wenn er den Bitstream in der richtigen Reihenfolge, d.h. der Reihenfolge, in der er ihn zum Decoder schicken muß, hat, den frühesten Presentation-Time-Stamp im Bitstream kennen und alle Daten bis einschließlich aller Daten, die diesen Presentation-Time-Stamp haben, zum Decoder schicken. In einer trivialen Implementation ist dies nicht effizient lösbar. Hier wurde der folgende Algorithmus benutzt (siehe Abbildung 6.17 auf der nächsten Seite).

Nach dem Empfang der Daten werden diese in einem Array, das als Ringbuffer dient, in der Sequenzreihenfolge eingefügt. Setzt man voraus, daß alle Daten eines Transmissionpakets mit aufsteigenden Sequenznummern übergeben werden, so muß man den Buffer jedoch nur einmal durchlaufen, was in vertretbarer Zeit möglich ist. Dabei kann man dann alle Pakete auf einmal einsortieren. Da in dem Buffer natürlich nur wieder indirekte Verweise auf die Pakete enthalten sind, ist das Einfügen sehr schnell.

Die Aufgabe ist nun noch, dasjenige Paket mit der frühesten Presentation-Time-Stamp zu finden und, falls es mehrere mit derselben gibt, unter diesen das, das am weitesten hinten ist, d.h. das als letztes zum Decoder geschickt wird.

Beides kann auf einmal wieder mit einem Heap gelöst werden, in dem Verweise auf die Buffereinträge enthalten sind und der in erster Linie nach deren Presentation-Time-Stamps und zweitens

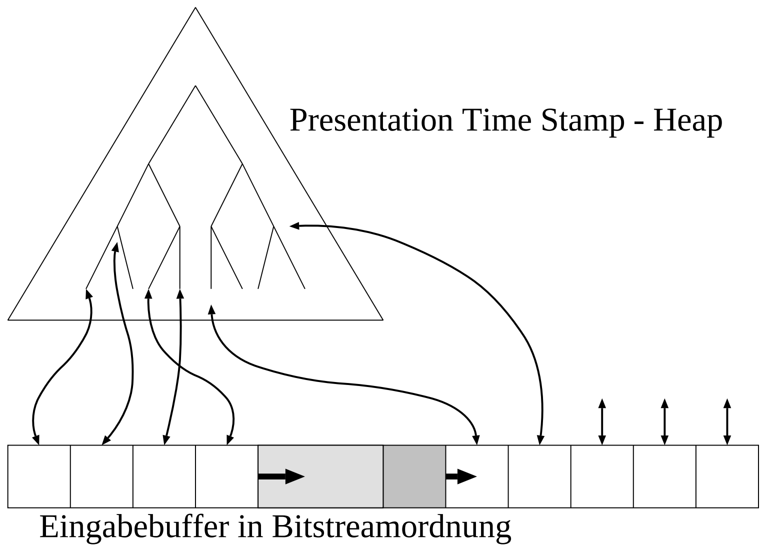


Abbildung 6.17: Receiver-Algorithmus

nach Bufferindizes sortiert ist. Hat ein Paket i die Presentation-Time-Stamp $P(i)$ und die Bufferposition $L(i)$, so definieren wir die Ordnung

$$a < b \iff P(a) < P(b) \ \vee \ (P(a) = P(b) \ \wedge \ L(a) > L(b))$$

In der Heapwurzel steht dabei das minimale Element.

Wir brauchen nun nur noch das Element an der Spitze des Heaps betrachten und haben damit sofort das gewünschte letzte Paket mit dem kleinsten Presentation-Time-Stamp. Jetzt warten wir auf den Zeitpunkt des Presentation-Time-Stamps, der in diesem Paket angegeben ist und schicken dann alle Pakete vom Anfang des Ringbuffers bis einschließlich des Pakets, auf das der Eintrag an der Spitze des Heaps verweist, an den Decoder.[2]

6.8 Fehlerkorrektur

Im folgenden sei beschrieben, wie anhand der Dependency-Kontrollpakete, die von dem Dependency-Transcoder (siehe Abschnitt 6.6.2) in den Datenstrom eingefügt wurden, sichergestellt wird, daß die zum Decoder gesendeten Daten syntaktisch korrekt sind und durch fehlende Bilder nicht aus Versehen falsche Bilder als Prädiktion benutzt werden.

Für jedes Frame gibt es drei Möglichkeiten, wieviele Slices vorhanden sind, die alle unterschiedliche Vorgehensweisen erfordern:

- **keine Slicedaten vorhanden** Das Frame kann weder dargestellt werden, noch können andere Frames dieses als Prädiktion benutzen.
- **alle Slicedaten vorhanden** Auch dieser Fall ist einfach: Das Frame kann dargestellt werden und andere Frames können es als Prädiktion benutzen.

[2]Falls beim Empfang ein Paket mit noch niedrigerer PTS eingefügt wurde, als die, auf die wir gerade warten, so muß das Warten abgebrochen werden und nun auf diese niedrigere PTS gewartet werden.

- **Slicedaten sind nur unvollständig vorhanden** In diesem Fall können wir das Frame anzeigen, was zwar nicht das vollständig richtige Bild ergibt, aber wenigstens werden die vorhandenen Slices mit den aktuellen Daten dargestellt. Aber wir können es nicht als Prädiktion für andere Frames benutzen, da die Slices, die nicht neu dargestellt wurden, falsche Daten für die Prädiktion enthalten.

Das Aussortier-Objekt (*Aussortierer*) benutzt einen Bitvektor, in dem für alle Sequenznummern vermerkt wird, ob das Encoderpaket mit dieser Sequenznummer empfangen wurde. Des weiteren wird ein Ringbuffer aus Dependency-Slots benutzt, in den die Daten aus den Dependency-Kontrollpaketen geschrieben werden. Jeder Dependency-Slot wird um drei zusätzliche Spalten erweitert, in denen vermerkt wird, ob die Daten dieses Slots vollständig empfangen wurden (Spalte *all-data*), ob wenigstens ein Paket des Slots empfangen wurde (Spalte *any-data*), bzw. ob alle Abhängigkeiten vorhanden sind (Spalte *complete*). Weitere drei Spalten werden eingefügt, die genauso die Abhängigkeiten angeben, wie die Spalten *Require-1, Require-2, Require-some*, wobei diesmal der Verweis in die andere Richtung zeigt, also es wird angegeben, welche Slots von diesem Slot abhängig sind (dazu reichen drei Spalten, da niemals mehr als drei Abhängigkeiten von einem Slot ausgehen). Dies ist in Abbildung 6.18 dargestellt.

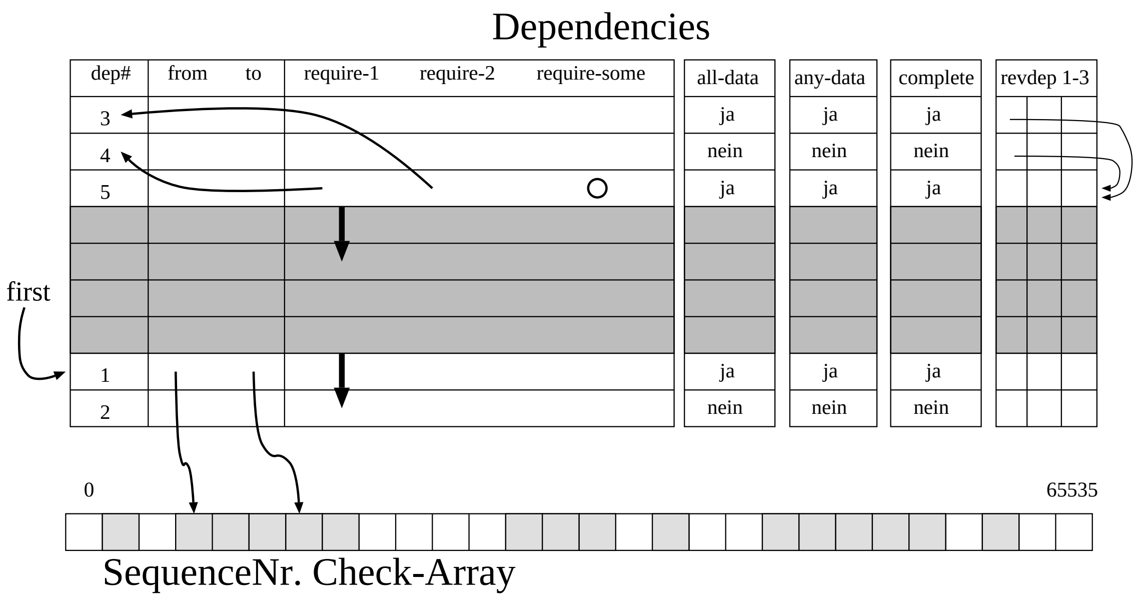


Abbildung 6.18: Fehlervermeidung anhand der Dependency-Daten

Alle Pakete, die in ihrem Slot in der Spalte *complete* ein `true` stehen haben, werden dann an den Decoder geschickt. Die drei Spalten mit den umgedrehten Abhängigkeiten werden beim Empfang der DependencySlots ausgefüllt, die anderen drei zusätzlichen Spalten werden mit `false` initialisiert. Jedesmal, wenn ein neues Encoderpaket empfangen wird, wird die Prozedur `DependencyCallbackNeuesPaket` aufgerufen, die den Sequenznummer-Bitvektor und die zusätzlichen Spalten des Dependency-Slots aktualisiert. Der zugehörige Programmcode ist in Pseudocode 6.8.1 angegeben. Aus Gründen der Übersichtlichkeit ist dieser sehr stark vereinfacht.

Programm 6.8.1 Ausfüllen der Zusatzspalten der DependencySlots

```
proc DependencyCallbackNeuesPaket(SeqNr seqnr)
  SeqNrCheck[seqnr] = true
  idx = first                                         erster DependencySlot
  while DependencySlot[idx].from < seqnr do
        idx := idx + 1
  od
  DependencySlot[idx].AnyData = true
  alldata := true
  for i := DependencySlot[idx].from to DependencySlot[idx].to do
      if SeqNrCheck[seqnr] = false then alldata := false fi
  od
  if alldata = true then DependencySlot[idx].AllData := true fi
  ChangeCompleteness(idx);
.
proc ChangeCompleteness(DepSlotNr idx)
  if DependencySlot[idx].complete = false ∧
     DependencySlot[idx].AnyData ∧
     DependencySlot[DependencySlot[idx].Require1].complete ∧
     DependencySlot[DependencySlot[idx].Require2].complete ∧
     DependencySlot[DependencySlot[idx].RequireSome].AnyData
     then
           DependencySlot[idx].complete := true
           changed := true
      else
           changed := false
  fi
  if changed = true then
                     ChangeCompleteness(DependencySlot[idx].revdep[1]);
                     ChangeCompleteness(DependencySlot[idx].revdep[2]);
                     ChangeCompleteness(DependencySlot[idx].revdep[3]);
  fi
.
```

6.9 Statistik

Der Statistikteil erfaßt:

- **die Paketverlustrate** in Abhängigkeit der Paketgröße,
- **die Paketlaufzeit** in Abhängigkeit der Paketgröße,
- **den Gesamtumfang der gesendeten Pakete** und
- **den Gesamtumfang der empfangenen Pakete**.

Das Problem mit der Messung dieser Daten liegt darin, daß der ideale Meßverlauf in Konflikt mit dem Videoversand liegt. Die Messung muß anhand der gesendeten Videodaten geschehen, ein großer Overhead für die Messung sollte möglichst vermieden werden. Dabei ist es natürlich Ziel der Videoübertragung, die Daten vollständig an den Client zu schicken. Die Messung der Verlustrate ist aber nur möglich, wenn in Kauf genommen wird, daß Pakete verloren gehen. Ein zusätzliches Problem ist, daß die Meßwerte eigentlich gleich zum Start der Übertragung benötigt werden, da der Benutzer erwartet, daß die Übertragung sofort mit dem Programmstart beginnt.

Zur Lösung wurden zwei Methoden kombiniert: Einerseits werden einige Pakete gezielt zu Messungen eingesetzt, was bedeutet, daß ab und zu auch Pakete geschickt werden, für die es unwahrscheinlich ist, daß sie ankommen. Andererseits schickt der Client schon zu Beginn der Übertragung dem Server die zu erwartende maximale Übertragungsrate und der Server speichert auch die gemessenen Werte in einer Datenbank, damit er bei der nächsten Übertragung an diesen Client schon zu Beginn eine recht genaue Abschätzung der Meßwerte erhält.

6.9.1 Ablauf der Messungen

Anhand der `ReceptionReport`-Pakete, welche der Client regelmäßig an den Server schickt, kann der Server sehr einfach feststellen, welche Pakete angekommen sind. Er muß sich also nur merken, wie groß jedes dieser Pakete war und hat damit sofort die Verlustrate in Abhängigkeit der Größe. Diesen Zusammenhang speichert er in einer Schrittweite von 1480 Bytes für die Größe. Dies entspricht genau der Größe eines Ethernet-Frames. Dies wurde gewählt, da ein Großteil der Netzwerke Ethernet-Netze sind und anzunehmen ist, daß die Verlustrate hauptsächlich abhängig von der Anzahl der Ethernet-Frames ist, in die ein Paket unterwegs aufgeteilt werden muß.

Für jede Paketgröße wird die Effektive Paketgröße (siehe Abschnitt 4.3) bestimmt. Die Größe, bei der dieser Wert das Maximum erreicht, wird als Paketgröße für den Paketversand an den Serveralgorithmus weitergeleitet.

Gelegentlich wird der Statistikteil aber auch andere Paketgrößen vorschlagen, damit er seine Messungen für diese Größen erweitern kann. Dabei warnt er aber den Server-Algorithmus, daß dieses Paket eine niedrigere Ankunftswahrscheinlichkeit hat. Der Server vermerkt sich dies und wenn ein Paket tatsächlich nicht ankommt und noch einmal angefordert wird, trennt er es in Pakete mit der normalen Größe auf und schickt sie so noch einmal. So kann das Paket auch dann noch den Client erreichen, wenn bei der zuerst vorgeschlagenen Paketgröße die Verlustrate 100% beträgt.

Um schnell eine passende Paketgröße zu finden, testet die Statistik die Paketgrößen nicht sequentiell durch, sondern benutzt eine Inverse-Binary-Ordnung (siehe Abschnitt 7.5). Die Größen werden also etwa in folgender Reihenfolge getestet: $1480, 65120, 32560, 16280, 48840, \ldots$. Da wir in Abschnitt 4.3 gesehen haben, daß es häufig Bereiche von Paketgrößen gibt, in denen die Verlustrate immer 100% beträgt, ist es nicht sehr sinnvoll, diese Bereiche jedesmal zu testen, da wir damit doch eine Gefahr des Datenverlustes eingehen. Man stelle sich nur einmal vor, daß die optimale Paketgröße ca. 6000 Bytes beträgt, was bei kleinen Bildgrößen durchaus 25 Bilder enthalten kann. Würden wir zum Test ein Paket mit 60000 Bytes schicken, das jedesmal verlorengeht und würde auch das Nachschicken nicht richtig klappen, so gingen auf einen Schlag bis zu 10 Sekunden Video verloren.

Deshalb wird das Verfahren leicht abgewandelt: Immer, wenn die Statistik ein Testpaket der Größe n auslösen will, wird eine Zufallszahl zwischen 0 und 1 bestimmt und das Testpaket nur dann geschickt, wenn diese Zufallszahl größer als $f \cdot l(n)$ ist. Dabei ist f ein frei wählbarer Parameter, der nicht weniger als etwa 0.9 betragen sollte. Ist die Zahl kleiner, wird das nächste Paket nicht in der Größe n geschickt, sondern in der Normalgröße und wir wechseln zur nächsten Testpaket-Größe. Mit diesem Verfahren werden also verstärkt Testpakete in den Größen geschickt, die wahrscheinlich ankommen, ohne nicht mehr auf eine Änderung in dem Netzwerkverhalten reagieren zu können.

Letztlich sei noch erwähnt, daß wir die Häufigkeit, in der Testpakete abgeschickt werden, mit der Zeit senken. Gesetzt den schlimmsten Fall, daß das Netzwerk plötzlich keine Pakete der aktuellen Größe mehr transportieren würde, so wird die Statistik trotzdem auf andere Paketgrößen wechseln, da die Statistik ja mit jedem gesendeten Paket erneuert wird und nicht nur durch die explizit ausgezeichneten Testpakete. Damit eine schnelle Reaktion möglich ist, wird ein Meßdaten-Alterungsverfahren benutzt. In diesem Fall sieht dieser ganz einfach so aus, daß immer nur die letzten n Pakete einer Größe zur Bestimmung der Verlustraten benutzt werden. Neuere Messungen können zusätzlich auch noch stärker gewichtet werden.

6.9.2 Speicherung von Messungen

Ungelöst blieb bisher noch das Problem, wie wir zu Beginn der Übertragung schnell eine gute Schätzung der Meßwerte bekommen. Dies lösen wir, indem wir die Messdaten, die wir in einer Übertragung gemessen haben, in einer zentralen Datei auf Serverseite abspeichern. Beginnen wir eine neue Übertragung mit einem bekannten Client, so können wir gleich mit den zuletzt gemessenen Daten beginnen. Das Problem hierbei ist aber, daß es einerseits sehr viele verschiedene Client-Rechner geben kann, andererseits uns das Verfahren in seiner einfachsten Form nichts bringt, wenn ein neuer Client-Rechner eine Verbindung aufbaut.

Wir modifizieren das Verfahren nun dahingehend, daß in dem Fall, in dem der Clientname noch nicht bekannt ist, wir so lange Komponenten des Namens abtrennen, bis wir in unserer Datei einen Rechner mit der gleichen Domain gefunden haben.

Die Datei besteht aus je einer Zeile pro Clientname und enthält in dieser zuerst den Clientname, bei dem zur einfacheren Suche die Namenskomponenten in umgekehrter Reihenfolge angegeben sind. Anstatt `trick.informatik.uni-stuttgart.de` steht in der Datei also `de.uni-stuttgart.informatik.trick`. Diesem Namen folgt das Datum, an dem der Eintrag zuletzt verändert wurde und die empfangene Bitrate. Darauf folgen 44 gleiche Einträge, die jeweils die Verlustrate und die mittlere Übertragungszeit bei den Größen $1480, 2960, \ldots, 65120$ angeben.

Bei einer Verbindung mit einem neuen Client werden alle Rechner gesucht, deren Domainname mit möglichst vielen gleichen Namenskomponenten beginnt. Aus allen diesen Rechnern werden die Mittelwerte der Daten berechnet und diese als Anfangs-Schätzung benutzt. Unter der Bedingung, daß die Daten in der Datei so gespeichert sind, daß Clients mit gleicher Anzahl übereinstimmender Namenskomponenten direkt aufeinanderfolgen, lassen sich die Daten mit dem Programmausschnitt 6.9.2 auslesen.

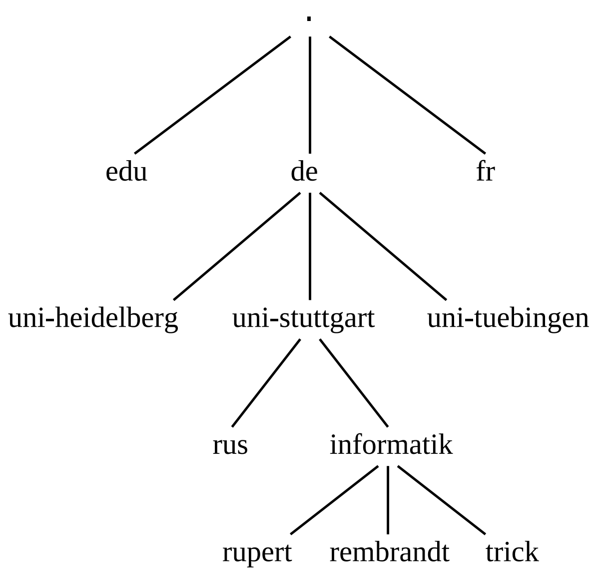


Abbildung 6.19: Baum der Domainnamen

Betrachten wir ein Beispiel. Unsere Datei laute folgendermaßen:

```
de.uni-tuebingen.zdv.aix06          10.10.97  10000  10 200  12 210  ...
de.uni-tuebingen.zdv.aix08          12.10.97  11000  12 180  11 220  ...
de.uni-stuttgart.informatik.trick   08.10.97  20000   5  50   7  80  ...
de.uni-stuttgart.informatik.tick    09.08.97  20200   5  45   8  90  ...
de.uni-stuttgart.informatik.rupert  15.08.97  50000   0  10   0  12  ...
de.uni-stuttgart.rus.servus01       10.09.97  15000   2  12   5  20  ...
edu.wustl.edu                       20.09.97   2000  20 500  25 600  ...
```

Würde nun der Client `de.uni-stuttgart.informatik.donald` eine Übertragung anfordern, so würde der Server den Mittelwert der Messdaten von `trick`, `tick` und `rupert` bilden, da nur bei diesen drei Rechnern drei Namenskomponenten übereinstimmen (`de`, `uni-stuttgart` und `informatik`). Würde ein Client mit dem Namen `fr.paris.www` eine Verbindung anfordern, so ist kein passender Eintrag in der Datei enthalten und es wird der Mittelwert über alle Einträge gebildet.

Der Grund, daß die Rechnernamen benutzt werden und nicht die numerische IP-Adresse liegt darin, daß man die IP-Adresse nicht so einfach in Komponenten aufteilen kann. Außerdem können Subnets benutzt werden, die anhand der IP-Adresse nicht mehr erkannt werden können. Da die Rechnernamen meist so gewählt sind, daß sie in etwa den Aufbau des Netzwerks widerspiegeln, eignen sie sich besser. Diese Heuristik versagt aber bei Rechnernamen, wie sie oft von Firmen benutzt werden (z.B. `www.firmenname.com`).

6.9.3 Pruning der Cachedatei

Um ein zu starkes Anwachsen der Cachedatei zu verhindern wird sie von Zeit zu Zeit von alten Einträgen befreit. Dazu kann man in Abhängigkeit von der Anzahl der Komponenten im Namen bestimmen, wie lange der Eintrag in der Datei erhalten bleiben soll. Namen mit mehr Komponenten veralten dabei schneller. Sagen wir einmal, daß Einträge mit 4 Namenskomponenten nach

Programm 6.9.1 Auslesen der gespeicherten Messwerte

```
maxnkomp := −1
quit := false
while ¬EndOfFile() ∧ ¬quit do
        zeile := LeseEineZeileAusDatei()
        dname := Rechnername(Zeile)
        nkomp := AnzahlGleicherNamenskomponenten(dname, clientname)
        if nkomp > maxnkomp then
                                    maxnkomp := nkomp
                                    InitialisiereStatistik()
        fi
        if nkomp = maxnkomp then
                                    ErweitereStatistikUmZeile(zeile)
        fi
        if nkomp < maxnkomp then
                                    quit := true
        fi
od
```

2 Monaten veralten, dann würde am 9.10.97 der Eintrag `de.uni-stuttgart.informatik.tick` einen neuen Namen bekommen: `de.uni-stuttgart.informatik`. Dieser veraltet dann wieder, wenn das Maximalalter für Namen mit drei Komponenten erreicht ist. Auf den ersten Blick scheint dieses Verfahren vielleicht sinnlos, da der Eintrag ja doch erst mit der letzten Komponente gelöscht wird. Das ist aber nicht der Fall, da durch dieses Verfahren gleiche Einträge entstehen können, die natürlich zusammengefaßt werden. Wenn am 15.10.97 z.B. der Eintrag `de.uni-stuttgart.informatik.rupert` zu `de.uni-stuttgart.informatik` unbenannt wird, kann dieser Eintrag mit obigem zu einem verschmolzen werden. Der verschmolzene Eintrag erhält das spätere Datum und den Mittelwert der Daten. Mit der Zeit reduziert sich also die „Auflösung“ der Schätzung.

6.10 Nachrichtenformate

6.10.1 Transmission-Paket

TransNr	Transmission-Nr. des Pakets	16 bits
replaces	PaketNr. die durch dieses Paket ersetzt wird	16 bits
resent	gleich 1, wenn Paket schon einmal gesendet wurde	1 bit
unused		15 bits
CurrTime	aktuelle Server-Uhrzeit	
EndACK	letzte PaketNr. für die ein Reception-Report empfangen wurde	15 bits
nPackets	Anzahl der nun folgenden EncoderPakete	16 bits
	EncoderPaket 1	
	...	
	EncoderPaket nPackets	

Tabelle 6.1: Transmission-Paket

In `TransNr` steht die fortlaufende Numerierung der Pakete, im Normalfall wird diese Nummer auch in `replaces` eingetragen. Wird ein Paket (z.B. Nr. 5) allerdings erneut gesendet und für das erneute Senden in mehrere kleinere aufgeteilt, so erhält jedes kleinere eine neue `TransNr` und im Feld `replaces` wird die Nummer des alten Pakets eingetragen (hier Nr. 5). Daran kann der Client merken, daß er das Paket Nr. 5, das er vorher nicht erhalten hat, nun in anderer Form bekommt. `resend` ist immer 0, wenn das Paket zum ersten Mal verschickt wird, sonst 1. Die Uhrzeit in `CurrTime` wird jedesmal, wenn das Paket verschickt wird, neu gemessen und eingetragen. `EndACK` ist das letzte Paket, für das der Server einen ReceptionReport bekommen hat.

An diese Informationen schließen sich `nPackets` EncoderPakete an. Die Länge dieser Pakete steht jeweils in deren Header, so kann sie der Client wieder einfach trennen.

6.10.2 Encoder-Pakete

unused		8 bits
nBytes	Anzahl der in diesem Paket enthaltenen Datenbytes	16 bits
SeqNr	Sequence-Nummer des Pakets	16 bits
PTS	Uhrzeit, zu der die Daten im Datenpaket anzuzeigen sind.	32 bits
Datenpaket		

Tabelle 6.2: Encoder-Paket

Dazu gibt es wenig zu sagen. Die Uhrzeit ist in Millisekunden angegeben. Der Nullpunkt kann beliebig gewählt werden, da die Clientuhr synchron zur Serveruhr eingestellt wird.

6.10.3 Dependency-Datenpaket

FirstSlotNr	Nummer der ersten Slots	16 bits
Slot 1 + FirstSlotNr		
From	erste SeqNr.	16 bits
To	letzte SeqNr.	16 bits
Require1		16 bits
Require2		16 bits
RequireSome		16 bits
...		
Slot n + FirstSlotNr		
From	erste SeqNr.	16 bits
To	letzte SeqNr.	16 bits
Require1		16 bits
Require2		16 bits
RequireSome		16 bits

Tabelle 6.3: Dependency-Datenpaket

Ausführliche Beschreibung in Abschnitt 6.6.2 auf Seite 65.

6.10.4 Reception-Report

Feldname	Beschreibung	Länge in Bits
MsgType	ReceptionReportID = 0x01	8 bits
MsgLen	Länge der Message in Bytes	16 bits
LastRecvNr	Transmission-Nummer des letzten empfangenen Pakets	16 bits
LastRecvDelay	Übertragungszeit des Pakets LastRecvNr	16 bits
StartNr	erste Transmission-Nummer in folgendem Bitvektor	16 bits
EndNr	letzte Transmission-Nummer in folgendem Bitvektor	16 bits
RecvFlag 1	gleich 1, wenn Paket angekommen	1 bit
...	...	...
RecvFlag n	gleich 1, wenn Paket angekommen	1 bit

Tabelle 6.4: ReceptionReport-Message

Dieses Paket schickt der Client dem Server über den Kontrollkanal. Verluste werden einfach ignoriert. Da die Paketgröße sehr klein ist (wird wohl immer in ein Ethernet-Paket passen), wird die Verlustrate aber sehr niedrig sein.

Die `LastRecvNr` wird vom Packet-Resending-Mechanismus benötigt. Die `LastRecvDelay` dagegen von der Statistik. Die Übertragungsverzögerung `LastRecvDelay` wird vom Client aus der Empfangsuhrzeit und der Serveruhrzeit, die im Transmissionpaket im Feld `CurrTime` eingetragen war, bestimmt. Zwar kann es durch diese Verschachtelung zweier völlig unterschiedlicher Informationen vorkommen, daß der Server für einige Pakete keine Übertragungszeit geliefert bekommt, was aber nicht weiter schlimm ist.

Die `RecvFlag`s am Ende der Message geben für die Transmissionpakete `StartNr` bis `EndNr` an, ob sie schon angekommen sind. Wurde ein Transmissionpaket durch ein anderes ersetzt (s. Feld `replaces` im Transmissionpaket), so wird auch für dieses ersetzte Paket eine „1“ eingetragen.

Diese Message wächst mit jedem empfangenen Paket, für das ein weiteres Bit angehängt wird. Empfängt der Server jedoch eine ReceptionReport-Message, so meldet er im nächsten Transmission-Paket, daß er jetzt alle Daten bis Trans.Nr. `EndNr` erhalten hat und der Client schickt in seiner nächsten ReceptionReport-Message die Daten ab TransmissionNr. `EndNr` + 1.

6.11 Der MPEG-Video-Encoder

6.11.1 Zuweisen von Zeitmarken an die Bilder

Der Encoder muß jedem Bild eine absolute Zeitmarke zuordnen, anhand derer der Server ersehen kann, wann die Daten zu schicken sind. In einem MPEG-System-Stream ist es nicht erforderlich, daß allen Bildern ein PTS zugewiesen wird. Vielmehr sind in dem Stream i.A. nur sporadisch PTSs vorhanden, die sich immer auf das nachfolgende Bild beziehen. Erschwert wird die Zuweisung richtiger PTSs noch dadurch, daß die Bilder nicht in der Reihenfolge der Darstellung (also in monoton aufsteigenden PTSs) vorliegen. Es ist aber für jedes Bild die absolute Position innerhalb einer GOP gespeichert. PseudoCode 6.11.1 soll verdeutlichen, wie die PTS-Zeiten ermittelt werden.

Während der Stream verarbeitet wird, werden der zuletzt gelesene PTS ($last\_pts$) und der DisplayOrder-Index ($doidx_{last\_pts}$) des Bildes gespeichert, auf den sich der $last\_pts$-Wert bezieht. Trifft man nun ein Bild (mit dem DisplayOrder-Index $doidx$) an, für das kein PTS gespeichert ist, so kann man anhand der Bildrate in SystemClock-Ticks ($rate$) einen ungefähren Wert pts' für dieses Bild gemäß

$$pts' = last\_pts + (doidx - doidx_{last\_pts}) \cdot rate$$

berechnen.

Da zu Beginn einer GOP der DisplayOrder-Index wieder bei 0 beginnt, muß der $doidx_{last\_pts}$ um die Länge der letzten GOP ($doidx_{max}$) nach vorne verschoben werden:

$$doidx_{last\_pts} := -(doidx_{max} + 1 - doidx_{last\_pts})$$

Synchronisation der Encoder-Uhr

Im Encoder läuft eine Echtzeituhr, die regelt, wann ein bestimmtes Bild gesendet werden soll. Bis zum Auftreten des ersten PTS (was auch nie geschehen kann) ist die Encoder-interne Uhr nicht zum MPEG-Stream synchronisiert und startet somit mit einer beliebigen Zeit. Sobald der erste PTS $last\_pts$ für ein Bild im Stream erscheint, wird die Uhr soweit vorgestellt, daß sie dann den Wert $last\_pts$ erreicht, wenn sie den Wert des vorläufig berechneten PTS pts' des aktuellen Bildes erreicht hätte. Im folgenden gilt die Uhr als synchronisiert und wird nicht mehr modifiziert.

6.11.2 Timing des Packetversands

Im Encoder läuft eine Echtzeituhr in PTS-Einheiten und jedes Bild des MPEG-Streams wurde mit einer PTS-Zeit versehen, zu dem das Bild geschickt werden soll. Da zukünftige I- und P-Bilder aber i.A. zur Rekonstruktion früherer Frames benötigt werden und deshalb im Stream vor zeitlich davor liegenden Bildern erscheinen, kann nicht einfach auf die jeweilige PTS-Zeiten gewartet werden.

Der Encoder wartet bei jedem B-Bild, da hinter diesem im Strem keine zeitlich vorhergehenden Bilder mehr folgen können. I- und P-Bilder werden sofort ohne Verzögerung geschickt, es sei denn, der DisplayOrderIndex des Bildes ist genau um eins größer als der bisher größte DisplayOrderIndex in der GOP. In diesem Fall können hinter dem betrachteten Bild keine Bilder mehr folgen, die von diesem abhängig sind. Diese letzte Regel ist unbedingt erforderlich, da ein MPEG-Stream durchaus nur I- oder nur I- und P-Bilder enthalten kann.

Programm 6.11.1 PTS (Presentation Time Stamp)-Zuweisung an die Einzelbilder

```
var
  first = TRUE;                                    ist das das erste Bild im Stream?
  sync = FALSE;            läuft die Encoder-Uhr schon synchron zum PTS-Wert des Streams?
  rate = const.                die Anzahl der PTS-Einheiten, die im Normalfall pro Bild vergeht
  clk = 0; (volatile)      die aktuelle Uhrzeit (in PTS-Einheiten), die in Echtzeit weiterläuft
  pts = -rate;                             der PTS, der dem aktuelle Bild zugewiesen wird
  last_pts = 0;                               der zuletzt aus dem Stream gelesene PTS-Wert
  doidx_{last_pts};        Display-Order-Index (DOIdx) des Bildes, zu dem der last_pts-Wert gehört
  doidx_{max};                           der höchste DOIdx in der aktuellen GroupOfPictures
end
begin
  for ever do
      ReadDataUntilStartcode();
      if StartCode() = GroupOfPicture
        then
            doidx_{last_pts} := -(doidx_{max} + 1 - doidx_{last_pts});
            doidx_{max} := -∞;
      fi;
      if StartCode() = Picture
        then
            doidx := CurrentPictureDisplayOrderIndex();
            doidx_{max} = max(doidx_{max}, doidx);
            if first
              then
                  first = FALSE;
                  doidx_{last_pts} := pidx;
            fi;
            pts' := last_pts + (doidx - doidx_{last_pts}) * rate; Zeit ohne Kenntnis des PTS-Wertes
            if PictureHasNewPTSValue()
              then
                  last_pts := CurrentPTS();
                  doidx_{last_pts} := doidx;
                  if ¬sync          Uhr jetzt synchronisieren, da PTS-Wert im Stream auftrat
                    then                                                    Uhr umstellen
                        clk := last_pts - (pts' - pts);    in pts steht PTS des letzten Bildes
                        sync := TRUE;
                  fi
                  pts' := last_pts;
            fi
            pts :=' pts;
            AssignPTSToPicture(pts);
      fi;
  od
end
```

Kapitel 7

Playbackjitter-Reduktion

7.1 Einführung

Durch unterschiedliche Einflüsse kann es geschehen, daß die zeitlichen Abstände zwischen aufeinanderfolgenden Bildern schwanken. Diese Schwankung bezeichnet man als *Playback Jitter* und sie wurde als mittlere Abweichung des Zeitabstands zwischen zwei Bildern über eine Sekunde definiert: Sei d_i der Zeitabstand zwischen Bild i und $i+1$, n die Anzahl der Bilder, so ist der Jitter j definiert als:

$$j := \frac{1}{n} \sum_{i=1}^{n} \left| \frac{1s}{n} - d_i \right|$$

Man beachte, daß der Jitter bei äquidistanten Bildern unabhängig von der Bildrate gleich 0 ist (siehe auch Tabelle 7.1).

Diese Definition hat meiner Meinung nach den Nachteil, daß sie das tatsächliche Jitter-Empfinden nicht genau abbildet. Dies kann man an den letzten beiden Zeilen in der Tabelle sehen. Eine Bildsequenz gemäß der vorletzten Zeile wirkt für den Betrachter gleichmäßiger als die letzte, aber beide haben laut Definition der Formel genau den gleichen Jitter.

Bildsequenz, $\frac{1}{30}$s pro Zeichen	Bilder/Sec	Jitter/msec
`XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX`	30	0
`X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-`	15	0
`XX-XX-XX-XX-XX-XX-XX-XX-XX-XX-`	20	17
`XX-X--XX-X--XX-X--XX-X--XX-X--`	15	22
`XXX---XXX---XXX---XXX---XXX---`	15	45
`XXXXX-----XXXXX-----XXXXX-----`	15	53
`XXX-XXX-XXX-XXX-XXX-XXX-XXX-X-`	22	15,4
`XXXXXXXXXXXXXX-X-X-X-X-X-X-X-`	22	15,4

Tabelle 7.1: Playback Jitter für einige Bildsequenzen (`X`, Bild dargestellt; `-`, Bild nicht dargestellt)

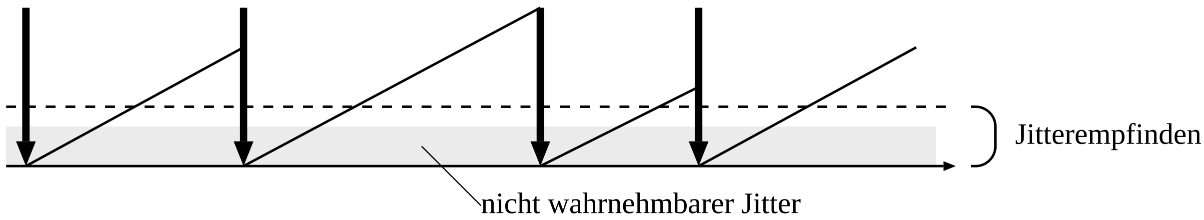
7.2 Jitterreduktion durch prioritisierte Übertragung

Werden auf dem Übertragungsweg oder aufgrund mangelnder Decoderperformance Frames verloren, so entsteht ein gewisser Playback-Jitter, den es zu minimieren gilt. Hat man das Hilfsmittel der prioritätsgesteuerten Übertragung, so kann man für eine Bildsequenz eine Prioritätsreihenfolge erstellen, die den Playback-Jitter trotz Paketverlust im Mittel verringert. Dabei wird angenommen, daß die Bilder mit höheren Prioritäten wahrscheinlicher beim Empfänger ankommen. Dies ist fast[1] äquivalent zu der Vorstellung, daß solange Bilder in einer Folge monoton absteigender Prioritäten gesendet werden, bis die vorhandene Bandbreite erschöpft ist.

Erschwert wird diese Neuordnung der Frames dadurch, daß bei vielen Videocodierungen Abhängigkeiten zwischen den Frames bestehen. Frames, die anderen als Referenz dienen, müssen deshalb generell eine höhere Priorität zugewiesen bekommen, als die davon abhängigen. Die Halbordnung der Abhängigkeiten muß also in der Ordnung der Prioritäten enthalten sein.

7.3 Ein mögliches Modell des Jitterempfindens

Dieses Modell geht von der Beobachtung aus, daß Menschen bei Wiedergabe einer Bildsequenz nach jedem Bild ein Nachfolgebild erwarten. Diese *Erwartung* steigt mit der Darstellungsdauer des letzten Bildes. Bei Darstellung des nächsten Bildes ist die Erwartung befriedigt und die Erwartung steigt von nun an wieder ausgehend von 0.



Jedem Bild B_i, das zur Zeit t_i dargestellt wird, werde nun die Funktion

$$f_i = \begin{cases} c \cdot (t - t_i) & \text{für } t \geq t_i \\ \infty & \text{sonst} \end{cases}$$

zugewiesen, die eine obere Schranke der Erwartung beschreibt, wenn das Bild B_i angezeigt wird. Das mittlere Jitterempfinden in dem Zeitraum $[0;\ T]$ berechnet sich dann einfach als Mittelwert der Erwartung eines neuen Bildes:

$$J = \frac{1}{T} \int_0^T \min_i f_i(t) dt$$

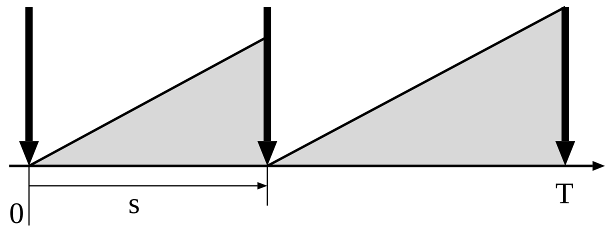
Diese Definitionen sind sehr ungenau, da sie alleinig der Intuition entspringen. Auch kommt es nicht auf einen absoluten Wert für c an, da im folgenden nur versucht wird, J zu minimieren. Der genaue Wert ist unwesentlich.

Dieses Maß für das Jitterempfinden hat im Grunde nichts mit obiger Definition von j zu tun. Allerdings besteht die Beziehung, daß J genau dann minimal ist, wenn $j = 0$ ist. Dies soll hier plausibel gemacht werden, indem wir die Position eines Frames zwischen zwei Frames bestimmen, die den minimalen Jitter erzeugt:

In diesem Fall gilt für den Bereich $t \in [0;\ T]$:

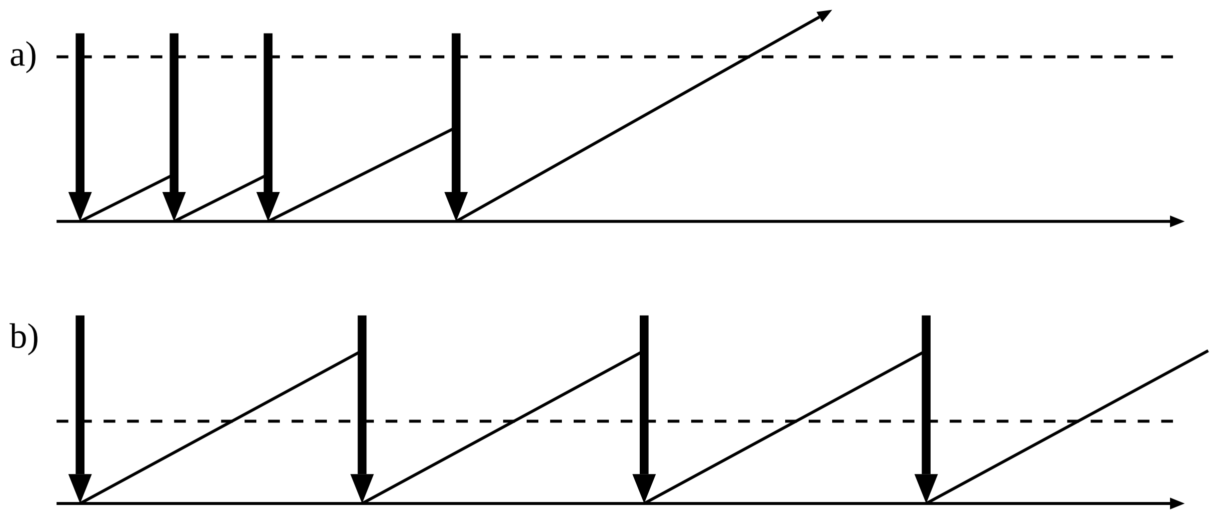
$$J = \frac{1}{T} \int_0^s xc\ dx + \frac{1}{T} \int_0^{T-s} xc\ dx = \frac{1}{2T} cx^2 \Big]_0^s + \frac{1}{2T} cx^2 \Big]_0^{T-s} = \frac{1}{2T} cs^2 + \frac{1}{2T} c(T-s)^2$$

[1]Bei einer prioritätsgesteuerten Übertragung ist es immer noch möglich, daß Frames mit niedrigen Prioritäten ankommen, wohingegen Frames mit höheren Prioritäten fehlen



$$\frac{dJ}{dt} = \frac{1}{T}\left(cs - c(T-s)\right) = \frac{2cs}{T} - c$$

Aus $\frac{dJ}{dt} \stackrel{!}{=} 0$ folgt dann direkt: $s = \frac{1}{2}T$. An dieser Position ist J natürlich ein Minimum. Betrachten wir nun eine Sequenz von Frames, ohne äquidistanten Abstand. Diese Sequenz kann für die gegebene Anzahl Frames und vorgegebene Dauer kein minimales J besitzen, denn wenn wir drei aufeinanderfolgende Frames betrachten, die zwei verschiedene Zeitabstände besitzen, so können wir den Jitter reduzieren, indem wir das mittlere Frame genau zwischen die beiden äußeren Frames plazieren. J ist also genau dann minimal, wenn die Framefolge äquidistant, also $j = 0$ ist.



7.4 Anti-Jitter für Daten mit Abhängigkeiten

In [Net] wird folgender Algorithmus (allerdings ohne Begründung der Vorgehensweise) beschrieben: Wir haben eine Folge von n Bildern $a_1 \ldots a_n$. Diesen Bildern werden nun sukzessive absteigende Prioritäten zugewiesen. Dazu bezeichnen wir Bildern, denen schon eine Priorität zugewiesen wurde als *schwarz* und die anderen als *weiß*. Bilder, die weiße Bilder als Referenzbilder benötigen, bezeichnen wir als *rot*. Diese roten Bilder können im nächsten Schritt nicht in die Sequenz aufgenommen werden. Für jedes weiße Bild bezeichne nun L die Anzahl der Bilder, die sich bis zum nächsten schwarzen links neben dem Bild befinden und R entsprechend die Anzahl der Bilder bis zum nächsten schwarzen auf der rechten Seite. W bezeichne die Länge der Kette von weißen oder roten Bildern, in der sich das Bild befindet. Es gilt also stets $W = L + 1 + R$. Die Bildfolge denken wir uns von schwarzen Bildern begrenzt.

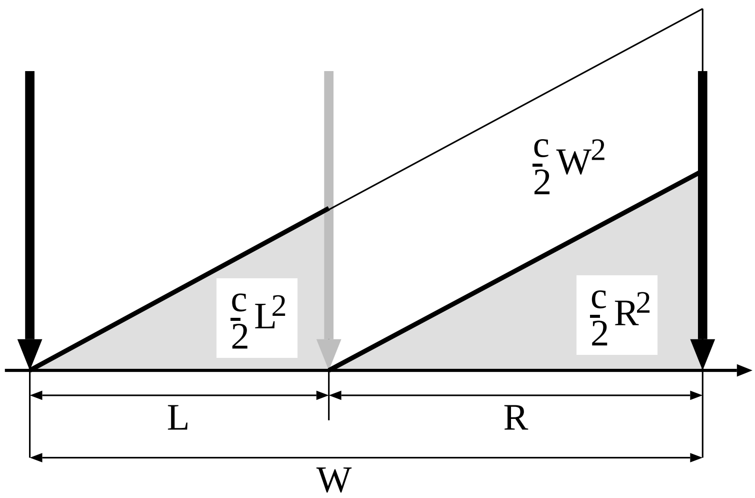


Abbildung 7.1: Wahl des Bildes mit größter Jitterreduktion

Nehmen wir einmal an, daß wir eine weiße Kette durch ein Frame in zwei Teile teilen (s. Bild 7.1). Dadurch verringert sich die Jitterempfindung $J \cdot T$ um den Wert $\frac{c}{2}W^2$ und erhöht sich um $\frac{c}{2}(L^2 + R^2)$.

Will man nun das Bild mit der nächstniederen Priorität finden, so sucht man natürlich dasjenige Bild, das den Jitter möglichst stark reduziert. Dazu berechne man für jedes weiße Bild den Wert $W^2 - L^2 - R^2$, der (bis auf einen konstanten Faktor) die Reduzierung des Jitters angibt und suche das Bild mit dem maximalen Wert. Diesem Bild wird nun die nächstniedrigere Priorität zugewiesen und als schwarz markiert. Gegebenenfalls müssen nun noch einige rote Bilder in weiße umgewandelt werden. Diesen Vorgang wiederholen wir nun so oft, bis alle Bilder schwarz sind und somit die Prioritäten vollständig verteilt sind.

Sind sämtliche Bilder einer Kette weiß, so wird dieser Algorithmus stets das mittlere Bild auswählen, wie man leicht analytisch zeigen kann. Dazu berechnen wir in Anhängigkeit von L das Extremum. Man beachte, daß W in diesem Falle eine Konstante ist.

$$\begin{aligned}
\frac{d\,(W^2 - L^2 - R^2)}{dL} &= -\frac{d\,(L^2 + (W - L - 1)^2)}{dL} \\
&= -\frac{d\,(2L^2 + (W-1)^2 - 2(W-1)L}{dL} \\
&= -4L + 2W - 2 \stackrel{!}{=} 0
\end{aligned}$$

$$L = \frac{W-1}{2}$$

Außerdem sucht sich der Algorithmus im Spezialfall, daß keine roten Bilder vorhanden sind, immer das mittlere Bild in der längsten weißen Bild-Kette heraus, denn

$$\begin{aligned} W^2 - L^2 - R^2 &= W^2 - (W-1)/2 - (W - (W-1)/2 - 1)^2 \\ &= W^2 - W/2 + 1/2 - (W/2 - 1/2)^2 \\ &= W^2 - W/2 + 1/2 - W^2/4 - 1/4 + W/2 \\ &= \frac{3}{4}W^2 + 1/4 \end{aligned}$$

wächst mit steigendem W (W ist stets größer als 0).

7.4.1 Erweiterung

Man kann den Algorithmus noch durch eine Zusatzregel etwas verbessern: Existieren mehrere Möglichkeiten, das nächste Frame zu wählen und sind alle gleich gut, dann wird dasjenige Frame gewählt, das nach dem Einfügen die kürzeste schwarze Kette bildet. Zwar ist die Auswahl dann immer noch nicht eindeutig, aber die entstehenden Sequenzen sind etwas besser.

7.5 Anti-Jitter für Daten ohne Abhängigkeiten

Nun soll als Spezialfall ein sehr einfacher Algorithmus für Videosequenzen ohne Abhängigkeiten vorgestellt werden (z.B. für Motion-JPEG Sequenzen).

Zunächst sei der Begriff der IBO (inverse binary order) beschrieben. Eine IBO einer Folge von natürlichen Zahlen von 0 bis n erhalten wir, indem wir die binäre Darstellung dieser Zahlen rückwärts schreiben, diese neue Folge dann nach aufsteigendem Wert sortieren und die Zahlen der sortierte Folge wieder umdrehen. Die IBO z.B. der Zahlen von 0-12 ist somit (0, 8, 4, 12, 2, 10, 6, 1, 9, 5, 3, 11, 7).

Sei nun $a_0 \ldots a_{n-1}$ unsere Folge von n Dateneinheiten mit streng monoton aufsteigenden Darstellungszeiten. Dann erhalten wir eine zugehörige Folge Prioritäten $p_0 \ldots p_{n-1}$, indem wir einfach $p = IBO(0 \ldots n-1)$ setzen.

Mit diesem Verfahren läßt sich auch sehr einfach eine Priorisierung von unkomprimierten Audiodaten realisieren. Dazu unterteilen wir die zu sendenden Audiodaten in Gruppen zu je n Samples $s_0 \ldots s_{n-1}$. Jede dieser Gruppen unterteilen wir wiederum in Pakete zu je $m \ll n$ Samples. Dabei besteht jedes Paket p_k aus den Samples

$$\{s(i \cdot m + k) \quad | \quad 0 \leq i < \frac{n}{m} \wedge i \cdot m + k < n\}$$

Samples aus fehlenden Paketen können dann durch Interpolation ersetzt werden. Störende Aliasingeffekte, die bei solch einer Unterabtastung auftreten können, sollen unbeachtet bleiben.

7.6 Neuer Jitteralgorithmus für Daten mit Abhängigkeiten

Dieser neue Algorithmus benutzt anstatt des Jitterempfindens J eine Jittergüte-Funktion $G(t)$, die angibt, wie *jitterfrei* eine Sequenz im Laufe der Zeit wirkt. Jedesmal, wenn ein Frame in die Sequenz eingefügt wird, wird die Gütefunktion G durch Addition einer Verbesserungsfunktion $V(t)$ modifiziert (s. Abbildung 7.2).

Nun werden sukzessive jene Frames der Sequenz dargestellt, bei denen $G(t)$ minimal ist (s. Abbildung 7.3 auf Seite 85). Dabei wird nur unter den Frames gesucht, deren Referenzframes schon dargestellt sind.

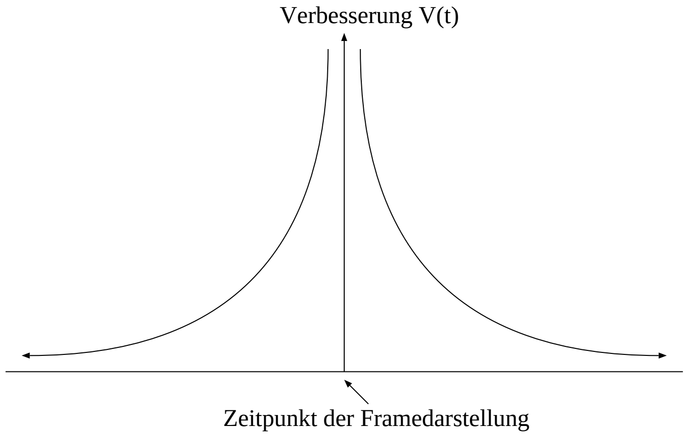


Abbildung 7.2: Beispiel einer Verbesserungsfunktion

Bleibt noch die Frage, welche Funktion für $V(t)$ zu wählen ist. Es scheint, daß dieses nicht besonders entscheidend ist. Sie sollte lediglich bei $t = 0$ einen Pol haben und für alle t sollte $V(t) >= 0$ sein. Um ein Verhalten zu erreichen, das im wesentlichen obigem Algorihmus entspricht, sollte die Summe zweier verschobener Funktionen $V(t)$ und $V(t-l)$ ein Minimum genau in der Mitte der beiden Bezugspunkte, also bei $l/2$ haben. Um ein Extremum in der Mitte zu bekommen, verlangen wir einfach, daß die Funktion symmetrisch zur y-Achse ist. Dieses Extremum soll ein Minimum sein, außerdem darf die Summe nicht konstant sein, da wir sonst überhaupt kein Extremum hätten. Dazu leiten wir im folgenden eine Bedingung ab:

$$G(t) = V(t) + V(t-l)$$

$$\frac{d^2G}{dt^2} = \frac{d^2V}{dt^2}(t) + \frac{d^2V}{dt^2}(t-l)$$

$$\frac{d^2G}{dt^2}\left(\frac{l}{2}\right) = \frac{d^2V}{dt^2}\left(\frac{l}{2}\right) + \frac{d^2V}{dt^2}\left(-\frac{l}{2}\right) = 2\frac{d^2V}{dt^2}\left(\frac{l}{2}\right) \stackrel{!}{>} 0$$

Da dies für alle l gelten soll, lautet die Bedingung somit einfach:

$$\frac{d^2V}{dt^2} > 0$$

Eine Funktion, die diese Bedingung erfüllt, ist

$$V(t) = \left|\frac{1}{t}\right|$$

da $\forall t : \frac{d^2V}{dt^2} = |2t^{-3}| > 0$ ist.

Da es unpraktikabel ist, diesen Algorithmus auf die gesamte Sequenz anzuwenden, wird man den Algorithmus auf Teilen der Sequenz getrennt anwenden. Allerdings ist es nicht ratsam, dies naiv zu machen, da dann im ersten Schritt z.B. das 1. Frame gewählt wird. Nachdem das zugehörige V zu G addiert wurde, befindet sich das Minimum von G nun beim letzten Bild. Würden keine weiteren Bilder mehr eingefügt, so bestünde die Endsequenz aus lauter Bildpaaren in großem Abstand, was einen denkbar ungünstigen Jitter zur Folge hätte (s. Abb. 7.4).

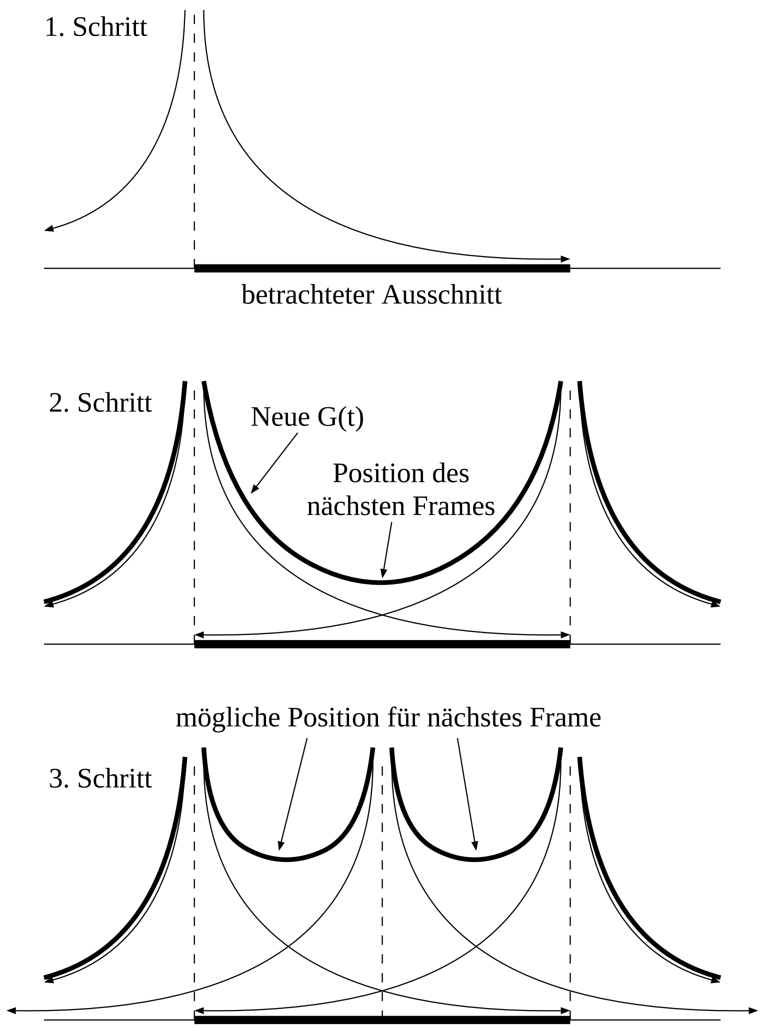


Abbildung 7.3: Prinzipieller Ablauf

Wählt man diese Teile jedoch so, daß der Inhalt bezüglich der Abhängigkeiten immer gleich strukturiert ist, so kann man annehmen, daß sich die Auswahl der Frames durch den Algorithmus in jedem Sequenzteil der Länge n in gleicher Weise wiederholen wird. Diese Annahme kann wiederum benutzt werden, eine bessere Jitterunterdrückung zu erreichen. Dazu addieren wir, wenn wir ein Frame i einfügen nicht nur V an der Stelle von diesem Frame, sondern für alle Frames $i+k\cdot n, k\in\mathbb{Z}$ die entsprechend verschobene Funktion V. Damit werden die Frames wieder gleichmäßig verteilt (s. Abb. 7.5).

Um dies effizient implementieren zu können, vernachlässigen wir den Einfluß der Frames mit einem Abstand größer als n und betrachten somit nur die Frames i, $i-n$ und $i+n$. Dies ist äquivalent dazu, $V(t)$ außerhalb des Bereiches $[-l,l]$ auf 0 zu setzen. Nun denken wir uns eine Teilsequenz als Ring zusammengefügt. Jedes i. Frame in diesem Ring repräsentiert außer dem i. Frame in der Teilsequenz auch das i. Frame der nächsten und der vorhergehenden Teilsequenzen (s. Abbildung 7.6).

Es reicht nun, V nur einmalig auf den G-Zyklus zu addieren. Da die entsprechenden Frames der anderen Teilsequenzen alle auf den gleichen Frame des G-Zyklus abgebildet sind, wird sich der

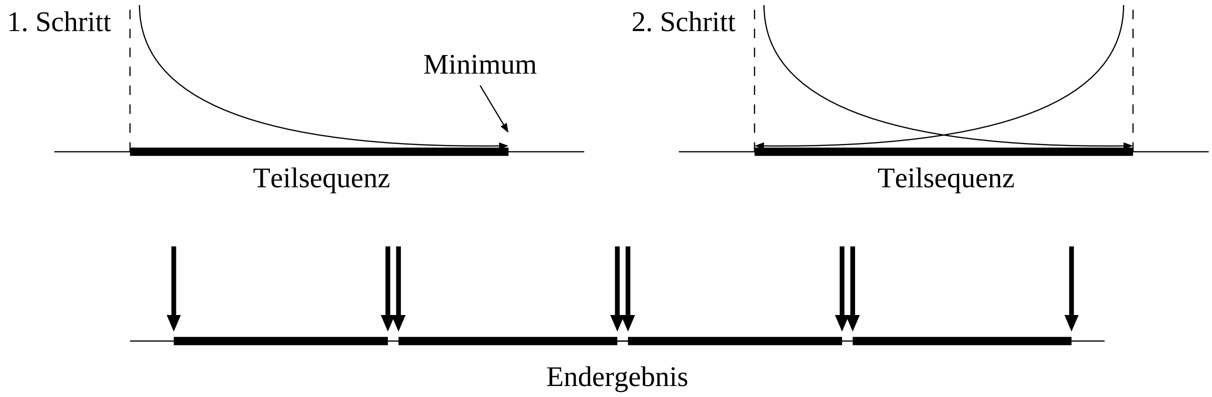


Abbildung 7.4: Ungünstiges Verfahren bei Anwendung auf Teilsequenzen

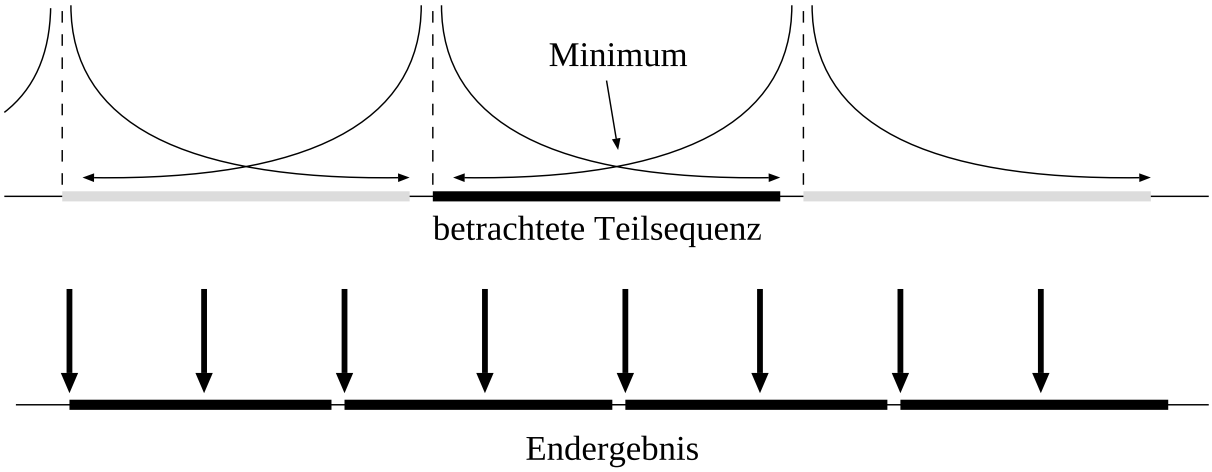


Abbildung 7.5: Verbessertes Verfahren bei Anwendung auf Teilsequenzen

Wert von G nur um einen konstanten Faktor unterscheiden. Da uns der absolute Wert aber nicht interessiert und wir nur das Minimum suchen, ist das ohne Bedeutung.

Auch dieser Algorithmus darf nur als grobes, rein intuitives Modell betrachtet werden.

7.7 Ergebnisse

Zum Vergleich der Ergebnisse des Algorithmus aus [Net] mit der oben beschriebenen Erweiterung (im folgenden als Algorithmus A bezeichnet) und des neuen Algorithmus (B) seien hier die Ausgaben beider Algorithmen für eine MPEG-codierte Videosequenz angegeben. Die jeweiligen Frametypen sind in der ersten Zeile angegeben. Die Werte für j sind in der letzte Spalte aufgeführt und sind bei beiden Algorithmen in diesem Beispiel gleich. Alle Frames, die in einem Schritt in der Endsequenz enthalten sind, sind durch ein `X` markiert. Im realen Programmbetrieb kann man sich vorstellen, daß Frames solange in der angegebenen Reihenfolge verschickt werden, bis die Bandbreite erschöpft ist. In der entsprechenden Zeile sieht man dann die beim Decoder angezeigte Bildsequenz.

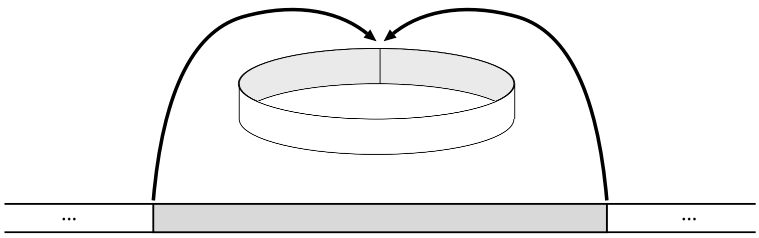


Abbildung 7.6: Umformung einer Teilsequenz in einen Ring aus Frames

7.7.1 Algorithmus A

```
   ibbbpbbbpbbbpbbbp
 1 X----------------   0
 2 X---X------------   4.5
 3 X---X---X--------   2.22222
 4 X---X---X---X----   0.375
 5 X-X-X---X---X----   1.12
 6 X-X-X-X-X---X----   1.11111
 7 X-X-X-X-X-X-X----   0.734694
 8 X-X-X-X-X-X-X---X   0.46875
 9 X-X-X-X-X-X-X-X-X   0.197531
10 XXX-X-X-X-X-X-X-X   0.42
11 XXX-XXX-X-X-X-X-X   0.495868
12 XXX-XXX-XXX-X-X-X   0.486111
13 XXX-XXX-XXX-XXX-X   0.426036
14 XXX-XXX-XXX-XXXXX   0.336735
15 XXXXXXX-XXX-XXXXX   0.231111
16 XXXXXXX-XXXXXXXXX   0.117188
17 XXXXXXXXXXXXXXXXX   0
```

Im ersten Schritt wird das I-Frame gewählt, das ja das einzige unabhängig decodierbare Frame in der Sequenz darstellt. Darauf werden die drei folgenden P-Frames gewählt. Diese müssen natürlich in der Originalreihenfolge ausgewählt werden, da sie ohne ihren Vorgänger nicht decodierbar sind. Man beachte, daß im Schritt 5 nicht das letzte P-Frame gewählt wird, sondern ein $B - Frame$. Dies liegt daran, daß das P-Frame sich ganz am Ende der Sequenz befindet und das Aufbrechen der 4er-Kette weißer Bilder in eine 3er-Kette und das Frame selbst kaum Jitterreduktion bewirkt.

Betrachtet man nun die Schritte 5,6,7,9, sowie 10-14, sieht man, daß der Algorithmus bei mehreren gleichberechtigten Möglichkeiten diese einfach von vorne nach hinten wählt. Wäre die Erweiterung nicht implementiert, wäre das Ergebnis noch schlimmer: Alle Lücken würden ab Schritt 10 von vorne nach hinten aufgefüllt, da die Sequenz in diesem Schritt immer abwechselnd aus schwarzen und weissen Frames besteht. Das ist der grundlegende Unterschied zu Algorithmus B.

7.7.2 Algorithmus B

```
   ibbbpbbbpbbbpbbbp
 1 X----------------   0
 2 X---X------------   4.5
 3 X---X---X--------   2.22222
 4 X---X---X---X----   0.375
 5 X-X-X---X---X----   1.12
 6 X-X-X---X-X-X----   1.11111
 7 X-X-X-X-X-X-X----   0.734694
 8 X-X-X-X-X-X-X---X   0.46875
 9 X-X-X-X-X-X-X-X-X   0.197531
10 X-X-X-X-XXX-X-X-X   0.42
11 X-XXX-X-XXX-X-X-X   0.495868
12 X-XXX-X-XXX-XXX-X   0.486111
13 X-XXXXX-XXX-XXX-X   0.426036
14 X-XXXXX-XXX-XXXXX   0.336735
15 X-XXXXXXXXX-XXXXX   0.231111
16 XXXXXXXXXXX-XXXXX   0.117188
17 XXXXXXXXXXXXXXXXX   0
```

Man sieht, daß sich die Wahl ab Schritt 6 von obigem Algorithmus unterscheidet. Die Framefolge wirkt gleichmäßiger, auch wenn sich das nicht auf den Wert j des Jitters auswirkt.

Beachtet man, daß Algorithmus B einfach zu implementieren ist und ohne Zusatzregeln auskommt, kann er als der bessere bezeichnet werden.

Kapitel 8

Die Programme

Nicht alle der in den vorherigen Kapiteln besprochenen Verfahren konnten im vorgegebenen Zeitrahmen implementiert werden. Vorhanden ist derzeit der UNIX-basierte Server, der bis auf das Zusammenpacken von Encoderpaketen in größere Transmissionpakete komplett ist und ein UNIX-basierter Client, bei dem hauptsächlich die Verarbeitung der Dependency-Kontrollpakete fehlt. Dieser Teil ist aber für den Betrieb nicht unbedingt erforderlich.

8.1 Verwendete Decodersoftware

8.1.1 MSSG-Decoder

Der Client greift auf den MPEG2-Decoder der *MPEG Software Simulation Group* zurück, da dieser Decoder als derzeit einziger den SNR-Scalability-Mode von MPEG2 unterstützt. Allerdings wird dies im Moment noch nicht benötigt, da dazu ein passenden Transcoder auf Serverseite nötig wäre. Die Vor- und Nachteil dieses Decoders sind:

– Er ist ziemlich langsam. (auf einem 133 MHz AMD 5k86 ist eine Echtzeitdecodierung nur bis zu einer Bildgröße von etwa 160×120 Pixel möglich).

– Er erwartet seine Eingabe strikt in Dateien und der Eingabeteil des Decoders ist schwierig zu ändern.

– Er decodiert stets so schnell er kann. Eine mit einer Uhr synchrone Decodierung ist nicht ohne große Modifikation möglich.

\+ Der Source-Code ist verfügbar und damit kann der Decoder prinzipiell in den Client integriert werden.

Der Client schreibt seine Ausgabe in eine Datei und startet den Decoder als eigenen Prozess mit dieser Datei als Eingabe. Der Decoder wurde dazu so modifiziert, daß er nicht abbricht, wenn das Dateiende erreicht wurde. Da die Daten in Echtzeit in die Datei geschrieben werden, wird der Decoder soweit gebremst, daß er das Video auch in Echtzeit anzeigt. Leider funktioniert dies nicht perfekt, da in MPEG immer ein paar Bilder im voraus decodiert werden müssen. In der Praxis sieht das dann so aus, daß er immer eine Gruppe von etwa 5 Bildern sehr schnell decodiert und dann eine Pause einlegt. Der Vorteil davon, daß die Daten gleichzeitig in eine Datei geschrieben werden, ist, daß man sich den empfangenen Datenstrom mehrmals ansehen kann und man das komplette Video auch dann sehen kann, wenn der Decoder die Decodierung nicht in Echtzeit schafft. Dies war während der Programmentwicklung sehr hilfreich.

8.1.2 mtv-Decoder

Alternativ kann der Client auch den kommerziellen mtv-Decoder für UNIX-Systeme ansteuern. Dazu wird der Datenstrom in die Standardeingabe des mtv-Programms geleitet. Vor- und Nachteile dieses Decoders sind:

+ Er ist sehr schnell.

+ Er decodiert bei MPEG-System-Streams Video und Audio und hält diese synchron.

– Für ihn ist kein Source-Code verfügbar und er ist damit auch nicht besser an den Client zu koppeln.

8.2 Implementierungsdetails

Server und Client wurden in C++ implementiert. Beide Programme benötigen POSIX-Threads, lassen sich aber auch an andere Thread-Implementierungen anpassen, da sämtliche threadspezifischen Funktionen in eigene Klassen gekapselt wurden. Dies ist insbesondere deshalb wichtig, da noch eine Implementierung des Client für Microsoft Windows folgen soll, bei dem die Threadaufrufe anders aufgebaut sind.

Jedesmal, wenn ein Client eine Verbindung zum Server aufbaut, wird ein ganz neuer Server-Prozess gestartet, der dann die Übertragung zu diesem Client durchführt. Dadurch können beliebig viele Clients zugleich auf den Server zugreifen. Dabei sollte man aber nicht vergessen, daß jeder Server auch unabhängig Rechenzeit für jeden Client benötigt, so daß Massenübertragungen nicht anzuraten sind.

Im Server laufen der Serveralgorithmus, der ServerMain-Teil und jeder Encoder als eigene Threads. Der Serveralgorithmus besteht dabei wieder aus zwei Threads. Einer davon ist für das Versenden der Daten zuständig, der andere bearbeitet die ReceptionReport-Messages und sortiert alte Daten aus dem Sende-Heap aus (siehe Abbildung 8.1).

Auch der Client besteht aus mehreren Threads (siehe Abbildung 8.2). Der Receiver-Thread wartet ständig, bis an einem Eingabe-Socket Daten empfangen wurden. Diese gibt er dann an den zuständigen Receiver-Algorithmus weiter. Jeder Receiver-Algorithmus läuft wieder in einem eigenen Thread, der hauptsächlich darauf wartet, Daten an den Decoder weiterschicken zu können. Die Decoder laufen entweder als eigene Threads oder (wie in der jetzigen Implementation) sogar als eigene unabhängige Prozesse. Des weiteren laufen derzeit noch die Reception-Log-Routinen als eigene Threads, dies könnte aber auch in den Receiver-Thread integriert werden, was wohl in einer späteren Programmversion erfolgt.

Die ReceptionReport-Messages unterscheiden sich in der jetzigen Programmversion leicht von dem in Abschnitt 6.10.4 beschriebenen Aufbau und die Semantik differiert leicht. Die Unterschiede sind aber nicht wesentlich, eine nächste Programmversion wird das neue Messageformat benutzen.

8.3 RTP-Protokoll

In der aktuellen Programmversion werden alle gesendeten Pakete nochmals in RTP konforme Nachrichten eingepackt. Ob das allerdings in den weiteren Programmversionen erhalten bleiben wird, ist ungewiß, da das Anwendungsgebiet von RTP vor allem in einem Multicasting liegt und nicht in der individuellen Clientversorgung.

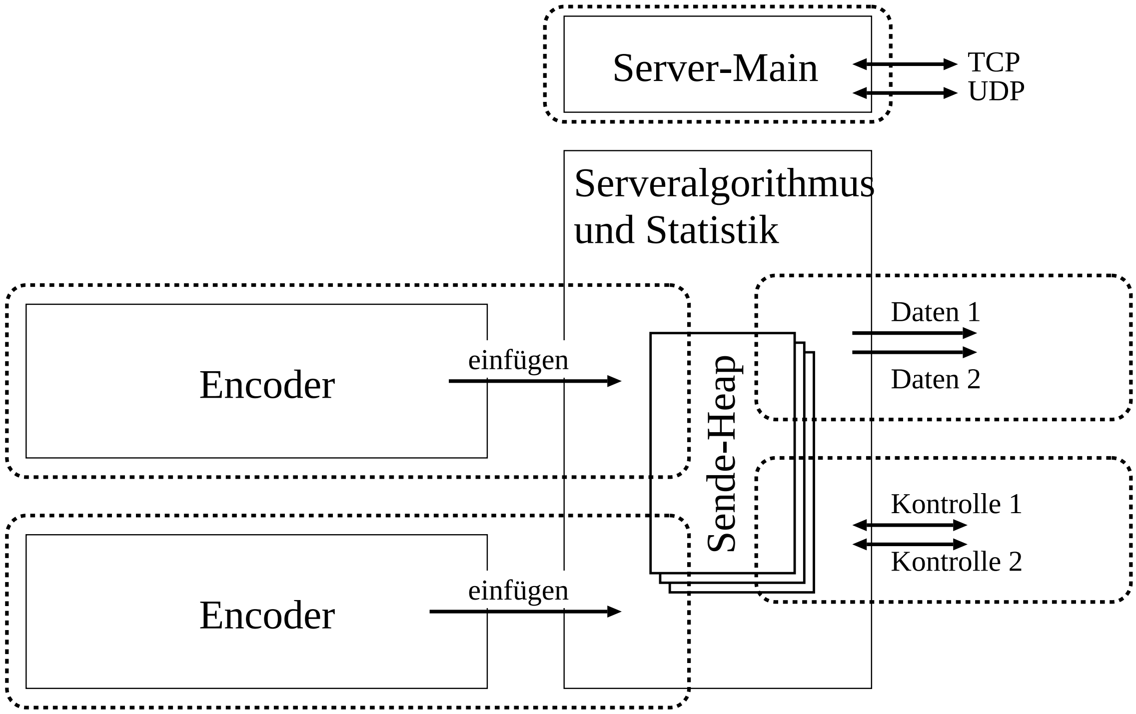


Abbildung 8.1: Im Server laufende Threads (gepunktet).

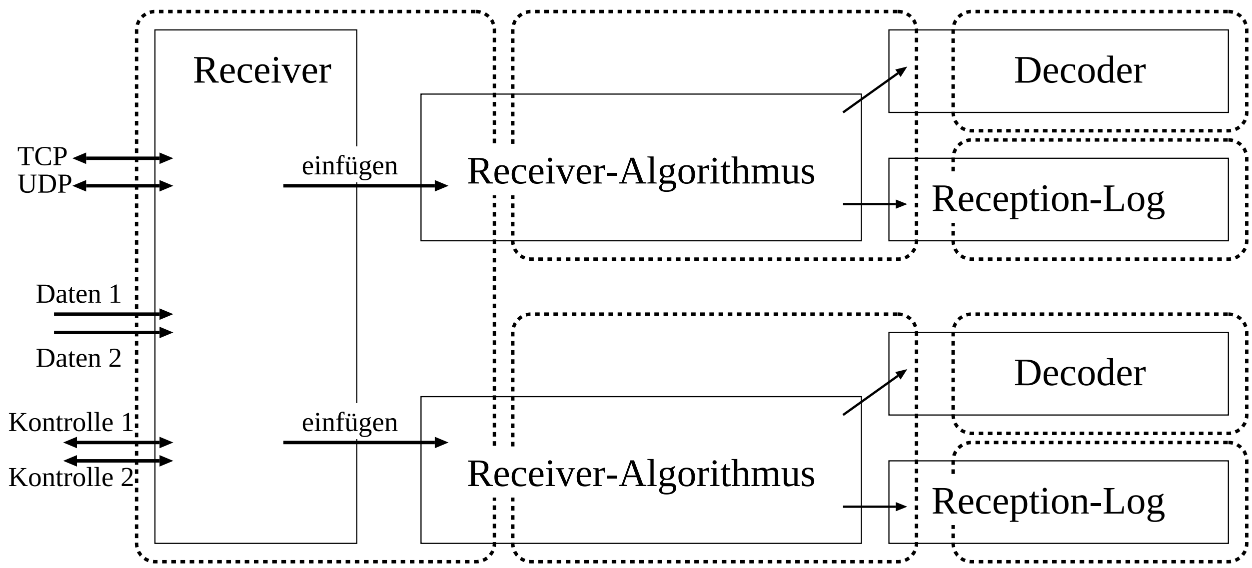


Abbildung 8.2: Im Client laufende Threads (gepunktet).

8.4 Programmaufruf

8.4.1 Beispiel eines Verbindungsaufbaus

Der Server `ms` wird auf einem UNIX-Rechner gestartet (in diesem Falle hieß der Rechner `piano`). Mit der Option `-f <name>` wurde der MPEG-Stream ausgewählt, der zu senden ist. Da keine spezielle Portadresse angegeben wurde, sucht sich der Server selbst eine freie heraus. Er meldet sich nun wie folgt:

```
dirk@piano:> ms -f ../../mpegs/gen.m1v
----------------------------------
MPEG-Stream:   ../../mpegs/gen.m1v
System-Stream: false
Port:          variabel
----------------------------------
waiting for client to connect to MainSocket at 192.0.0.1:1029 ...
```

Er hat sich also die Portadresse 1029 ausgesucht. Wir starten nun unseren Client (hier auf dem Rechner `chopin`) und übergeben ihm den Namen des Serverrechners und den Port des Servers. Der Client baut daraufhin die Verbindung auf und die Portnummern der Daten- und Kontrollsockets werden ausgetauscht:

```
dirk@chopin> cl -s piano -p 1029
----------------------------------
server port:  1029
loss rate:    0
buffer delay: 5sec
----------------------------------
connecting to 192.0.0.1:1029 ... OK
Client Socket: 192.0.0.2:1142
Server RTP Port:  1030
Client-RTP-Port:  1143
Server RTCP Port: 1031
Client-RTCP-Port: 1144
```

Auf Serverseite erscheint eine entsprechende Meldung:

```
 OK, main server-connection established
 main server-socket is connected to 192.0.0.2:1142
RTP-Adressen austauschen ---------------
Server RTP-port:  1030
Client RTP-port:  1143
RTCP-Adressen austauschen ---------------
Server RTCP-port: 1031
Client RTCP-port: 1144
server-START
```

Daraufhin startet der Client den Decoder und die Übertragung beginnt.

8.4.2 Server-Optionen

```
 MPEG-Streaming-Server      1997 Dirk Farin
---------------------------------------------
f <file>   - Eingabefile
s          - Eingabe ist System-Stream
p <port>   - Server-Main Port festlegen
m          - Server kann mehrere Clients gleichzeitig bedienen
l <class>  - Logging fuer die angegebene Nachrichtenklasse anschalten
v <level>  - Verbosity-Level fuer nachfolgende '-l's
h          - Hilfe

Vorhandene Logging-Klassen: rtp-send, rtp-recv, rtp, rtcp-send, rtcp-recv, rtcp,
                            threads, mpeg, recvrep, decdata
```

Option `-p` erlaubt es, den Server an einen festen Port zu binden, falls dieser noch nicht belegt ist. Dies ist nützlich, wenn man einen Serverdienst fest installieren möchte, allerdings stört es bei der Entwicklung sehr, da Portadressen erst einige Zeit nach dem Programmende wieder freigegeben werden (Linux und SunSolaris).

Mit Option `-m` erlaubt man es, daß sich mehrere Clients an einen Server verbinden. Wird diese Option nicht angegeben, so wird kein eigener Serverprozeß erzeugt, wenn eine Verbindung aufgebaut wird. Das erleichtert die Fehlersuche mit dem Debugger.

Auch zur Fehlersuche kann man Loggingmeldungen über verschiedene Programmteile in unterschiedlichen Ausführlichkeiten anschalten. Dies geschieht, indem man mittels Option `-v` den Umfang der Meldungen festlegt (0 - keine Meldungen, ...) und dann mit einer `-l` Option den Programmteil wählt, der Loggingmeldungen erzeugen soll. Alle verfügbaren Programmteile sind in der Hilfeseite aufgelistet. Für jeden Programmteil kann die Ausführlichkeit getrennt eingestellt werden, indem jeder `-l` Option eine `-v` Option vorausgeht.

8.4.3 Client-Optionen

```
 MPEG-Streaming-Client      1997 Dirk Farin
---------------------------------------------
s <addr>     - Server-Adresse
p <port>     - Server-Port
f <lossrate> - Anzahl der zufaellig verlorenen RTP-Packets in Prozent
d <delay>    - Groesse des Empfangsbuffers in Sekunden
l <class>    - Logging fuer die angegebene Nachrichtenklasse anschalten
v <level>    - Verbosity-Level fuer nachfolgende '-l's
h            - Hilfe

Vorhandene Logging-Klassen: rtp-send, rtp-recv, rtp, rtcp-send, rtcp-recv, rtcp,
                            threads, mpeg, recvrep, decdata
```

Hier ist besonders interessant, daß die Empfangsbuffergröße bei Programmstart mit `-d` frei eingestellt werden kann. Für Testläufe ist die Option `-f` interessant, mit der man einstellen kann, wieviele der empfangenen Pakete sofort weggeworfen werden sollen. Dies ist sehr nützlich, da die bei der Programmentwicklung verfügbaren Netzwerkverbindungen eine so hohe Bandbreite hatten, daß fast keine Pakete verlorengingen. Allerdings ist dieser Parameter etwas dürftig, da die Verlustrate nicht von der Paketgröße abhängig angegeben werden kann. Dies wird in einer nächsten Programmversion erweitert werden.

8.5 Ergebnisse

Noch liegen keine aussagekräftigen Ergebnisse vor. Mit der jetzigen Programmversion ist auf lokalen Netzen auch noch mit (künstlich durch die `-f` Option erzeugten) Paketverlusten von über 75% eine fast fehlerfreie Videoübertragung möglich. Dieser Wert ist aber nicht sehr aussagekräftig, da die Paketübertragungszeiten sehr kurz sind und ein Paket somit irgendwann einmal durchkommen wird. Auch ist die Netzwerkbandbreite im Vergleich zum zu übertragenden Video so hoch, daß praktisch immer der komplette Videostream geschickt werden konnte.

Praxisnähere Ergebnisse werden erst dann verfügbar sein, wenn die Übertragung mit stark eingeschränkter Netzwerkbandbreite (z.B. durch eine Modemverbindung) möglich ist.

8.6 Was noch zu tun bleibt

Wie oben angemerkt, ist die Implementierung im aktuellen Stadium noch nicht komplett. Dies wird in nächster Zeit zu vervollständigen sein. Derzeit fehlt die Audiounterstützung infolge eines passenden Encoders noch ganz, auch bleibt die Frage, wie eine hierarchische, komprimierende Audiocodierung auszusehen hat.

Außerdem ist es noch geplant, einen Client für Microsoft Windows zu erstellen, um den Dienst einer breiteren Öffentlichkeit verfügbar zu machen. Dazu ist geplant, auf die MPEG-Audio- und Video-Decoder aufzubauen, die Microsoft in *DirectShow* implementiert hat und die gut an andere Programme ankoppelbar sind.

Kapitel 9

Erweiterungsmöglichkeiten

9.1 Fraktale Bildcodierung

Es wäre interessant zu untersuchen, wie sich andere Videokomprimierungsverfahren mehrschichtig codieren lassen. Eine Grundidee für fraktal komprimierte Videos soll hier kurz angedeutet werden:

Gehen wir davon aus, daß ein Quadtree- oder HV-Partitionierungsverfahren benutzt wird, so läuft die Codierung grob gesprochen so ab:

- Suche für jeden Bildbereich eine möglichst gute Transformation.
- Berechne den Bildfehler e_{rms} in diesem Bildbereich.
- Falls der Bildfehler über einer eingestellten Schwelle liegt und noch nicht die maximale Bitzahl pro Bild codiert wurde, unterteile den Bildbereich weiter in kleinere Bildbereiche und codiere diese getrennt.

Man könnte nun ein Video in verschiedene aufeinander aufbauende Layer codieren, indem man das Bild ersteinmal mit einem recht großen Wert e^1_{rms} codiert und abspeichert. Danach setzt man die Schwelle tiefer auf e^2_{rms} und unterteilt die Bildbereiche weiter, deren Fehler über dieser Schwelle liegt. In dem zweiten Layer werden dann nur noch die veränderten Transformationen gespeichert. Es läßt sich bestimmt eine Codierungsweise finden, in der der Overhead für diese layered Codierung nicht zu groß ist.
Eine HV-Partitionierung in Layer 1 könnte z.B. so aussehen, wie in Abbildung 9.1 gezeigt und diese Partitionierung um Layer 2 ergänzt so, wie in Abbildung 9.2. Eine Übertragung wäre dann mit dem vorhandenen Server-Algorithmus ohne Modifikation möglich.

9.2 Mehrschichtige Audiocodierung

Noch existieren keine brauchbaren Ansätze für eine mehrschichtige Audiocodierung. In der Literatur wird in dieser Hinsicht nur auf den trivialen Fall der mehrschichtigen Codierung von unkomprimierten Audiodaten verwiesen, wie sie auch in Abschnitt 7.5 auf Seite 83 beschrieben wurde. Vielleicht ist es möglich, dieses Verfahren mit der bekannten Subband-Codierung zu verknüpfen: In einer Filterbank zerlege man das Audiosignal in mehrere Frequenzbänder, die zwar gemeinsam wie in MPEG-Audio üblich codiert werden [Zö97], aber deren Ausgabedaten dann für jedes Frequenzband getrennt gespeichert werden. So könnte man vielleicht den Frequenzumfang des empfangenen Signals mit der Anzahl der empfangenen Layer erhöhen.

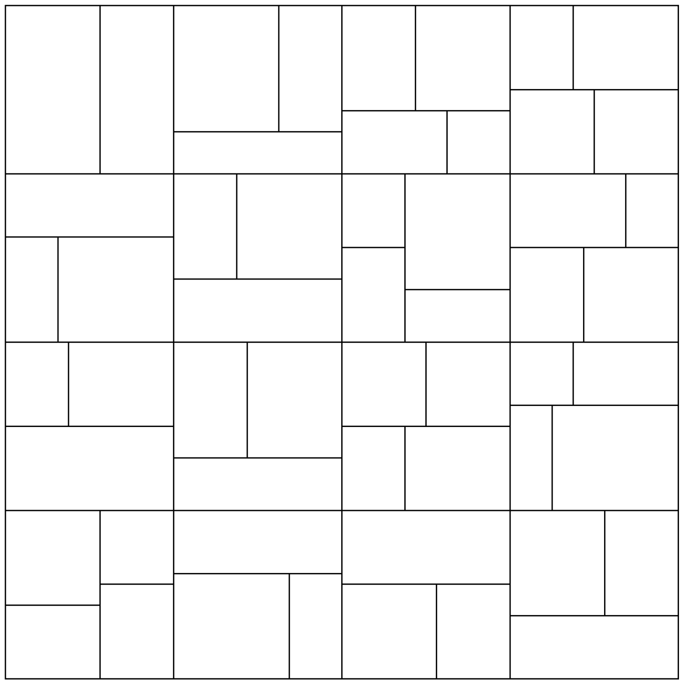


Abbildung 9.1: Layer 1 einer HV-Partitionierung

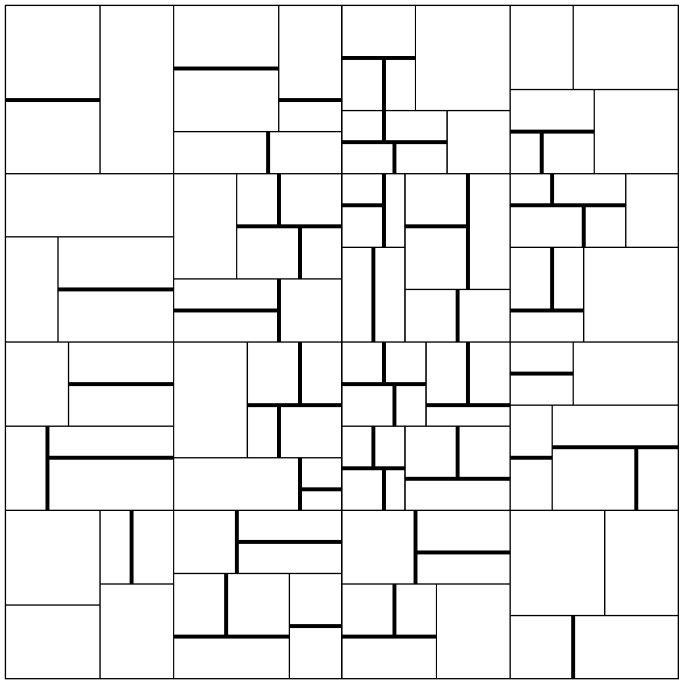


Abbildung 9.2: Layer 1+2 einer HV-Partitionierung

Literaturverzeichnis

[AAN88] Y. Arai, T. Agui, and M. Nakajima. A fast dct-sq scheme for images. *Trans. of the IEICE*, page E 71(11):1095, November 1988.

[ACR96] R. Aravind, M. Reha Civanlar, and Amy R. Reibman. Packet loss resilience of mpeg-2 scalable video coding algorithms. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 426–435, October 1996.

[ATS95] United States Advanced Television Systems Committee. *Digital Audio Compression (AC-3) Standard*, April 1995.

> Der offizielle AC-3 Standard. Ein sehr kompaktes aber ausreichendes Dokument. Vorkenntnisse über digitale Audiokompression sollte vorhanden sein.

[BTW] Jean-Chrysostome Bolot, Thierry Turletti, and Ian Wakeman. Scalable feedback control for multicast video distribution in the internet. Technical report, INRIA Sophia-Antipolis Cedex, France.

[BVG96] Jean-Chrysostome Bolot and André Vega-García. Control mechanisms for packet audio in the internet. In *IEEE Infocom '96 Vol. 1*, 1996.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[DHT] Christophe Diot, Christian Huitema, and Thierry Turletti. Multimedia applications should be adaptive. Technical report, INRIA Sophia Antipolis, France.

[Fis94] Yuval Fisher. *Fractal Image Compression.* Springer-Verlag, 1994.

[FL90] E. Feig and E. Linzer. Discrete cosine transform algorithms for image data compression. *Proceedings Electronic Imaging '90 East*, pages 84–87, 1990.

[GMS94] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *Der LaTeX Begleiter.* Addison-Wesley, 1994.

[H26] *Video Codec for Audiovisual Services at p×64 kbits, ITU-T Recommendation H.261.*

[HPN97] Barry G. Haskell, Atul Puri, and Arun N. Netravali. *Digital Video: an introduction to MPEG-2.* Digital Multimedia Standards. Chapman & Hall, 1997.

> Das Standardwerk über MPEG-2-Video. Die Informationen sind sehr umfangreich und detailliert, für eine Implementation dürften sie aber teilweise zu knapp erklärt sein. Auch MPEG-Audio und -Systems werden knapp

dargestellt. Enthalten sind aber zahlreiche weiterführende Kapitel über die Scalability Modes, Digital Networks, HDTV und MPEG-4.

[KT] Marwan Krunz and Satish K. Tripathi. Modeling bit rate variations in mpeg sources. Technical report, University of Maryland Institute for Advanced Computer Studies.

[LCY94] Simon S. Lam, Simon Chow, and David K.Y. Yau. An algorithm for lossless smoothing of mpeg video. Technical report, Department of Computer Sciences, The University of Texas at Austin, February 1994.

[LF91] E. Linzer and E. Feig. New dct and scaled-dct algorithms for fused multiply/add architectures. *ICASSP*, pages 2201–2204, 1991.

[Mä91] Rudolf Mäusl. *Fernsehtechnik*. Hüthig, 1991.

[MPFL96] Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg, and Didier J. LeGall. *MPEG Video Compression Standard*. Digital Multimedia Standards. Chapman & Hall, 1996.

Das Standardwerk über MPEG-1-Video. Die Informationen sind so umfangreich, daß sie für eine Implementation eines Codecs ausreichen. Enthalten ist auch eine Einführung in MPEG-2, sowie weitergehende Kapitel über effiziente Motion-Vector Suche, Variable Quantisierung und andere Videokompressions-Standards wie H261.

[MPGa] *Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbit/s, ISO/IEC 11172-2: Video.*

[MPGb] *Generic Coding of Moving Pictures and Associated Audio Information: Video, ISO/IEC 13818-2: Draft International Standard.*

[Net] ftp://mm-ftp.cs.berkeley.edu/pub/papers/network.ps.gz.

[PM93] William B. Pennebaker and Joan L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrad Reinhold, 1993.

Das Standardwerk über JPEG. Ein sehr ausführliches Buch, das die Teile 1 und 2 des JPEG-Standards komplett im Anhang enthält. Ein sehr großer Teil des Buches beschäftigt sich mit der Arithmetischen Codierung.

[PSR93] Ketan Patel, Brian C. Smith, and Lawrence A. Rowe. Performance of a software mpeg video decoder. Technical report, Computer Science Division-EECS, University of California, 1993.

[RFCa] *RTP: A Transport Protocol for Real-Time Applications, RFC 1889.*

[RFCb] *RTP Payload Format for H.261 Video Streams, RFC 2032.*

[RFCc] *RTP Payload Format for MPEG1/MPEG2 Video, RFC 2038.*

[RKR96] P. Venkat Rangan, Srihari Sampath Kumar, and Sreerang Rajan. Continuity and synchronization in mpeg. *IEEE Journal on selected Areas in Communications*, pages 52–60, January 1996.

[Rog83] B.E. Rogowitz. Spatial/temporal interactions: Backward and forward metacontrast masking with sine wave gratings. *Visual Research*, pages 23(10):1057–73, 1983.

[SKZ96] Huifang Sun, Wilson Kwok, and Joel W. Zdepski. Architectures for mpeg compressed bitstream scaling. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 191–199, April 1996.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated Volume 1*. Addison-Wesley professional computing series. Addison-Wesley, 1994.

[Ste95] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley professional computing series. Addison-Wesley, 1995.

[TB] Thierry Turletti and Jean-Chrysostome Bolot. Issues with multicast video distribution in heterogeneous packet networks. Technical report, INRIA Sophia-Antipolis Cedex, France.

[TPB] Thierry Turletti, Sacha Fosse Parisis, and Jean Bolot. Experiments with a layered transmission scheme over the internet. Technical report, INRIA Sophia-Antipolis Cedex, France.

[Wan95] Brian A. Wandell. *Foundations of Vision*. Sinauer Associates, Inc, Sunderland, Massachusetts, 1995.

[WSS96] Marcelo K. Weinberger, Gadiel Seroussi, and Guillermo Sapiro. Loco-i: A low complexity, context-based, lossless image compression algorithm. *Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March-April 1996*, 1996.

[Zö97] Udo Zölzer. *Digitale Audiosignalverarbeitung*. B.G.Teubner Stuttgart, 1997.

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

____

(Dirk Farin)