Universität Bonn

Institut für Informatik, Abteilung IV

Praktikumsbericht
zum Projekt
“Echtzeit-Audioübertragung mit QoS-Management in einem DiffServ-Szenario”

Thomas Dreibholz (Dreibholz@bigfoot.com)
Jan Selzer (Selzer@cs.uni-bonn.de)
Simon Vey (Vey@cs.uni-bonn.de)

14. August 2000

Inhaltsverzeichnis

Kapitel 1

Einleitung

von Simon Vey

Während das Internet früher fast ausschließlich durch TCP-Verkehr geprägt war, kommen heute und wohl auch in Zukunft immer mehr multimediale Anwendungen hinzu. Diese Anwendungen haben andere Anforderungen als zum Beispiel FTP. Oft ist eine völlig fehlerfreie Übertragung aller Daten nicht unbedingt notwendig, dafür aber zum Beispiel eine maximale Ende-zu-Ende-Verzögerung des Signals.
Das Internet ist ein Best-Effort Netz und gibt als solches keinerlei Servicegarantien, die aber erforderlich wären, um eine erfolgreiche Übertragung eines Multimediastroms sicherzustellen. Zwei Ansätze, Dienstgüte (Quality of Service) im Internet zu gewährleisten, sind Integrated Services (IntServ) und Differentiated Services (DiffServ). Die Idee von IntServ ist es, dynamische Reservierungen auf Ende-zu-Ende Ebene durchzuführen. Router auf dem Pfad von Sender zum Empfänger ordnen dann anhand eines Flowlabels jedes Paket einem reservierten Datenstrom zu und behandeln das Paket in entsprechender Weise. Bei diesem Verfahren kann es aber zu sehr vielen reservierten Datenströmen kommen, wodurch ein Skalierbarkeitsproblem entsteht ([RFC 2208]). Aus diesem Grund wurde DiffServ ([InternetQoS]) entwickelt. Hier werden nicht für jeden Datenstrom eigene Reservierungen vorgenommen, sondern Pakete können unterschiedlichen Serviceklassen zugeordnet werden, die die Router (statisch) zu Verfügung stellen. Je nach Serviceklasse wird das Paket in den Routern bevorzugt oder weniger bevorzugt behandelt. Strikte Garantien können aber auch so nicht gegeben werden.
Aufbauend auf DiffServ wurde im Rahmen dieses Praktikums ein Quality-of-Service-Management (QoS-Management) entwickelt, das strom- und anwendungsübergreifend die Transportströme (ein Anwendungsstrom kann aus mehreren Transportströmen bestehen) auf die zur Verfügung stehenden Serviceklassen abbildet. Hierbei müssen die QoS-Anforderungen der einzelnen Ströme und die aktuellen Eigenschaften der Klassen berücksichtigt werden. Außerdem soll die Zuordnung der Ressourcen möglichst fair ablaufen (siehe hierzu Abschnitt: 2.3.3). Die Informationen über die einzelnen Service-Klassen werden von einem Reservierungsmodul bereitgestellt, das die Reservierungsschnittstelle zum QoS-Management darstellt. Auf diese Weise soll von verschiedenen Reservierungsverfahren abstrahiert werden. Das Reservierungsmodul wurde von Jan Selzer entwickelt, der sich außerdem um die Bereitstellung der Meßumgebung sowie die DiffServ Implementation gekümmert hat. Thomas Dreibholz programmierte die Server- und Clientanwendungen (RTP Audio) sowie die benötigten RTP-Transportmodule und Meßwerkzeuge. Der QoS-Manager

wurde von Simon Vey entwickelt.
Die folgenden Kapitel sind wie folgt gegliedert: In Kapitel 2 werden die nötigen Grundlagen besprochen. Kapitel 3 beschreibt dann das in diesem Praktikum entwickelte System, wobei am Anfang des Kapitels zuerst ein Überblick über das Gesamtsystem gegeben wird. In den weiteren Abschnitten des Kapitels werden dann die einzelnen Komponenten ausführlicher beschrieben. Kapitel 4 beinhaltet Meßergebnisse, die die Funktionalität des Systems herausstellen sollen, sowie eine Beschreibung der dazu benötigten Meßwerkzeuge. Abschließend folgt eine Zusammenfassung und ein Ausblick darüber, welche Aspekte in Zukunft noch behandelt werden können.

Kapitel 2

Die Grundlagen

Dieses Kapitel gibt eine Einführung in die Grundlagen des DiffServ-Routers, des RTP Audio-Systems und des QoS-Managers. Es werden zuerst die für das Verständnis der folgenden Unterkapitel notwendigen Netzwerkprotokolle beschrieben; darauf folgen die Grundlagen des DiffServ-Routers und des QoS-Managements. Das letzte Unterkapitel beschreibt die Grundlagen der digitalen Audioaufzeichnung.

2.1 Die grundlegenden Netzwerkprotokolle

von Thomas Dreibholz

In diesem Unterkapitel wird eine Einführung in die Netzwerkprotokolle gegeben, auf welchen der DiffServ-Router, RTP Audio und der QoS-Manager aufbauen.

2.1.1 Einführung

Der Transport von Daten über das Internet (z.B. mit FTP) ist eine komplexe Aufgabe. Es ist daher sinnvoll, diese in aufeinander aufbauende Schichten (Layers) zu zerlegen, welche jeweils ein Teilproblem lösen (Hierarchieprinzip). Dabei benutzt die n-te Schicht über eine genau defininierte Schnittstelle (Interface) die genau definierten Leistungen der *n-1*-ten Schicht und erbringt selbst wieder genau definierte Leistungen über ihre Schnittstelle für die *n+1*-te Schicht. Schicht n auf Station A kommuniziert dabei mit Schicht n auf Station B über ein genau definiertes Protokoll. Einzelne Schichten lassen sich - sofern sich Leistungen und Schnittstellen nicht ändern - problemlos austauschen.

Die wichtigsten Schichtenmodelle sind das Reference Model for Open Systems Interconnection (OSI) der International Standards Organization (ISO) in Tabelle 2.1 und das Referenzmodell der [IETF] in Tabelle 2.2, wobei letzteres eine Vereinfachung des OSI-Modells auf vier Blöcke darstellt:

Internet: Diese Schicht transportiert ein Paket vom Quellrechner zum Zielrechner. Die wichtigsten Protokolle hierfür sind z.B. IPv4 und IPv6 (siehe Abschnitte 2.1.2 und 2.1.3).

Transport: Auf dieser Ebene können Verlust- und Fehlererkennung sowie Fluß- und Congestionkontrolle realisiert werden. Protokolle sind hierfür z.B. TCP (Abschnitt 2.1.5) und UDP (Abschnitt 2.1.4).

7	Application Layer
6	Presentation Layer
5	Session Layer
4	Transport Layer
3	Network Layer
2	Data Link Layer
1	Physical Layer

Tabelle 2.1: Das OSI Referenzmodell

7 6 5	Application Layer	z.B. HTTP, FTP, Telnet oder RTP AUDIO
4	Transport	z.B. UDP, TCP
3	Internet	z.B. IPv4, IPv6
2	Host to Network	z.B. Ethernet oder UMTS

Tabelle 2.2: Das Referenzmodell der [IETF]

Die Protokolle der Internet- und Transportschicht werden von der Internet Engineering Task Force [IETF] "standardisiert", ihre Beschreibung ist in den Requests for Comments (RFCs) zu finden, welche bei [IETF] und [RFC Editor] zu downloadbar sind.

Application: In dieser Ebene sind die anwendungsspezifischen Protokolle, wie z.B. FTP, HTTP, Telnet und natürlich RTP AUDIO/RTP (siehe 2.1.7) zu finden. Viele Protokolle dieser Ebene sind ebenfalls in den RFCs (siehe oben) beschrieben.

Host to Network: Hier wird die eigentliche Übertragung von Paketen der Internetschicht geregelt.

2.1.2 Das Internet Protocol, Version 4 (IPv4)

Auf der Host to Network-Ebene ist das Internet ein Zusammenschluß vieler unterschiedlicher Teilnetze, basierend auf unterschiedlichsten Technologien (Ethernet, Token Ring, GPRS, UMTS, ATM, Avian Carrier[1], ...), welche jeweils eigene Adreßformate und Paketgrößen besitzen. Um Pakete über all diese verschiedenen Netzwerke hinweg versenden zu können, wurde das Internet Protocol (IP) in [RFC 791] definiert. Dieses Protokoll stellt mittels eindeutiger IP-Adresse und Fragmentierung sicher, daß ein Paket von einem Rechner im Netzwerk 1 durch beliebige andere Netzwerke zu einem anderen Rechner in Netzwerk 2 geleitet wird, wobei - falls ein Teilnetz nur geringere Paketgrößen unterstützt - gegebenenfalls große Pakete in mehrere kleine zerlegt (Fragmentierung) und am Zielrechner wieder zusammengesetzt werden.

IP selbst erbringt nur einen Best Effort-Dienst, d.h. es wird nicht garantiert, daß das Paket auch ankommt (z.B. aufgrund von Überlast). Dies - falls notwendig - durch Quittungen und Wiederholungen sicherzustellen, ist Aufgabe der Transportschicht (z.B. mit TCP, siehe Kapitel 2.1.5).

Aktuell ist Version 6 des IP-Protokolls (IPv6), Version 4 (IPv4) besitzt jedoch im Moment noch die größte Verbreitung. In den Tabellen 2.3 und 2.4 sind die Header von IPv4 und IPv6 zu sehen. Die Felder im IPv4-Header haben dabei die folgenden Bedeutungen:

Version: Dies ist die IP-Versionsnummer (4).

[1] Siehe dazu [RFC 1149] und [RFC 2549]...

Internet Header Length (IHL): Hier ist die Header-Länge in 32-Bit Words gespeichert. Dem eigentlichen IPv4-Header können weitere Optionen - z.B. für Routing - folgen. Daher ist seine Länge nicht konstant.

Total Length: Dies ist die Gesamtlänge des Paketes (Header + Payload).

Identification, Flags, Fragment: Wenn ein Paket vom Sender oder einem Router fragmentiert wird, muß es der Zielrechner auch wieder zusammensetzen können. *Identification* identifiziert das Paket, zu dem das Fragment gehört (*Identification* ist bei allen Fragmenten eines Paketes gleich); *Fragment Offset* ist die Position des Fragmentes innerhalb des Paketes. *MF* (More Fragments) ist gesetzt, falls dem Fragment weitere folgen; *DF* (Don't Fragment) ist gesetzt, wenn das Paket nicht fragmentiert werden darf.

Protocol: Hier wird die Nummer des Protokolls für den Payload gespeichert, z.B. 6 für TCP oder 17 für UDP.

Checksum: Dies ist eine Prüfsumme für den IPv4-Header (nicht jedoch für den Payload!). Zu ihrer Berechnung siehe [RFC 1071].

Time to Live (TTL): Dieses Feld gibt die Anzahl von Routern an, über die das Paket noch laufen darf. Jeder Router dekrementiert diesen Wert um 1, bei *TTL*=0 wird das Paket verworfen. Damit wird verhindert, daß ein Paket "ewig" zwischen zwei fehlerhaften Routern hin und her laufen kann. Zu beachten ist, daß das Dekrementieren die Prüfsumme des Paketheaders ändert. Ein Router muß diese daher immer neu berechnen und setzen!
Ursprünglich sollte in diesem Feld die verbleibende Lebenszeit des Paketes in Sekunden angegeben werden, daher die Bezeichnung *Time to Live*. In IPv6 wird stattdessen die passendere Bezeichnung *Hop Limit* verwendet.

Source Address, Destination Address: Dies sind die Quell- und Zieladresse (jeweils 32 Bit für IPv4).

Type of Service (TOS): Hiermit wird die DiffServ-Behandlung des Paketes in Routern durch Priorität und Serviceklasse festgelegt. Dieses Feld wird auch als *DS Field* oder *Traffic Class* bezeichnet, wobei im folgenden für dieses Byte immer die Bezeichnung *Traffic Class* verwendet werden wird. Es wird ausführlich im Kapitel 2.2 behandelt. Zudem sei auf die RFCs [RFC 2474] und [RFC 1349] (alte Version) verwiesen.

2.1.3 Das Internet Protocol, Version 6 (IPv6)

Seit der Spezifikation von IPv4 im September 1981 in [RFC 791] hat sich das Internet sehr stark vergrößert. Dies führte unter anderem zu folgenden Problemen:

- Der 32-Bit-Adreßraum mit $2^{32} = 4{,}294{,}967{,}296$ Adressen ist zu klein. Zudem ist die Verteilung sehr ineffizient, daher können wesentlich weniger Adressen auch wirklich verwendet werden.

- IPv4 ist schlecht erweiterbar - es können dem Header zwar Optionen hinzugefügt werden, jedoch muß Kompatibilität zu alter Soft- und Hardware sichergestellt sein.

Länge	Inhalt
4 Bit	Version (4)
4 Bit	Internet Header Length
8 Bit	Type of Service (⇔ Tr. Class)
16 Bit	Total Length
16 Bit	Identification
3 Bit	Flags: unbelegt/DF/MF
13 Bit	Fragment Offset
8 Bit	Time to Live (⇔ Hop Count)
8 Bit	Protocol (⇔ Next Header)
16 Bit	Checksum (nur für **Header**)
32 Bit	Source Address
32 Bit	Destination Address
variabel	Optionen

Tabelle 2.3: Der IPv4-Header

Länge	Inhalt
4 Bit	Version (6)
8 Bit	Traffic Class (⇔ TOS)
20 Bit	Flowlabel
16 Bit	Payload Length
8 Bit	Next Header (⇔ Protocol)
8 Bit	Hop Limit (⇔ Time to Live)
128 Bit	Source Address
128 Bit	Destination Address

Tabelle 2.4: Der IPv6-Header

Länge	Inhalt
8 Bit	Next Header
8 Bit	Extension Header Length

Tabelle 2.5: Der IPv6 Erweiterungs-Header

Länge	Inhalt
8 Bit	Type
8 Bit	Length
variabel	Value

Tabelle 2.6: TLV-Format einer IPv6 Option

- Routing mit IPv4 benötigt viel Rechenzeit: Große Routingtabellen, variable Headerlänge, notwendiges Neuberechnen der Prüfsumme (wegen TTL). Dies wird vor allem bei schnellen Netzwerken (Gigabit, Terabit, ...) zum Problem.

- Außer Traffic Class (TOS-Feld): Keine Unterstützung für Echtzeitverkehr.

Im Dezember 1998 wurde dazu in [RFC 2460] IPv6[2] standardisiert. Dabei gibt es folgende grundlegende Unterschiede zu IPv4:

- Erweiterter Adreßraum mit 128 Bit:
  2^{128} = 340,282,366,920,938,463,463,374,607,431,768,211,456 Adressen. Dies sind - selbst beim ineffizientesten Verteilungsszenario - noch ca. 1000 Adressen pro Quadratmeter Erdoberfläche; bei realistischeren Szenarien dagegen viele Billionen.

- Fragmentierung ist nur noch auf dem Quellrechner erlaubt. Kann ein Router das Paket aufgrund seiner Länge nicht verarbeiten, so verwirft er es und sendet eine Fehlermeldung zurück (siehe Abschnitt 2.1.6). Zudem entfällt die Checksumme. Damit sind IPv6-Pakete wesentlich effizienter zu routen.

- IPv6 ist durch Erweiterungsheader sehr einfach erweiterbar. Die eigentliche IPv6 Headerlänge ist damit konstant (40 Bytes).

- Flowlabels zum Markieren von Strömen (siehe unten).

- Unterstützung für automatische Konfiguration (siehe [RFC 2462]) .

- Unterstützung von Mobilfunk (siehe [3GPP]).

In den Tabellen 2.4 und 2.6 sind ein IPv6-Header sowie ein Erweiterungsheader zu sehen, wobei die einzelnen Felder folgende Bedeutungen haben:

Version: Dies ist die IP-Versionsnummer (6).

Traffic Class: [3]Dies ist der Traffic Class-Wert. Es ist äquivalent dem *Type of Service*-Feld in IPv4 (siehe Beschreibung zu TOS bei IPv4).

Payload Length: Dies ist die Länge des Payloads, also die Paketlänge - 40 Bytes IPv6-Header.

Next Header: Hier steht die Protokollnummer des nächsten Headers. Dies kann ein IPv6-Erweiterungsheader sein (z.B. Fragmentierung oder Routing) oder analog zu IPv4 *Protocol* die Protokollnummer für den Payload (z.B. UDP oder TCP).

Hop Limit: Dieses Feld gibt äquivalent zum *Time to Live*-Feld in IPv4 die Anzahl von Routern an, über die das Paket noch laufen darf.

[2]IPv5 war schon belegt durch ein experimentelles Realtime-Stream-Protokoll.

[3]In alten RFCs hieß dieses Feld Priority und besaß nur 4 Bits. Dies wurde zur Erhaltung der Kompatibilität mit IPv4 TOS in RFC 2460 geändert.

Source Address, Destination Address: Dies sind die Quell- und Zieladresse (jeweils 128 Bit für IPv6).

Flowlabel: Das Flowlabel ist eine 20-Bit[4] lange, zufällig aus [0x00001, 0xFFFFF][5] gewählte Zahl, welche einen oder mehrere Ströme von **genau einer** IPv6-Quelladresse zu **genau einer** IPv6-Zieladresse kennzeichnen kann. Zusammen mit der Quelladresse ist dies eine eindeutige Identifizierung. Diese kann z.B. dazu verwendet werden, mit dem Resource ReSerVation Protocol (RSVP, siehe [?]) Bandbreite für diese Ströme zu reservieren. Der Vorteil gegenüber dem Traffic Class-Wert ist, daß es erstens mit 2^{20}-1 = 1048575 Werten deutlich mehr Möglichkeiten gibt und zweitens die Belegung vom Kernel (zumindest unter Linux, siehe 3.2.1) verwaltet wird.
Beispiel: Benutzt ein Sender Φ auf Rechner A das Flowlabel ϑ, um z.B. 5 Ströme zu Rechner B zu senden, so kann niemand anderes dieses Flowlabel ϑ ebenfalls nutzen (der Kernel würde die Anforderung zurückweisen, es sei denn, dies wird ausdrücklich erlaubt). Reserviert man nun die Strecke von Rechner A/Flowlabel ϑ zu Rechner B mit RSVP, so ist die exklusive Nutzung dieser Reservierung für die fünf mit Flowlabel ϑ markierten Ströme des Senders Φ sichergestellt[6]. Im Gegensatz dazu kann jeder beliebige User den Traffic Class-Wert für seine Übertragungen ohne Probleme beliebig setzen.
Durch die Verwendung von Zufallszahlen kann bei Routern eine Hashtabelle für Flowlabels eingesetzt werden, wobei ein Router jedoch nicht annehmen darf, daß die meisten Ströme Flowlabels besitzen. Die Gültigkeitsdauer eines Flowlabels ist begrenzt: Es darf innerhalb der maximalen Lebensdauer nicht für die Markierung eines neuen Stromes verwendet werden. Die genauen Begrenzungen sind durch übergeordnete Protokolle (z.B. RSVP) festzulegen.
Für weitere Informationen hierzu siehe [RFC 2460] und [?].

Der optionale Erweiterungsheader enthält ein *Next Header*-Feld analog zum IPv6-Header; die Länge wird in 8-Bytes-Blöcken - 1 angegeben, daher beträgt die minimale Länge 8 Bytes. Den 2 Bytes des Erweiterungsheaders folgen nun Optionen der Form TLV (Type, Length, Value) - siehe Tabelle 2.6.

Optionen können z.B. Fragmentierungs- oder Routinginformationen sein. Zudem gibt es die Jumbogram-Option: Statt die Payloadlänge in das 16 Bit breite Headerfeld *Length* einzutragen[7], wird hierfür die Jumbogram-Option benutzt, welche ein 32-Bit-Feld für die Länge enthält. Somit werden Längen bis zu 4 GBytes möglich!

2.1.4 Das User Datagram Protocol (UDP)

Das im August 1980 in [RFC 768] definierte User Datagram Protocol (UDP) liegt in der Transportschicht und ermöglicht eine verbindungslose Kommunikation auf Basis von IP. Die UDP-Pakete

[4] In alten RFCs besaß dieses Feld 24 Bits. Dies wurde beim Ersetzen von Priority (4 Bits) durch Traffic Class (8 Bits) in RFC 2460 auf 20 Bits geändert.

[5] 0x00000 steht für kein Flowlabel.

[6] Natürlich könnte noch jemand mittels IP+Flowlabel-Spoofing einen Angriff auf die reservierte Strecke durchführen. Aber vor unprivilegierten Usern sollte diese relativ sicher sein.

[7] Es wird einfach auf 0 gesetzt.

Länge	Inhalt
16 Bit	Source Port
16 Bit	Destination Port
16 Bit	Datagram Length
16 Bit	Checksum

Tabelle 2.7: Der UDP-Header

werden mittels IPv4 bzw. IPv6 verschickt, wobei keine Maßnahmen zur Sicherstellung der korrekten Paketreihenfolge oder Quittierung des Empfanges durch den Sender getroffen werden. Ebensowenig wird für eine Fluß- und Überlastkontrolle gesorgt. Dies sind Aufgaben der auf UDP basierenden Anwendung.

In Tabelle 2.7 ist ein UDP-Header dargestellt, wobei die vier Felder folgende Bedeutungen haben:

Source Port, Destination Port: Diese Werte identifizieren die lokalen Endpunkte der Übertragung (Ports). Zusammen mit den Quell- und Zieladressen im IP-Header sind damit die Dienstzugangspunkte für den Transport eindeutig festgelegt: Während die IP-Adresse den Rechner identifiziert, steht der Port für die Anwendung auf diesem Rechner.

Datagram Length: Dies ist die Länge des UDP-Paketes (incl. UDP-Header, daher also mindestens 8). Durch den 16-Bit-Wert ist die Größe daher auf 64 KBytes beschränkt. Unter IPv6 besteht die Möglichkeit, mittels Jumbograms eine Größe bis zu 4 GBytes zu benutzen, indem dem IPv6-Header ein Jumbogram-Erweiterungsheader angehängt wird, welcher einen 32-Bit-Wert für die Paketlänge enthät. In diesem Fall enthält der UDP-Header als Länge den Wert Null. Näheres hierzu ist in RFC [RFC 2675] zu finden.

Checksum: Dies ist eine Prüfsumme für das UDP-Paket. Unter IPv4 durfte diese durch Setzen von Null[8] weggelassen werden, mit IPv6 ist sie jedoch zwingend erforderlich. Zu ihrer Berechnung siehe [RFC 1071].

2.1.5 Das Transmission Control Protocol (TCP)

Für viele Internet-Anwendungen wie z.B. HTTP, FTP oder Telnet ist ein verbindungsorientierter, zuverlässiger Ende-zu-Ende Transport erforderlich. Dazu wurde im Januar 1980 in [RFC 761] mit Erweiterungen in [RFC 1122] und [RFC 1323] das Transmission Control Protocol (TCP) spezifiziert. Es erfüllt folgende Anforderungen:

- Verbindungsorientiert: Verbindungen werden kontrolliert auf- und wieder abgebaut.
- Zuverlässiger Ende-zu-Ende-Transport: Der Empfänger sendet Quittungen, verlorene Pakete werden wiederholt.
- Flußkontrolle: Der Empfänger signalisiert dem Sender, wann er zur Verarbeitung neuer Daten bereit ist.

[8] Eine wirkliche Prüfsumme 0x0000 wird durch 0xFFFF dargestellt!

- Congestionkontrolle: TCP regelt die Bandbreite, abhängig von Paketverlust, Timeouts, Roundtripzeiten usw.. Somit paßt sich der Strom der aktuellen Belastung des Netzwerkes an.

Da das Thema TCP sehr komplex ist und unser Projekt nur am Rande betrifft, sei für eine ausführliche Behandlung auf die angegebenen RFCs verwiesen.

2.1.6 Das Internet Control Message Protocol (ICMPv4 und ICMPv6)

Zu Steuerungsaufgaben in IP-Netzwerken wurde im September 1981 in [RFC 792] das Internet Control Message Protocol (ICMP, auch als ICMPv4 bezeichnet) für IPv4 und im Dezember 1998 in [RFC 2464] ICMPv6 für IPv6 spezifiziert. Zu den wesentlichen Aufgaben dieses Protokolls gehören:

1. Fehlermeldungen an einen Sender zurückgeben.
   Beispiele:
   - Der Zielrechner wurde nicht gefunden.
   - Der Zielrechner verweigert die Paketannahme (z.B. TCP- oder UDP-Port ist nicht belegt).
   - Der Hop Count ist abgelaufen.
   - Fehler im IP-Header.
   - ...
2. Routerverwaltung - siehe hierzu die angegebenen RFCs.
3. Tests: Ein Rechner sendet einen Echo Request an einen zweiten Rechner, dieser schickt als Antwort einen Echo Reply. Auf diesem Prinzip beruhen unter anderem die Netzwerk-Testprogramme *ping* und *traceroute*.

Da für die Meßwerkzeuge (siehe Abschnitt 3.3) und das QoS-Management (siehe Abschnitt **??**) nur die Testfunktionalität von ICMPv4 bzw. ICMPv6 zur Bestimmung der Roundtripzeiten (siehe unten) notwendig ist, wird auf eine ausführliche Beschreibung der weiteren Aufgaben verzichtet. Diese sind in den angegebenen RFCs zu finden.

Tabelle 2.8 enthält einen ICMPv4/ICMPv6-Header für Echo Requests und Echo Replies. Für diese ICMP-Typen gibt es zwischen ICMPv4 und ICMPv6 keine wesentlichen Unterschiede. Die Bedeutungen der Felder sind dabei folgende:

Type: Hier wird der Typ der ICMP-Nachricht angegeben, also z.B. Echo Request (0 für ICMPv4, 129 für ICMPv6) oder Echo Reply (8 für ICMPv4, 128 für ICMPv6).

Code: 0 für Echo Request und Echo Reply.

Checksum: Dies ist die Prüfsumme für das ICMP-Paket (Header und Payload). Zu ihrer Berechnung siehe [RFC 1071].

Länge	Inhalt
8 Bit	Type (Echo Request/Echo Reply; jeweils für IPv4 und IPv6)
8 Bit	Code (0 für Echo Request/Reply)
16 Bit	Checksum
16 Bit	Identification
16 Bit	Sequence Number
variabel	Payload, z.B. ein Zeitstempel für den Versand des Paketes

Tabelle 2.8: Der ICMPv4/ICMPv6-Header für Echo Requests und Echo Replys

Identification: Diese Identifikationsnummer kennzeichnet den Sender (z.B. seine Prozeß-ID). Einen Wert zu wählen bleibt der Anwendung überlassen.

Sequence Number: Hier wird die Sequenznummer des Paketes eingetragen.

Payload: Hier können beliebige Daten stehen. Diese werden vom einem Empfänger, der den Echo Request empfängt, als Kopie mit dem Echo Reply zurückgeschickt. Sinnvolle Daten sind z.B. ein Zeitstempel für den Zeitpunkt des Echo Request-Versandes. Beim Empfang des Echo Replys läßt sich damit zusammen mit der Ankunftszeit der Antwort die Round Trip Time berechnen. Dies ist die Zeit vom Versand des Requests bis zur Ankunft des Replys, also näherungsweise[9] die doppelte durchschnittliche Transportverzögerung.

2.1.7 Das RTP-Protokoll

Das Realtime Transport Protocol (RTP) und das RTP Control Protocol (RTCP) aus [RFC 1889] (Januar 1996) basieren auf einem Protokoll der Transportebene, wobei in der Regel UDP/IP verwendet wird. Es dient als **Rahmen** zur Implementation des Ende-zu-Ende-Transports von Multimedia-Daten wie Audio und Video. Dabei werden sowohl Multicast- als auch Unicast-Übertragungen unterstützt. Unter anderem werden die Behandlung von Sequenznummern und Zeitstempeln zur Berechnung von Jitter (zur Definition siehe unten) und Paketverlusten sowie Synchronisation und Überwachung der Echtzeitströme mit dem RTP-Protokoll spezifiziert. RTP und RTCP selbst besitzen keine Mechanismen zur Garantie der Transportgüte, Synchronisation sowie Fluß- und Congestionkontrolle - dies zu realisieren ist die Aufgabe der auf RTP/RTCP aufbauenden Protokolle. Das RTCP-Protokoll (siehe Abschnitt 2.1.8) ist dabei für Steuerungs- und Kontrollaufgaben zuständig.

Je Medium (z.B. Audio oder Video) wird eine eigene RTP-Sitzung verwendet. Diese besteht aus mindestens zwei Teilnehmern, welche sich durch jeweils eine sitzungseindeutige SSRC (Synchronization Source) - eine zufällige 32-Bit-Zahl - sowie durch einen global eindeutigen CNAME (Canonical End-Point Identifier) - bestehend aus Benutzer- und Hostnamen - identifizieren. Verschiedene RTP-Sitzungen können für den gleichen Teilnehmer verschiedene SSRCs verwenden, der CNAME bleibt jedoch immer gleich (z.B. user@domain.xy mit *ssrc1* für Audio und *ssrc2* für Video).

[9] Der Rückweg könnte z.B. durch Congestion länger oder kürzer sein. Zudem könnte er über eine andere DiffServ-Klasse (z.B. Best Effort) verlaufen.

Länge	Inhalt
2 Bit	Version (aktuell: 2)
1 Bit	Padding
1 Bit	Extension
4 Bit	CSRC Count
1 Bit	Marker
7 Bit	Payload Type
16 Bit	Sequence Number
32 Bit	Time Stamp
32 Bit	SSRC
variabel	CSRC[0..16]

Tabelle 2.9: Der RTP-Header

Ein RTP-Header hat das in Tabelle 2.9 angegebene Format. Die einzelnen Felder haben dabei folgende Bedeutungen:

Version: Dies ist die Versionsnummer. Aktuell ist momentan 2.

Padding: Für Verschlüsselungen ist es evtl. notwendig, Padding durchzuführen - also das Paket um einige Füllbytes zu erweitern. Ist dieses Bit gesetzt, enthält das Paket am Ende Padding-Bytes, wobei das **letzte** Byte deren Anzahl enthält.

Extension: Bei Bedarf kann der RTP-Header erweitert werden. Durch Setzen dieses Bits wird angezeigt, daß dem RTP-Header ein Erweiterungsheader folgt. Näheres dazu in [RFC 1889].

CSRC, CSRC Count: RTP sieht die Möglichkeit vor, daß Mixer verschiedene Ströme zusammenmischen. Die CSRCs (Contributing Sources) geben dann die SSRCs der Teilnehmer an, die zum Strom beigetragen haben; *CSRC Count* ist ihre Anzahl. Da RTP AUDIO dies nicht verwendet, sei für weitere Informationen auf [RFC 1889] verwiesen.

Marker: Dieses Markierungsbit kann von der Anwendung zur Kennzeichnung von bestimmten Paketen verwendet werden, z.B. das erste Paket eines neuen Frames.

Payload Type: Hier ist der Typ des Payloads vermerkt. In [RFC 1890] sind dazu einige feste Werte, z.B. für Videokonferenzen, definiert.

Sequence Number: Hier befindet sich die Sequenznummer, welche pro Paket um 1 erhöht wird. Sie kann zur Ermittlung der Paketreihenfolge und zur Berechnung von Paketverlusten verwendet werden.

Time Stamp: Dies ist ein Zeitstempel für das Paket, welcher monoton und linear mit der Zeit steigt. Zusammengehörige Pakete (z.B. ein Frame) dürfen den gleichen Zeitstempel enthalten. Er kann zur Synchronisation mit anderen Medien und zur Berechnung des Jitter (siehe unten) verwendet werden. Die Zeiteinheit wird dabei von der Anwendung festgelegt (z.B. $\frac{1}{16}$Millisekunden in RTP AUDIO*)*.

SSRC: Hier ist die SSRC-Nummer des Paketsenders gespeichert.

Wie bereits erwähnt kann mit Hilfe der Sequenznummern der Verlust von Paketen berechnet werden, wobei der dazu nötige Algorithmus schon in [RFC 1889] gegeben ist. Zudem ist durch die Zeitstempel die Berechnung des Jitters (statistische Schwankung der Paketankunftszeiten) möglich, welcher folgendermaßen definiert ist:

$Jitter_{Neu} := Jitter_{Alt} + \frac{1}{16} * (\mid D_{i-1,i} \mid - Jitter_{Alt})$ mit $D_{i,j} = (R_j - R_i) - (S_j - S_i)$,

wobei S_n der RTP Zeitstempel und R_n die Paketankunftszeit des n-ten Paketes sind. Beide Zeitstempel müssen dabei natürlich in den gleichen Einheiten angegeben werden.

2.1.8 Das RTCP-Protokoll

Während das RTP-Protokoll (siehe Abschnitt 2.1.7) hauptsächlich zur Übertragung der eigentlichen Nutzdaten dient, wird für Steuerdaten (z.B. Starten einer Wiedergabe, Ändern der Qualität) das RTP Control Protocol (RTCP) benutzt, welches ebenfalls in [RFC 1889] definiert wird. Zudem ermöglicht es QoS-Monitoring, Fluß- und Congestionkontrolle sowie die Identifikation von Teilnehmern. Bei RTCP gibt es fünf verschiedene Pakettypen:

Receiver Report (RR): Dies ist eine Rückmeldung der empfangenen Qualität von einem Empfänger an den Sender. Sie beinhaltet eine Liste von Reception Reports für jeden empfangenen Strom, mit deren Hilfe der Sender seine Qualität ggf. anpassen kann:

- Paketverluste (aus Sequenznummern berechnet),
- Jitter (aus Zeitstempeln berechnet),
- Zeitstempel des Empfanges des letzten Sender Reports (nächster Typ, siehe unten) und
- Zeit zwischen Empfang des Sender Reports und Versand dieses Receiver Reports.

Sender Report (SR): Dieser Typ wird von aktiven Sendern verschickt und enthält Informationen über den von einem Sender ausgesandten Strom, welche der Empfänger zur Verlust- und Durchsatzberechnung verwenden kann:

- Der aktuelle NTP[10]-Zeitstempel - die Anzahl der Mikrosekunden seit dem 01. Januar 1970,
- der aktuelle RTP-Zeitstempel,
- die Anzahl der gesendeten Pakete und Bytes, und
- falls der Sender auch Empfänger von Strömen ist (z.B. Videokonferenz): Reception Reports analog zum Receiver Report.

Source Description (SDES): Hier sind Informationen über einen Teilnehmer einer RTP Sitzung enthalten, wobei jeweils ein eigener Untertyp verwendet wird. Der wichtigste Untertyp ist der CNAME (siehe oben), welcher immer vorhanden sein muß. Weitere in RFC 1889 [RFC 1889] definierte, optionale Untertypen sind die EMail-Adresse, Telefonnummer, Standort usw. sowie der optionale Untertyp PRIV, welcher applikationsspezifische Daten enthält.

[10]NTP bezeichnet das Network Time Protocol. Genaueres dazu ist in [RFC 1305] zu finden.

BYE: Hiermit signalisiert ein Teilnehmer, daß er die Sitzung verläßt.

APP: Dieser Typ ist applikationsspezifisch. In RTP AUDIO (siehe Abschnitt 3.2) wird er verwendet, um Benutzerbefehle wie Änderung von Qualität, Medium oder Position vom Empfänger an den Sender zu übermitteln.

Mehrere RTCP Reports und SDES-Nachrichten dürfen in einem Paket zusammengefaßt werden (compound packet), um den Transportaufwand zu reduzieren. Um das Netz nicht mit Kontrollnachrichten zu überlasten, sollte der Bandbreiteanteil der RTP-Sitzung für RTCP nicht mehr als 5% betragen. Reports und SDES-Nachrichten werden außerdem in zufälligen Intervallen übertragen. Ein im [RFC 1889] angegebener Algorithmus berechnet dazu das Sendeintervall mittels Zufallsgenerator und Angaben über Teilnehmer und Bandbreiten.

2.2 DiffServ Grundlagen

VON JAN SELZER

2.2.1 IntServ, DiffServ, TOS-Feld

Um die Aufgabe von QoS im Internet zu erfüllen, wurden von der IETF zwei Ansätze vorgeschlagen: Integrated Service (IntServ) und Differentiated Service (DiffServ). IntServ beruht auf der Reservierung von Bandbreite für einzelne Ströme auf Ende-zu-Ende-Basis mittels RSVP (Reservation Protocol).

Die IntServ-Architektur hat jedoch erhebliche Nachteile:

- Bei großer Anzahl von Ende-zu-Ende-Verbindungen wächst der Prozeß-Overhead und Speicherbedarf der Router. Dies führt zu schlechter Skalierbarkeit.
- Alle Router müssen sehr gut ausgerüstet sein: z.B. mit RSVP, Scheduler, etc.

Der DiffServ-Ansatz löst diese Probleme. Er basiert auf der Zuordnung von Strömen zu bestimmten DiffServ-Klassen durch Setzen der Traffic Class in den Paketheadern. Die einzelnen Ströme werden an den Netzwerkübergängen abhängig von ihrer Service-Klasse behandelt. Damit kann eine wesentlich größere Anzahl von Strömen in einem Router verwaltet werden, was das erste Problem von IntServ löst. Außerdem benötigen die Router innerhalb der DiffServ-Domain weniger Funktionalität.

Wegen der oben angeführten Nachteile von IntServ wie schlechter Skalierbarkeit und zu grossem Umfang an Funktionen, die jeder Router unterstützen soll, wurde der DiffServ-Ansatz für QoS bevorzugt. DiffServ (DS)-Klassen können den Paketen durch das Setzen des Traffic Class Feldes zugeordnet werden. Aufgrund ihrer Traffic Class werden die Pakete in den Routern entsprechend behandelt.

Dabei handelt es sich um ein Markierungsschema, welches die Prioritäten den einzelnen DS-Klassen und damit den einzelnen Strömen zuordnet. Da DiffServ direkt auf dem IP-Protokoll basiert (Traffic Class), werden alle darüberliegenden Protokolle unterstützt - insbesondere UDP und TCP.

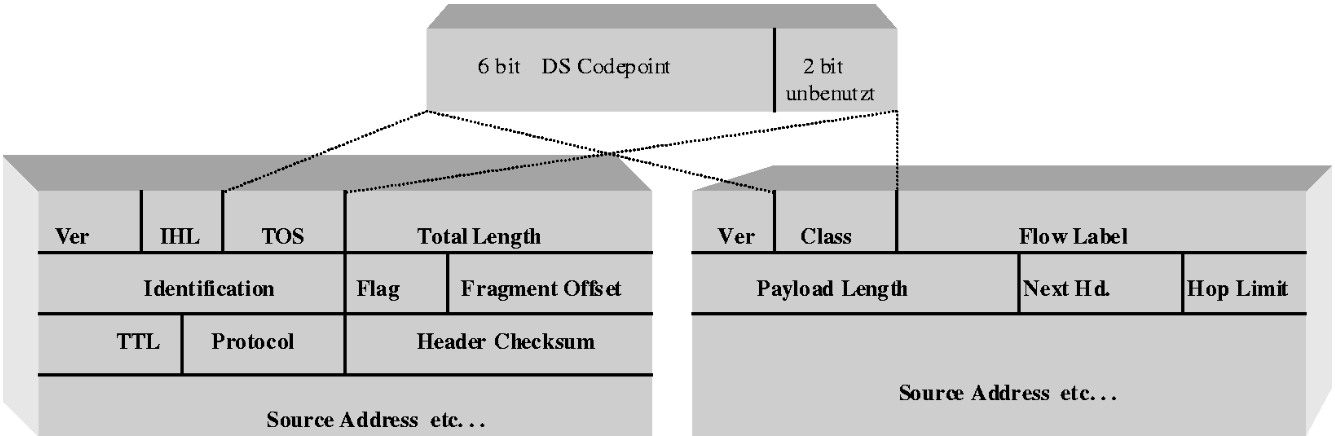


Abbildung 2.1: Traffic Class (DS-Feld) in IPv4- und IPv6-Paketen

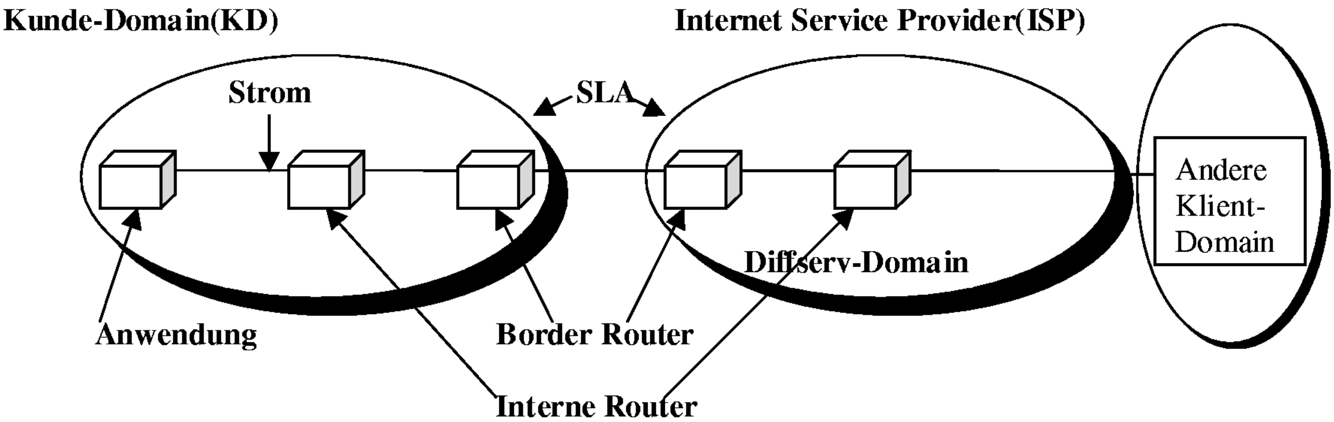


Abbildung 2.2: Mögliches Schema für DiffServ-System

2.2.2 Ein System für DiffServ

Damit der Kunde den DiffServ-Service überhaupt in Anspruch nehmen kann, muß er einen Vertrag mit dem Internet Service Provider (ISP) abschließen. Dieser Vertrag wird Service Level Agreement (SLA) genannt. Darin werden z.B. die Bandbreiten für die entsprechenden DS-Klassen vereinbart. Man unterscheidet:

Statische SLA: D.h. die Vereinbarung gilt über eine längere Zeit und der Kunde kann den Service jederzeit benutzten. Diese Art der SLA wird in DiffServ am häufigsten verwendet.

Dynamische SLA: D.h. die Bandbreite wird auf Anforderung mittels RSVP (reservation protocol) reserviert. Diese SLA stellt noch eine Herausforderung für DiffServ dar. In IntServ ist es üblich RSVP, und folglich die dynamische SLA zu verwenden.

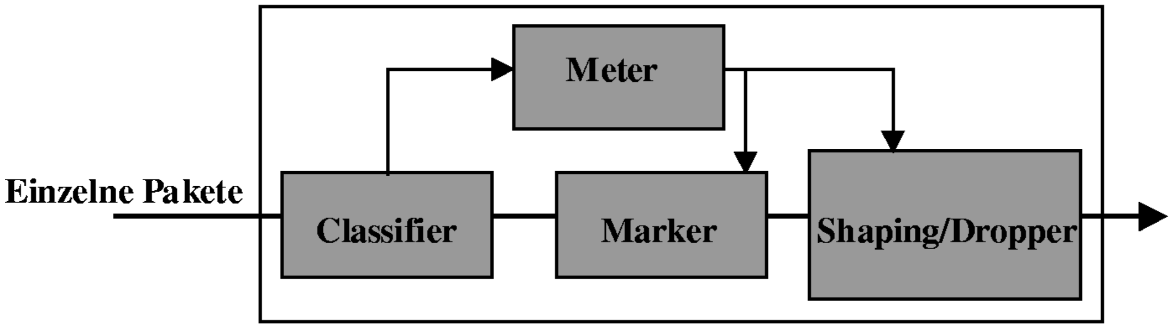


Abbildung 2.3: Die zu unterstützenden Aufgaben der am Netzübergang liegenden Router

2.2.3 Zu unterstützende Funktionalitäten

Man definiert für DiffServ die Funktionen, welche Router in der Kundendomain (KD) und Border Router sowie Anwendungen auf dem Weg der DS-Ströme unterstützen sollten:

- Markierung der Pakete. Dieses kann auch von einer Anwendung durchgeführt werden.
- Klassifizierung der Pakete bzgl. ihres Headers.
- Policing. Behandeln von OUT-Paketen. (Pakete, die nicht in die reservierte Bandbreite passen)
- Ummarkierung: kann als Teil des Policing-Prozesses aufgefaßt werden.

In der Abbildung 2.3 ist Funktionalität des Border-Routers zu sehen. Der Begriff *Meter* kennzeichnet dabei den Teil des Systems, welcher Pakete nach ihren Eigenschaften sortiert. Zum Beispiel kann der Meter bestimmen, ob ein Paket ein IN- oder OUT-Paket ist. Durch Shaping können die Pakete nach bestimmten Regeln verzögert werden, wenn sie ihre Rate übersteigen.

Die Router innerhalb der ISP-Domain sollen lediglich die Klassifizierung der Pakete nach der Traffic Class, d.h. nach DS-Klassen, ermöglichen. Dadurch werden Pakete von diesen Routern schnell weitergeleitet.

2.2.4 Die DiffServ-Services

Außer dem best-effort Service, der jetzt im Internet verwendet wird, können durch DiffServ noch die folgenden DS-Services unterstützt werden:

1. *Premium Service* mit den Eigenschaften *low-delay* & *low-jitter*. Dieser wird im weiteren als Expedited Forwarding (EF) bezeichnet.
2. *Assured Forwarding (AF)* sorgt für bessere Zuverlässigkeit im Vergleich zu best-effort.
3. *Olympic Service* mit *gold*, *silver*, *bronze* Services mit absteigender Qualität.

Die Verantwortung für die bereitzustellenden Services liegt beim ISP. Dadurch können aber Probleme entstehen: Wenn verschiedene ISP-Domains, welche auf dem Weg der Pakete liegen, verschiedene DS-Services unterstützen. Hierbei kann es z.B. vorkommen, daß ein Netz EF und BE unterstützt, während ein anderes den Olympic-Service implementiert. Dann ist eine Umsetzung erforderlich.

2.2.5 Eigenschaften und Implementierung der EF, AF und BE-Klassen

EF-Service

EF-Service wird als der beste (premium) Service bezeichnet und hat folgende Eigenschaften:

- kleine Verzögerungen und Jitter. Deshalb kann er für Internet-Telefonie und für Video-Konferenzen benutzt werden.
- EF-Pakete werden vor AF-Paketen gesendet (bedeutet eine größere Priorität von EF-Paketen).
- OUT-Pakete sollen verworfen werden, d.h. der Kunde soll dafür verantwortlich sein, die reservierte Bandbreite nicht zu überschreiten.
- EF kann statische oder dynamische SLAs verwenden.

Dadurch, dass EF seine Bandbreite nicht übersteigen darf, wird die Beeinträchtigung der anderen DiffServ-Klassen wie AF und BE vermieden.

Das Policing und der Sender selbst verhindern normalerweise die Überschreitung der Bandbreite. Es kann zur Überschreitung der SLA-Bandbreite in den ISP Domänen selbst kommen, wenn die Sender im ISP EF-Ströme verschicken. Diese Ströme werden nicht durch Policing-Maßnahmen überwacht, was zu obengenannten unerwünschten Effekten führt. Um dieses Problem zu lösen wird vom [IETF] das sogenannte Constrained-Based-Routing vorgeschlagen. Näheres darüber kann dem [InternetQoS] entnommen werden.

AF-Service

Der Assured Forwarding Service ist dafür da, den Benutzern einen zuverlässigen Service auch bei Netzüberlastungen bereitzustellen. Da man für AF normalerweise eine statische SLA verwendet, kann der Benutzer jederzeit den AF-Service in Anspruch nehmen.

Generell werden 4 verschiedene AF Klassen definiert. Sie unterscheiden sich in der relativen Verzögerung. Jede dieser Klassen kann jeweils 3 Unterklassen besitzen. Die Unterklassen unterscheiden sich in den Wahrscheinlichkeiten für die Verwerfung der Pakete. Insgesamt existiert also 12 Arten von AF-Service-Klassen.

Für die Implementierung von AF wird das sogenannte Random Early Detection (RED, siehe [DSFolien]) Queue Management oder einer seiner Nachfolger wie RIO oder GRIO verwendet. Durch RED mit seiner intelligenten Pufferverwaltung kann man in vielen Fällen verhindern, daß Router überflutet werden und alle sequentiell ankommenden Pakete verlorengehen.

Da der RED- Mechanismus z.B. auf DiffServ-Folien schon vorgestellt wurde, betrachten wir das RIO-Queue Management näher. RIO besitzt zwei Schranken für jede Queue (siehe Abbildung 2.4). Es verwendet im Grunde genommen zwei RED-Algorithmen. Dabei gelten folgende Regeln:

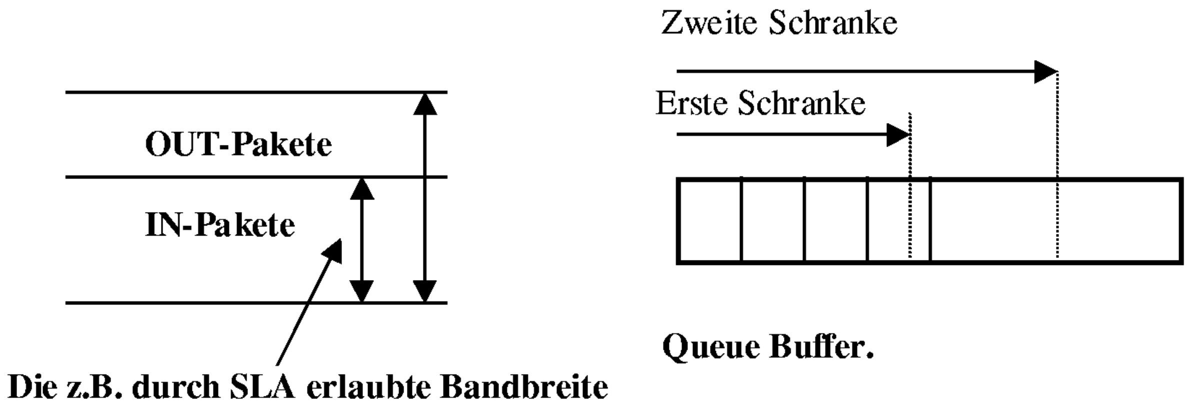


Abbildung 2.4: IN und OUT Pakete. RIO-Schranken

- Wenn die Queue-Größe unter der ersten Schranke liegt, dann wird nichts verworfen.
- Wenn die Queue-Größe zwischen zwei Schranken liegt, dann werden nur OUT-Pakete zufällig verworfen.
- Wenn die Queue-Größe über der zweiten Schranke liegt, werden sowohl OUT- als auch IN-Pakete verworfen. Die OUT-Pakete werden aber intensiver gedroppt.

Aufgrund dieser Eigenschaften wird verhindert, daß mehrere Ströme gleichzeitig ihre Datenraten erhöhen, wenn es die Bandbreiten-Kapazität erlaubt. Außerdem verhindert RIO die Beeinträchtigung anderer Ströme, da die OUT-Pakete stärker verworfen werden.

Da die Wahrscheinlichkeiten, daß IN-Pakete verworfen werden, gering sind, bekommt der Benutzer einen relativ zuverlässigen Service. Im Falle der relativ freien Bandbreite werden die OUT-Pakete nicht intensiv verworfen, und der Strom kann deshalb mehr Bandbreite nutzen. Daraus resultiert eine bessere Netzauslastung.

2.3 Quality of Service

von Simon Vey

2.3.1 Was ist "Quality of Service"?

Mit dem Begriff "Quality of Service" (QoS) werden ganz unterschiedliche Vorstellungen verbunden. Ziel ist es immer, eine möglichst hohe Dienstgüte zu erreichen. Was aber genau mit Dienstgüte gemeint ist, ist oft unklar. So geht z.B. aus der Anforderung, ein Video möglichst hochauflösend zu übertragen, noch nicht direkt hervor, welche Anforderungen damit an das Netz gestellt werden.

Daher ist es sinnvoll, die Parameter zu ermitteln, die bei einem speziellen Dienst für die Güte dieses Dienstes verantwortlich sind. Wichtige Parameter sind beispielsweise Datenrate, Ende-zu-Ende-Verzögerung, Verlustrate und Verfügbarkeit.
Die **Datenrate** beschreibt die Datenmenge, die in einer gewissen Zeit (normalerweise eine Sekunde) übertragen wird. Sie wird beispielsweise in Mbit/sec (1000000 Bit pro Sekunde) gemessen. Aber nicht alles, was beim Sender gesendet wird, muß beim Empfänger auch ankommen. Abhängig von der Verlustrate können sich diese beiden Datenraten durchaus unterscheiden.
Von der Datenrate (Throughput) zu unterscheiden ist der **Goodput**. Während die Datenrate nur von der Anzahl der übertragenen Bits oder Bytes abhängig ist, betrachtet der Goodput die Menge der sinnvoll übertragenen Daten. D.h., daß z.B. verfälschte oder unnötig wiederholt gesendete Pakete aus der Berechnung herausgenommen werden. Entscheidend für den Goodput sind nur die Pakete, die beim Empfänger wieder sinnvoll zu einem Datenstrom zusammengesetzt werden können.
Die **Ende-zu-Ende-Verzögerung** (End-to-End Delay) beschreibt die Zeit, die ein Paket vom Sender zum Empfänger braucht. Diese Zeit ist allerdings nicht ganz einfach zu ermitteln. Will man bei Sender und Empfänger jeweils die Sende- bzw Ankunftszeit messen, benötigt man auf den beiden Rechnern zwei exakt aufeinander abgestimmte Uhren, um eine sinnvolle Differenz der beiden Zeiten ermitteln zu können. Eine andere Möglichkeit, die Verzögerung zu ermitteln, ist das Messen der Round-Trip-Zeit, also der Zeit, die nötig ist, um ein Paket vom Sender zum Empfänger und sofort wieder zurück zu schicken. Dieser Wert ist einfacher zu ermitteln, da keine synchronisierten Uhren erforderlich sind. Aber die Round-Trip-Time (RTT) läßt nur eine Schätzung des Delay zu. Um das Delay zu berechnen, wird die RTT normalerweise durch zwei geteilt. Dabei geht man davon aus, daß Pakete von Sender zu Empfänger genausolange unterwegs sind wie Pakete in Rückrichtung. Dies ist aber problematisch, wenn die Pakete in den beiden Richtungen unterschiedliche Routen nehmen oder von den Routern auf der Strecke unterschiedlich behandelt werden. Die Übertragungszeiten können sich dann erheblich unterscheiden und die berechnete Ende-zu-Ende-Verzögerung stellt dann einen Mittelwert dieser beiden Zeiten dar.
Oft ist nicht nur die Ende-zu-Ende-Verzögerung sondern auch deren Schwankung entscheidend. Echtzeitanwendungen z.B. sind in der Wahl ihrer Empfangspuffer-Größe von dieser Schwankung abhängig: Je stärker das Delay variiert, desto größer muß der Puffer sein. Ein Maß für diese Schwankung ist der **Jitter**. Der Jitter ist die Varianz der Paket-Zwischenankunftszeiten beim Empfänger, die direkt von der Varianz des Delay abhängig ist.
Die **Verlustrate** beschreibt, wieviele Daten pro Sekunde verloren gehen. Unterschiedliche Anwendungen haben unterschiedliche Anforderungen an die Verlustrate. Eine TCP-Verbindung gleicht Verluste durch wiederholtes Senden der Pakete aus. Eine Echtzeitanwendung hat hierfür aber nicht unbedingt genügend Zeit. Es ist aber durchaus möglich, daß sie eine gewisse Verlustrate akzeptiert, was sich dann in einer Verminderung der Qualität bemerkbar machen kann.
Diese Merkmale der Dienstgüte müssen aber noch weiter differenziert werden. So macht es zum Beispiel einen Unterschied, ob man bei der Datenrate den maximalen, den minimalen (garantierten) oder den durchschnittlichen Wert betrachtet. Für eine Anwendung, die eine gewisse Mindestbandbreite braucht, ist der minimale Datendurchsatz natürlich wichtig, wohingegen bei FTP eher die durchnittliche Datenrate ins Gewicht fällt. Ähnliches gilt auch für Verlustrate und Ende-zu-Ende-Verzögerung.
Daß die **Verfügbarkeit** eine wesentliche Rolle spielt, ist offensichtlich. Ein ansonsten sehr guter

Dienst hat wenig Nutzen, wenn er fast nie in Anspruch genommen werden kann, weil der Server nicht aktiv oder überlastet ist.

2.3.2 Anforderungen an die Dienstgüte bei multimedialer Kommunikation

Multimediale Anwendungen stellen Anforderungen an das Netz, die sich von denen herkömmlicher Anwendungen, wie z.B. FTP, in einigen Punkten unterscheiden. Oft sind solche Anwendungen in der Lage, sich auf verändernde Bedingungen im Netz durch Variation der Signalqualität anzupassen. Da die Qualität aber nicht beliebig verringert werden soll, ist eine garantierte minimale Bandbreite erforderlich, um wenigstens in der minimalen Qualität senden zu können. Eventuell wird das Signal auch in Basisdaten und Erweiterungsdaten unterteilt. Die Basisdaten müssen dann auf jeden Fall den Empfänger erreichen, um das Signal überhaupt darstellen zu können, die Erweiterungsdaten verbessern dann die Signalqualität. Die Basisdaten müßten also verlustfrei übertragen werden, während bei den Erweiterungsdaten eventuell hohe Verluste akzeptiert werden. Wie oben bereits erwähnt, ist auch der Jitter wichtig. Bei einer gegebenen Puffer-Größe sollte der Jitter einen gewissen maximalen Wert nicht überschreiten, um einen Pufferüberlauf zu verhindern. Die Ende-zu-Ende-Verzögerung spielt beispielsweise bei Videokonferenzen eine große Rolle. Die Verzögerungen des Video- und Audiosignals sollten nicht zu deutlich wahrgenommen werden, um das Gefühl einer direkten Kommunikation zu erhalten.

2.3.3 Strombasierte und sessionbassierte Fairneß

Werden mehrere Ströme zu einer logischen Einheit, einer Session, zusammengefaßt, stellt sich die Frage, ob eine strombasierte Fairneß oder eine sessionbasierte Fairneß angestrebt werden soll. Bei der sessionbasierten Fairneß werden die Sessions als Einheiten aufgefaßt, unter denen die Bandbreite fair aufgeteilt werden soll. Das heißt, daß jede Session eine gewisse, eventuell von einer Gewichtung abhängige, Bandbreite zugeteilt bekommt, unabhängig von der Anzahl der in ihr enthaltenen Ströme. Sinnvoll könnte das sein, wenn eine Session mit einem Benutzer in Verbindung gebracht wird, der insgesamt für alle seine Ströme einen gewissen Preis bezahlt. Allerdings erhält man auf diese Weise keine absoluten Bandbreite-Garantien sondern nur eine relative Bevorzugung bzw. Benachteiligung gegenüber anderen Sessions. Strombasierte Fairneß bedeutet hingegen, daß die Bandbreite fair unter den Strömen aufgeteilt wird, unabhängig davon, zu welcher Session sie gehören.

2.4 Grundlagen analoger und digitaler Audiosignale

von Thomas Dreibholz

Bei Audiosignalen handelt es sich um Wellen, welche im Frequenzbereich von 20 Hz bis 20000 Hz hörbar[11] sind. Um diese im Computer verarbeiten zu können, müssen sie vom analogen Signal (Werte- und Zeitbereich sind kontinuierlich, siehe Abbildung 2.5) in digitale Werte (Werte-

[11] Einige Tiere, z.B. Hunde, können auch Frequenzen über 20000 Hz wahrnehmen.

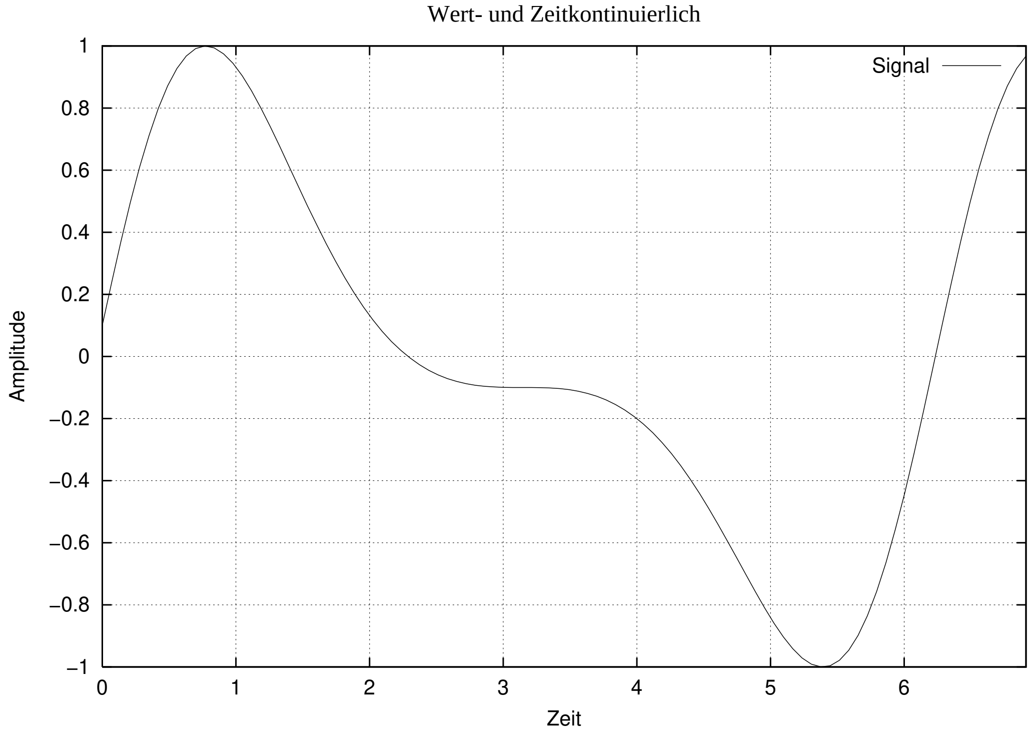


Abbildung 2.5: Ein Analoges Signal

und Zeitbereich sind diskret, siehe Abbildung 2.8) umgewandelt werden. Dies geschieht in zwei Schritten:

Abtastung (Sampling): In einem festgelegten Abstand ΔT (Sampling Rate ist $\frac{1}{\Delta T}$, z.B. $\frac{1}{44100}$ Sekunde) wird der Wert des Signales aufgezeichnet. Man erhält somit ein zeitdiskretes Signal, wobei die Werte (Samples) jedoch noch kontinuierlich sind (siehe Abbildung 2.6).

Quantisierung: Hierbei werden die Samples Wert-diskretisiert, indem sie z.B. auf 8-Bit-Werte (Bereich 0 bis 255) oder 16-Bit-Werte (Bereich 0 bis 65535) abgebildet werden (siehe Abbildung 2.8; Abbildung 2.7 zeigt zum Vergleich ein wertdiskretes, aber noch zeitkontinuierliches Signal).

Bei mehreren Kanälen (z.B. 2 bei Stereo) wird die Digitalisierung gleichzeitig auch für alle anderen Kanäle ausgeführt. Je höher die Abtastrate und je größer der Wertebereich bei der Quantisierung, desto besser ist die resultierende Qualität der digitalen Audioaufzeichnung. Einige typische Beispiele für Audioqualitäten sind in Tabelle 2.10 angegeben .

Das Abtasttheorem besagt, daß für eine fehlerfreie Rekonstruktion eines Signals mit Grenzfrequenz ν die Sampling Rate größer oder gleich $2*\nu$ sein muß. Mit 44100 Hz Abtastrate sind daher Frequenzen bis 22050 Hz möglich. 16 Bits ergeben mit 65536 Möglichkeiten zwar einen etwas geringeren Bereich als den, welchen das menschliche Ohr wahrnehmen kann (etwa 1 Millionen Möglichkeiten), in der Praxis ist der hierdurch entstehende Quantisierungsfehler jedoch vernachlässigbar.

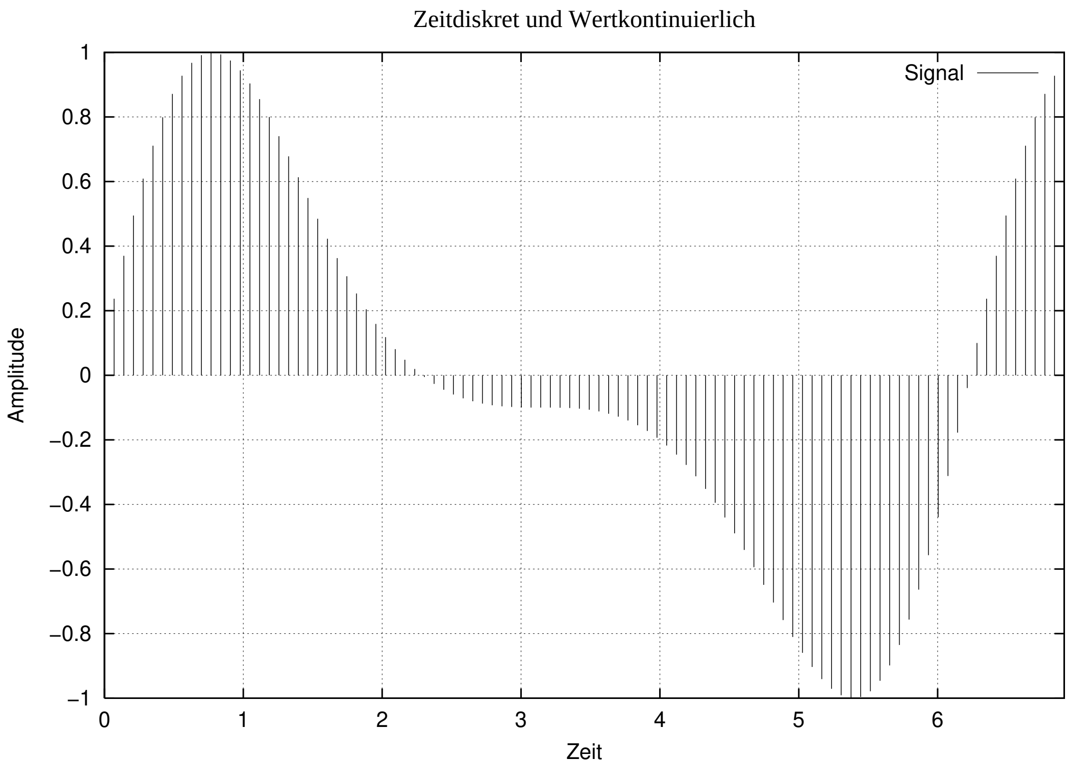


Abbildung 2.6: Zeitdiskretes Signal

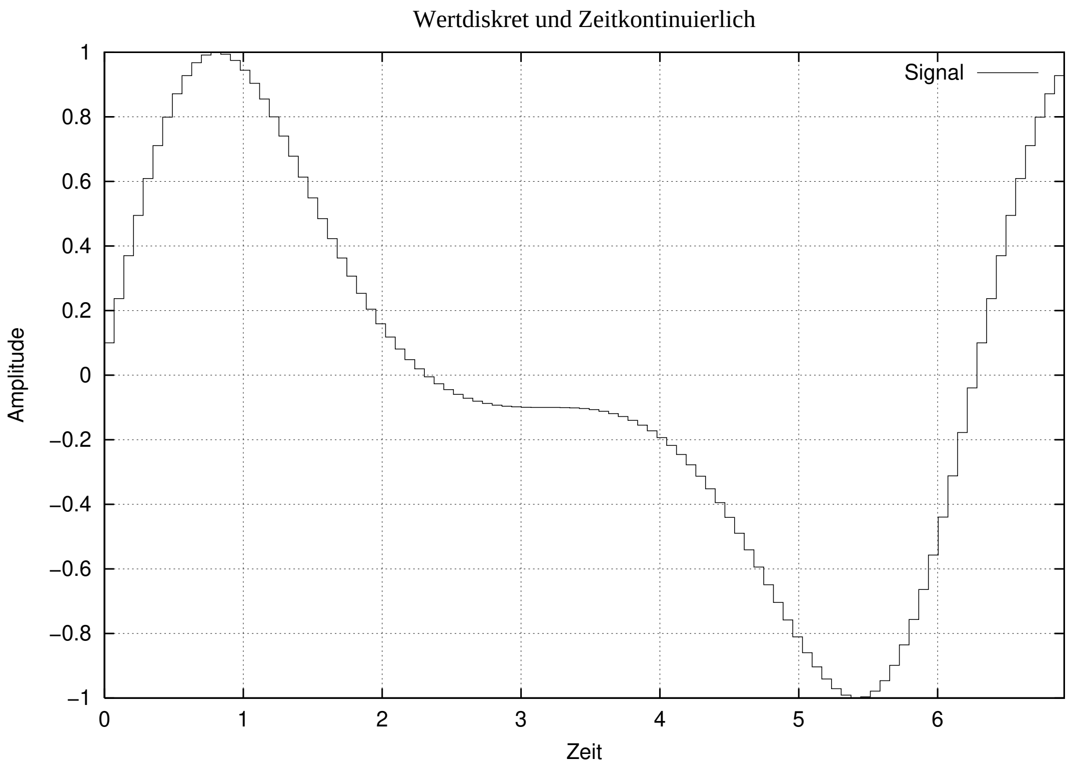


Abbildung 2.7: Wertdiskretes Signal

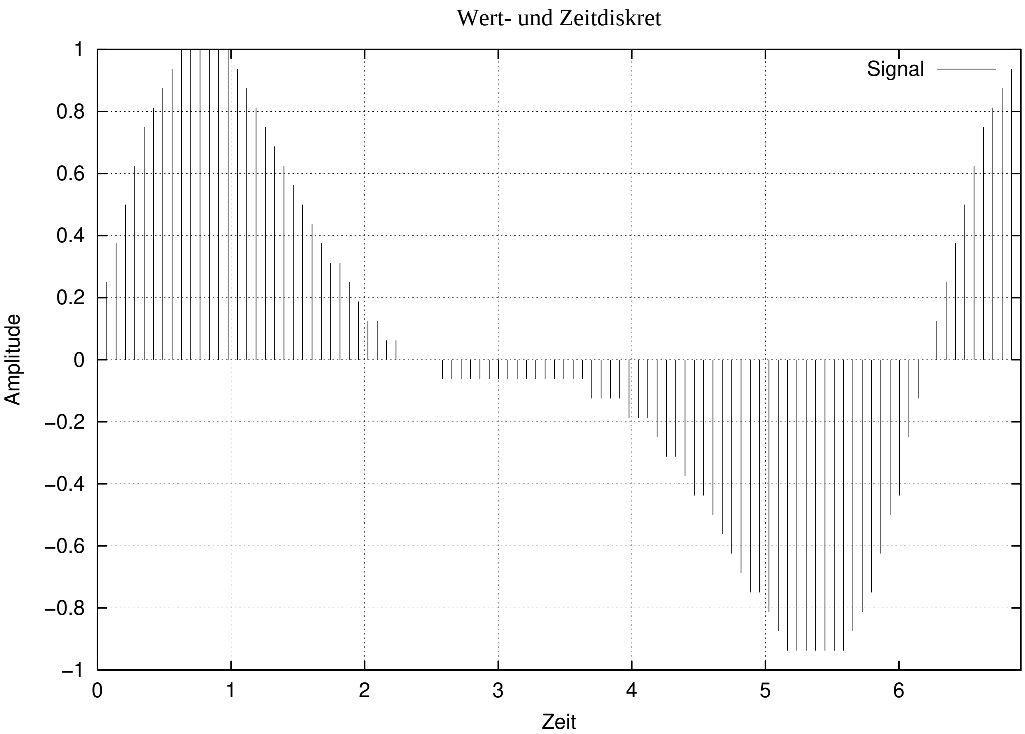


Abbildung 2.8: Ein Digitales Signal

Anwendung	Sampling Rate	Bits	Kanäle	Bits/Sekunde
Telefon (Amerika, Japan)	8000	7	1 (Mono)	56,000
Telefon (Europa)	8000	8	1 (Mono)	64,000
ältere PC-Soundkarte	22050	8	2 (Stereo)	352,800
CD	44100	16	2 (Stereo)	1,411,200

Tabelle 2.10: Typische Audioqualitäten

Kapitel 3

Die Systembeschreibung

In diesem Kapitel befindet sich die Beschreibung des implementierten Systems bestehend aus DiffServ-Router von Jan Selzer, dem RTP Audio-System und den Meßwerkzeugen von Thomas Dreibholz, sowie dem Quality of Service Management von Simon Vey. Die komplette Implementation ist als Download auf der [RTP Audio] Homepage zu finden.

3.1 Der DiffServ-Router

von Jan Selzer

3.1.1 Einleitung

Die letzten Linux-Kernel-Versionen 2.2.x bieten vielfältige Funktionen zur Unterstützung der sogenannten Traffic Control (Netzverkehrskontrolle). Mehrere Kernel-Teile und Benutzer Programme für Traffic Control, die von Alexey Kuznetsov ([LinuxTC]) implementiert sind, unterstützen IntServ. Für die Verwendung von DiffServ muß zusätzlich ein von Almesberger entwickeltes Patch im Kernel installiert werden. Dadurch werden die für DiffServ benötigten Teile in den Kernel eingefügt. Mit Hilfe des TC-Benutzerprogramm [IPRoute], welches Alexey Kuznetsov geschrieben wurde, kann ein Linux-Rechner als Router mit IntServ- und/oder DiffServ-Regeln konfiguriert werden. Das TC-Programm interagiert mit der Kernel durch den rtnetlink-Mechanismus des Linux-Kernels.

3.1.2 Kernel-DiffServ Architektur

Den DiffServ-Aufbau des Kernels kann man sich wie in Abbildung 3.2 vorstellen: Bei der Ankunft der Pakete wird in der "Input de-multiplexing" Komponente entschieden, ob sie an den Rechner selbst oder in das Netz weiterzuleiten sind. Die "Forwarding" Komponente bekommt die Pakete für das Weiterschicken entweder vom Rechner selbst ober vom Netz.

Nach dem Forwarding-Prozess werden die Daten einer Queue dem gewünschten Output-Interface zugeordnet. An dieser Stelle tritt die *Traffic Control* mit ihren Funktionen auf. Hier können die Pakete nach bestimmten Regeln z.B. verworfen oder mit gewissen Prioritäten gesendet werden.

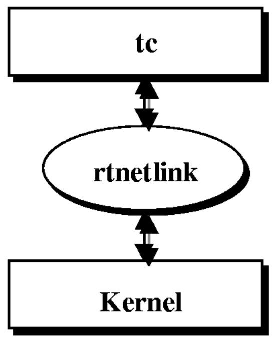


Abbildung 3.1: Kommunikation mit Kernel durch tc-Programm

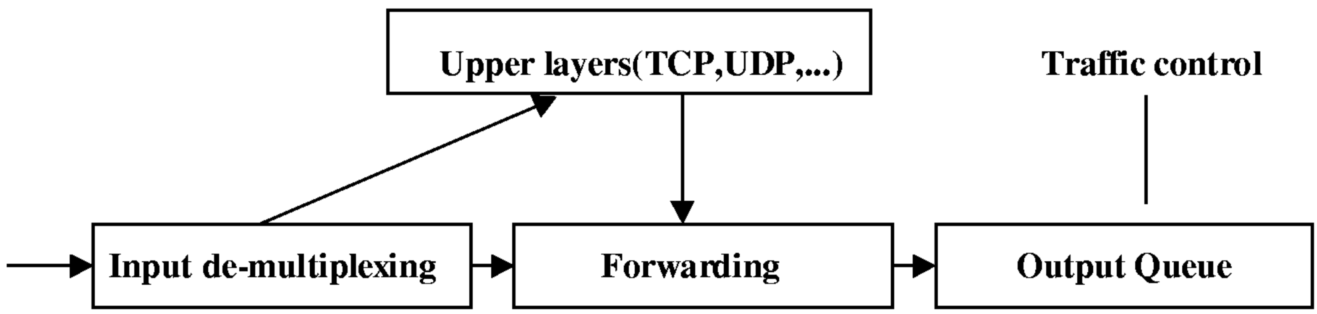


Abbildung 3.2: Kernel-Architektur

Abbildung 3.3: Einfache Queueing Discipline

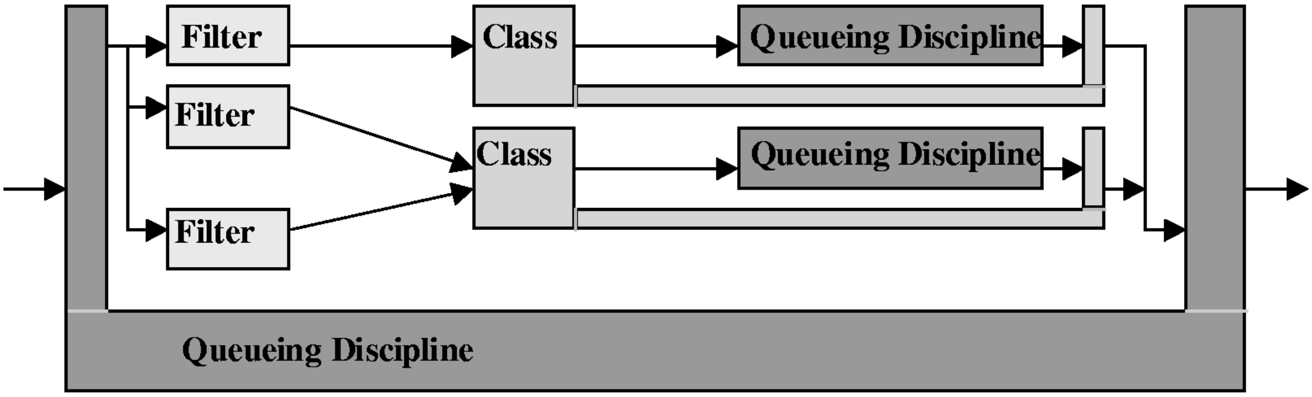


Abbildung 3.4: QD mit verschiedenen Klassen

Es ist auch möglich die Pakete zu verzögern. Nachdem ein Paket versandbereit ist, nimmt der Device-Driver es auf und leitet es in das Netz weiter.

3.1.3 DiffServ-Komponenten im Kernel

Der Traffic Control-Code im Kernel besteht aus den folgenden konzeptuellen Komponenten:

- *Queueing Discipline (QD)*
- *Classes*
- *Filters*
- *Policing*

Jedes Netzwerkdevice hat eine mit ihm assoziierte *Queueing Discipline.* Sie kontrolliert, wie ankommende und abgehende Pakete behandelt werden. Eine einfache *Queueing Discipline*, wie in Abbildung 3.3, kann die ankommenden Pakete in der Reihenfolge speichern, in der sie ankommen (FIFO). Das Weiterleiten geschieht sofort, wenn das Device frei wird.
Eine kompliziertere *Queueing Discipline* (QD) kann *Filters* für die Unterscheidung zwischen verschiedenen *Klassen* von Paketen verwenden. Dabei können die Klassen unterschiedlich z.B. durch die Vorgabe ihrer Prioritäten, behandelt werden. Abbildung 3.4 demonstriert einen möglichen Aufbau einer QD.

QD und *Classes* sind eng miteinander verbunden. Filters aber können beliebig mit QD und *Classes* kombiniert werden. Wie man auch aus Abbildung 3.4 sieht, speichert eine *Class* die Pakete

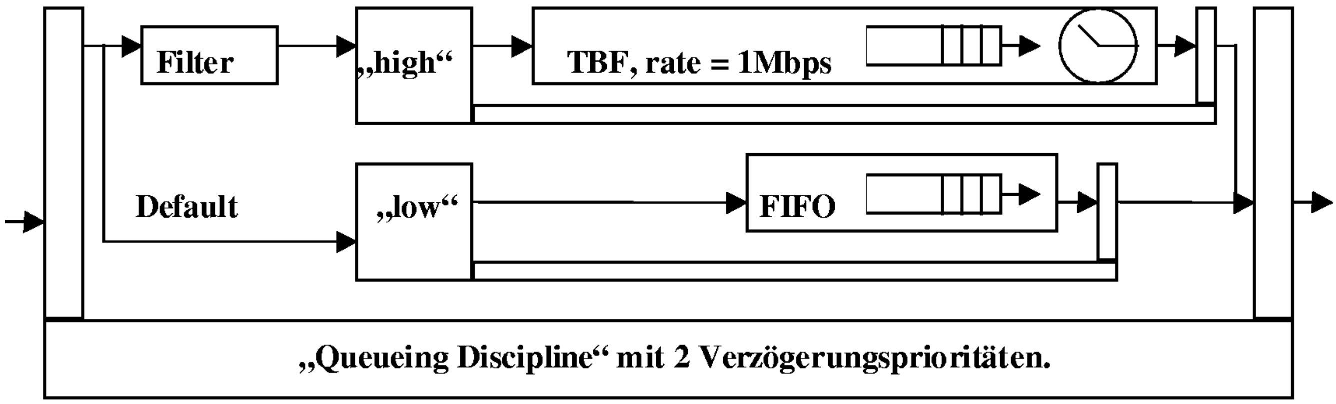


Abbildung 3.5: TBF und FIFO als QD

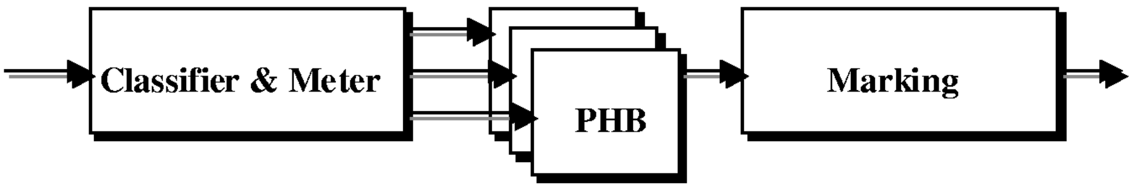


Abbildung 3.6: DiffServ-Struktur

nicht selbst, sondern mit Hilfe einer anderen Queue. Jede Queue kann weitere *Classes* haben. So wird ein rekursiver Aufbau von *Classes* und *Queues* ermöglicht. In Abbildung 3.5 wird ein weiteres Beispiel angeführt, wo man zwei *Classes* mit verschiedenen Prioritäten verwendet. Durch *Filters* werden die Pakete, die zu einer *Class* mit höherer Priorität gehören, gewählt und früher als die Default-Pakete weitergeschickt. Damit die Default-Pakete nicht zu stark verzögert werden, wird hier ein *Token Bucket Filter* (TBF) verwendet, der auch für das Shaping im TC-Programm benutzt werden kann.

Normalerweise hat eine *Class* ein QD. Es kann aber auch eine QD viele Klassen besitzen wie in Abbildung 3.4. Es ist auch möglich, daß viele Klassen eine QD besitzen. Der letzte Fall kann z.B. bei der Implementierung der AF-Klassen in einer Unterklasse vorkommen (siehe [IPRoute]). Hierbei besitzen verschiedene Unterklassen genau eine physische Queue, wie in Abschnitt 3.1.6 erklärt wird.

3.1.4 DiffServ-Erweiterungen im Linux-Kernel und Implementierung von Per Hop Behavior (PHB)

Die allgemeine Struktur des DiffServ Forwarding-Pfades sieht wie in Abbildung 3.6 aus. PHB bezeichnen die Eigenschaften und das Verhalten der Ströme, die bestimmten DiffServ-Klassen angehören. Dieses ist näher in den 2.2.4 Abschnitten und 2.2.5 erklärt. Zunächst werden die

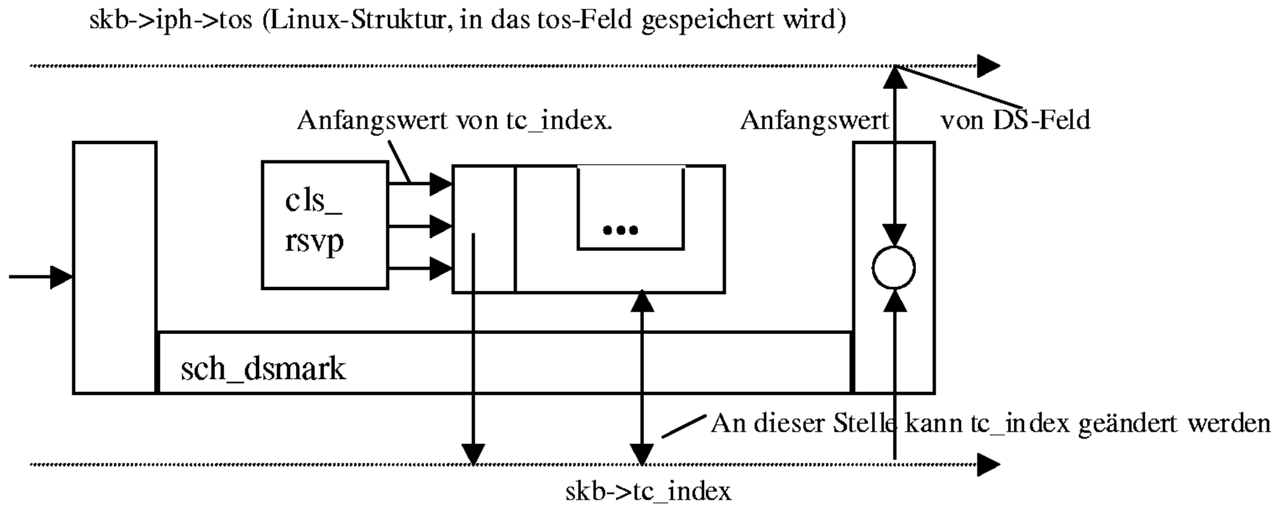


Abbildung 3.7: Micro-flow Classifier

Ströme nach ihren Eigenschaften mit Hilfe eines Meters klassifiziert, dann den einzelnen PHBs zugeordnet und am Ende ggf. markiert.

3.1.5 Die neuen DiffServ-Komponenten

In die Kernel-Struktur wurden drei neue Elementen eingefügt:

1. QD sch_dsmark. Sie dient dazu, das TOS-Feld zu lesen und zu beschreiben.

2. Classifier cls_tcindex. Er benutzt die Informationen aus sch_dsmark.

3. QD sch_gred. Sie unterstützt verschiedene Drop-Prioritäten und Bufferverteilungen.

Abbildung 3.7 zeigt die Verwendung von sch_dsmark, wenn die Pakete nur noch die DiffServ-Domain erreichen und folglich unmarkiert sind. Die Klassifizierung der Ströme und die Kontrolle der Senderate (metering) wird vom *Microflow-Classifier* (cls_rsvp) durchgeführt. Der *Microflow-Classifier* basiert auf der Klassifizierung von Strömen aufgrund von Quell- und Zieladresse, etc..

Dann wird aufgrund der Messung den Paketen eine DiffServ-Klasse zugewiesen. Ihre Bezeichnung wird in der Linux-Struktur skb->tc_index gespeichert. Dann können die inneren QD den Wert in dieser Struktur lesen und wenn nötig ändern. Am Ende von sch_dsmark QD wird die skb->tc_index Struktur gelesen und das TOS-Feld entsprechend gesetzt.

In Abbildung 3.8 funktioniert der Meter schon aufgrund des vorher gesetzten TOS-Feldes, d.h. mit Hilfe der cls_tcindex-Komponente. In der Literatur wird dies als behavior aggregate bezeichnet.

Die cls_tcindex und sch_dsmark Strukturen können durch das TC-Programm beeinflußt und konfiguriert werden. Mit Hilfe der von TC weitergegebenen Parameter werden die speziellen internen Tabellen erzeugt. In den Tabellen werden alle durch TC definierten Klassen und Regeln

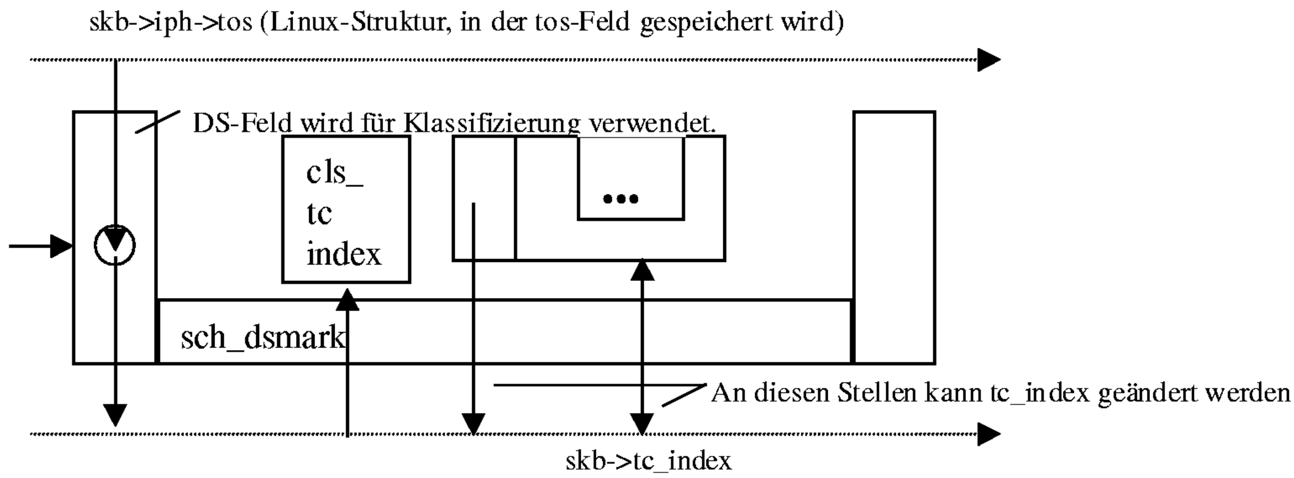


Abbildung 3.8: DS-Classifier

gespeichert. Mit Hilfe von speziellen Mechanismen können DiffServ-Kernel-Strukturen diese Daten verwenden.

3.1.6 sch_gred

Besonders interessant ist das neue Element sch_gred mit dessen Hilfe die AF-Sevice-Klasse implementiert wird. Die 4 AF-Klassen bekommen bei ihrer Implementierung bestimmte physische Bandbreiten zugeordnet. Dabei unterscheiden sich die Klassen nur in den Prioritäten. Je kleiner die Priorität, desto schneller wird das Paket abgeschickt. Dabei werden alle 3 Unterklassen einer AF-Klasse der einzigen physischen Queue von AF zugeordnet.

Wenn die benötigte AF-Klasse gefunden ist, verwendet sch_gred die skb->tc_index Struktur für die Wahl der richtigen *Virtual Queue* (VQ) aus der physikalischen Queue der jeweiligen AF-Klasse. Den virtuellen Queues innerhalb einer physikalischen Queue werden verschiedenen Wahrscheinlichkeiten für das Verwerfen der Pakete zugeordnet.

Für die Konfiguration des Kernels kann in TC ein Generalized RED (RIO) Management benutzt werden. (GRED (GRIO)) - Abbildung 3.9 zeigt das GRED-Prinzip.

Die Besonderheiten von GRED sind:

1. Bei GRED kann die Wahl der VQ von jedem *Meter* oder *Classifier*, die auf dem Weg eines Paketes liegen, beeinflußt werden.

2. Außerdem ist GRED in der Anzahl der VQs nicht beschränkt.

3. Für GRED gibt es keine vordefinierten Parameter. Die Prioritäten und Dropparameter kann der Benutzer selbst konfigurieren.

Zur Zeit sind im skb->tc_index 4 Bits für VQ verfügbar. Das bedeutet, daß man 16 verschiedene VQ definieren kann. GRED ist also ein allgemeiner Fall für RED, RIO und weitere Queue Management-Methoden.

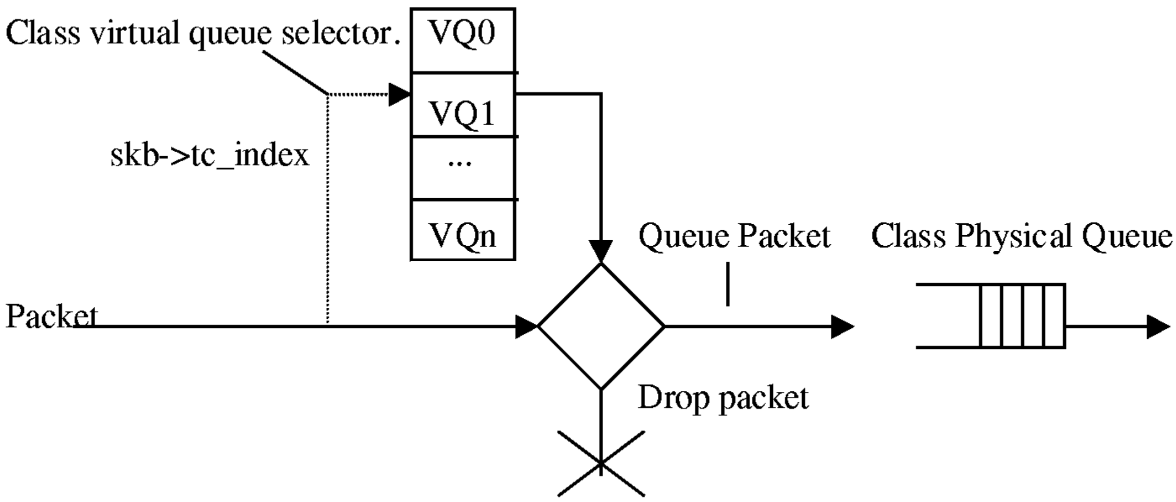


Abbildung 3.9: GRED und Verwendung von skb->tc_index

3.1.7 Kurzbeschreibung der Konfiguration und Verwendung des TC-Tools

Den allgemeinen Weg zur Konfiguration eines Szenarios kann man folgendermaßen beschreiben:

1. sch_dsmark wird an Output-Device angehängt.
2. Dann wird die von sch_dsmark verwendete QD definiert (z.B. Class Based Queue - CBQ).
3. Darauhin werden die zu verwendenden Klassen definiert und numeriert.
4. Jeder Klasse kann dann eine QD zugeordnet werden.

Durch TC können Filters definiert werden , die auf *"Routing Table", u32 Classifier, RSVP-Classifier*, etc basieren. Eingabe-Schema für TC-Tool sieht folgendermaßen aus:

```
tc [OPTION] OBJECT { COMMAND | help }
where OBJECT := { qdisc | class | filter }
      OPTION := { -s[statistics] | -d[details] | -r [raw] }
```

Auf die Erläuterung der weiteren Parameter sei auf [AdvNet] verwiesen.

3.1.8 System-Konfiguration und Beschreibung

Im System gibt es generell zwei Hauptrouten, durch die verschiedene Ströme gesendet werden. Zum einen versendet ein Media Server die Daten an zwei Clients, die in verschiedenen Domains liegen, zum anderen kann ein Störsender die Daten an einen Störempfänger schicken. Das Szenario wird in Abbildung 3.10 vorgestellt. Netz 1 und Netz 3 bilden die Kunden-Domains laut dem Modell aus den DiffServ-Grundlagen. Netz 2 bildet die DiffServ-Domain (z.B. Internet Service Provider), vgl. Abschnitt 2.2.2.

Das System, welches wir für die Testzwecke konfiguriert haben, besteht aus drei Netzen. Netz 1 enthält:

- Die Anwendung, die sich auf einem Server befindet. Diese besitzt das eingebaute QoS Management und die Möglichkeit, Pakete zu markieren. Die Menge der markierten Ströme, die einer bestimmten DiffServ-Klasse angehören, wird durch das SLA zwischen Netz 1 (Kundendomain) und Netz 2 (DS-Domain) definiert.
- Border Router, auf dem die Bandbreiten für die einzelnen DiffServ-Klassen mittels TC definiert sind. Dabei sind hier auch die Policing- und Ummarkierungsregeln installiert.

Netz 2 enthält:

- einen Klient, der den Service des Servers in Anspruch nimmt.
- den Core-Router, welcher die Ströme in die anderen Netze weiterleitet. Dabei achtet er auch darauf, daß die DiffServ-Ströme nicht die für sie allokierte Bandbreite überschreiten.
- Einen Störsender, der die Hindergrundströme für unsere Anwendung erzeugt.

Netz 3 enthält:

- einen Klient, welcher die Dienste des Servers in Anspruch nimmt.
- einen Störempfänger.

Beide Router werden mit 4 DS-Klassen konfiguriert: EF, AF1, AF2 und BE. Wir haben EF, AF2, AF1, BE DiffServ-Klassen im SLA definiert. Dabei hat EF die größte Priorität und andere Klassen bekommen ihre Prioritäten absteigend, so daß BE die niedrigste Priorität besitzt. DS-Klassen wurden mit folgenden Queue-Typen durch TC implementiert:

- EF durch eine FIFO-Queue,
- AF durch eine GRED-Queue,
- BE durch eine RED-Queue.

Die Aufteilung der gemeinsamen Bandbreite wurde mit Hilfe einer Class Based Queue organisiert. Die Router wurden folgendermaßen konfiguriert:

- Auf dem Border Router werden die Bandbreiten für alle DS-Klassen definiert. Also besteht die Aufgabe des Border-Routers darin, die Pakete, welche nicht in die allozierte Bandbreite passen, aus allen DS-Klassen zu verwerfen und damit die DS-Domain vor Überflutung zu schützen. Auf diesem Router können auch die Ummarkierungsregeln installiert werden.

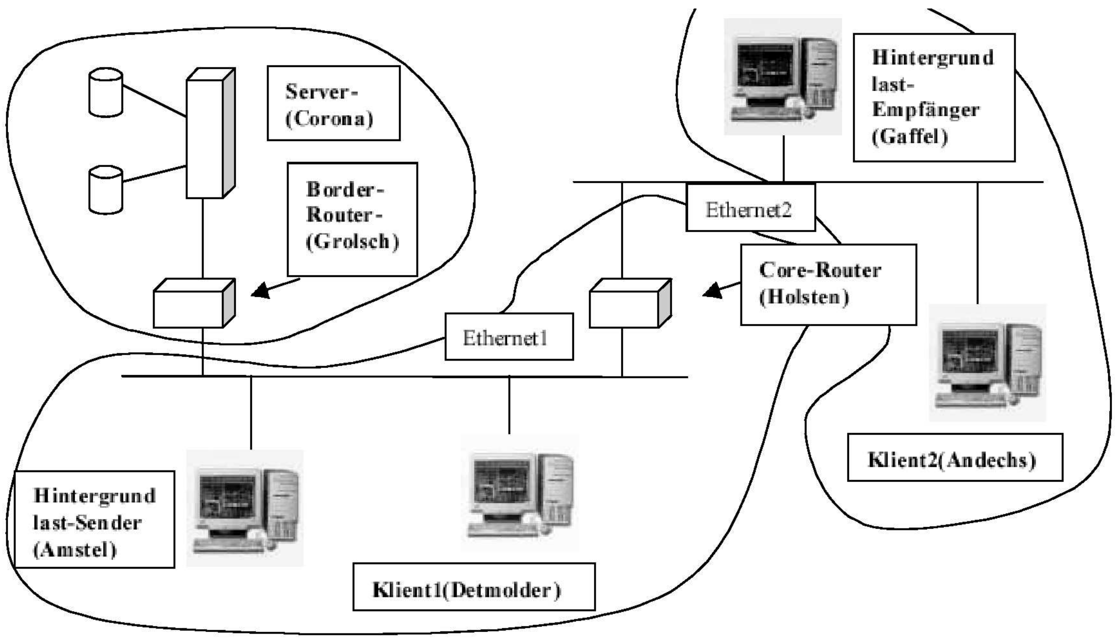


Abbildung 3.10: Das DiffServ-Szenario

- Auf dem Core-Router werden die Pakete nach dem TOS-Feld sortiert und entsprechend behandelt. Dabei werden hier auch die Grenzen der Bandbreiten für jede DS-Klasse gesetzt, um aus der DS-Domain kommende Pakete, welche die allozierte Bandbreite überschreiten, zu verwerfen. Der Core-Router unterstützt durch seine Funktionalität das richtige DiffServ-Verhalten. Das bedeutet z.B., daß EF-Pakete vor Paketen der AF- und BE-Klassen weitergeleitet werden.

Installation:
Um die Subnetzte aufzubauen wurden folgende Schritte durchgeführt:

1. Zusätzliche Ethernet-Karten wurden in den entsprechenden Rechnern installiert. Dadurch kann man die Subnetze definieren und die Messungen in ihnen relativ unabhängig von anderem Netzverkehr durchführen.

2. Die Routing-Tabellen wurden auf allen Rechnern mit Hilfe des *netconf*-Tools erweitert, um die für unser Szenario benötigten Regeln festzulegen. Die *netconf*-Tools befinden sich im Verzeichnis /sbin.

3. Die DiffServ-Funktionen wurden als SLA mittels TC installiert.

Folgende Probleme traten mit der TC-Konfiguration auf:

- Nach einem Reboot geht die Konfiguration verloren, da die Daten in Kernel-Strukturen gehalten werden.

- Bandbreite wird nicht genau alloziert, da diese nach inneren Tabellen des Kernels berechnet wird.

- TC funktioniert nur unter Linux.

3.1.9 Systembeschreibung

In diesem Kapitel werden die folgenden Systemschwerpunkte abgedeckt:

- QoS-Management

- Congestion-Management

- Ansätze für die Anbindung vom Anwendungsserver an das DiffServ-Netz.

Das System stellt einen Multimedia-Server dar, der die in einer Multimedia-Bilbliothek gespeicherten Daten für Clients bereitstellt und den Zugriff auf sie kontrolliert. Die Multimedia-Ströme können weiter in Substreams, d.h. Ströme mit Hauptdaten und Ströme mit Erweiterungsdaten, zerlegt werden. Diese Ströme können durch verschiedene DiffServ-Kanäle transportiert werden. Abbildung 3.11 zeigt das eigentliche QoS- und Congestion-Management System des CORAL-Projektes.

Das System kann an einen DiffServ-Router angebunden sein. Das Kodieren der Datenströme wird in der Server-Anwendung implementiert. Im System wird das Real-Time Transport Protocol (RTP) für den Ende-zu-Ende Transport benutzt. Das Management-System vom Server besteht aus zwei wichtigen Elementen: dem intra-stream und dem inter-stream Management.

Das Layering Control Modul (LCM) bildet den Hauptteil des Intra-Streams-Managements. Es verbindet die RTP-Ströme mit den Stream-Informationen. Um verschiedene Streams zu synchronisieren und zu transportieren, verwendet das LCM eine QoS-Beschreibung, welche es vom Server übergeben bekommt. Zum einen kontrolliert das LCM den Skalierungsprozess, zum anderen teilt es damit verbundene Informationen der Anwendung mit. Es ist anzumerken, daß jeder Stream mit einer DiffServ-Klasse assoziiert ist.

Das QoS-Management Modul ist ein zentrales Element des Systems. Seine Aufgabe ist die Zuordnung der verfügbaren Ressourcen des Systems an die Anwendungsströme. Wenn es doch zur Überschreitung der Ressourcen kommen sollte, dann stößt es die benötigte Skalierung an. Dabei kann man die QoS-Informationen in zwei Teile zerlegen:

- statische Infos: Beschreibungsdaten für spezielle Stromtypen.

- dynamische Infos: z.B. Infos über Gewichte einzelner Ströme.

Die Daten über die verfügbaren Ressourcen für die einzelnen DiffServ-Klassen sind im RESV-Modul gespeichert. Diese Infos bilden z.B. die reale Verteilung der Bandbreiten zwischen DiffServ-Klassen ab. Diese Verteilung wird durch ein SLA zwischen der Anwendungsdomain und der DiffServ-Domain (Internet Service Provider) festgelegt. Die SLA-Daten werden auf dem Border Router installiert.

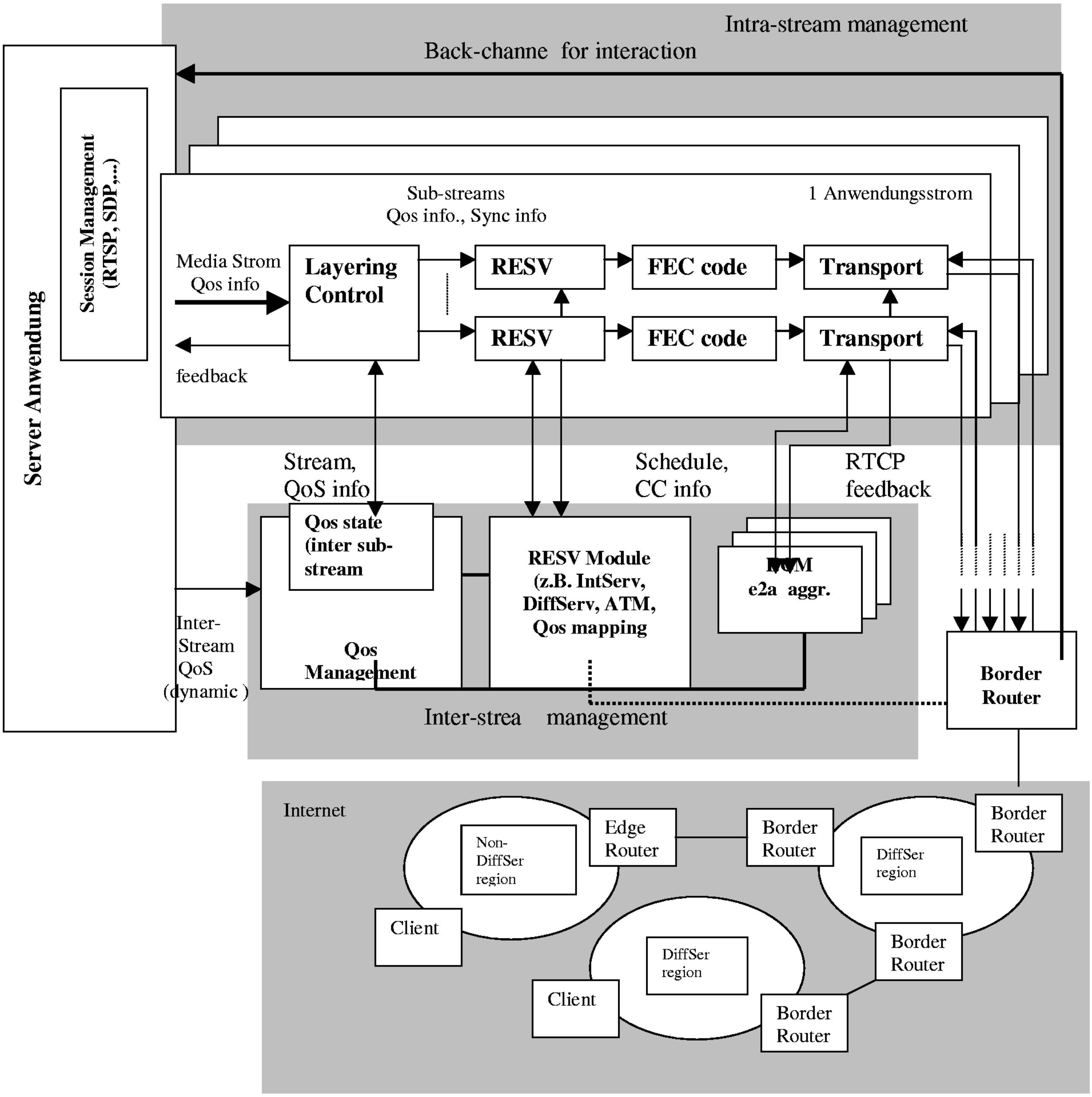


Abbildung 3.11: CORAL Konzept

Die zweite große Komponente des Systems ist das Inter-Stream Endpoint Congestion Management (ECM). Das ECM kontrolliert Streams auf Ende-zu-Ende Basis. Alle RTCP-Empfänger-Reporte werden von ECM abgefangen und abgearbeitet. Dabei werden die Packet Loss Rate Informationen an das QoS-Management geliefert. Außerdem kann das ECM mit Hilfe der Congestion Control Informationen die Größe der Pakete und die Rate der Streams einstellen.

Zu erwähnen ist, daß das statische SLA durch ein dynamisches SLA ersetzt werden kann. Dafür muß die Anwendung u.a. als Bandwidth Broker fungieren. Dadurch könnte dann die Anwendung abhängig von der Situation im Netz die Ressourcen der DiffServ-Klassen umverteilen und daraus Nutzen ziehen.

Von dem Clients werden die folgenden Funktionalitäten erwartet:

1. RTP- und RTCP-Unterstützung.
2. Bei der Verwendung von RSVP soll der Client RSVP-fähig sein. (Bei der Verwendung von DiffServ muß der Client keine zusätzliche Funktionalität zu haben.)
3. Der Client kann durch die Mitteilung von Infos über seine Ressourcen dem QoS-Management helfen.
4. Alle anderen Funktionen sind anwendungsabhängig.

3.2 Das RTP Audio System

von Thomas Dreibholz

Dieses Unterkapitel gibt eine Beschreibung der Implementation von RTP Audio. Dies ist ein System zum Echtzeit-Transport von Audiodaten über ein Netzwerk unter Benutzung des RTP-Protokolls. Im ersten Teil werden die grundlegenden Funktionalitäten wie Socket- und Thread-Implementation beschrieben, auf welchen das System aufbaut. Anschließend wird auf die Realisierung der einzelnen Programmteile eingegangen: Der RTP/RTCP-Transport, der RTP Audio Server und die RTP Audio Clients.
Die Benutzerdokumentation der Programme befindet sich in Anhang A, UML-Diagramme im Anhang B. Die komplette Dokumentation sämtlicher Klassen sowie das RTP Audio Paket zum Download sind auf der [RTP Audio] Homepage zu finden.

3.2.1 Grundlegende Funktionalitäten und Klassen

Grundlegende Anforderungen an das RTP Audio-System waren Erweiterbarkeit, Wiederverwendbarkeit, Portabilität und die Unterstützung aktueller und zukünftiger Entwicklungen im Hard- und Softwarebereich wie z.B. 64-Bit-Prozessoren, Parallelrechner, IPv6 und Flowlabels. Um dies alles zu erreichen, wurden für die Implementation des Systems folgende elementare Design-Entscheidungen getroffen:

- Betriebssystem: Linux, aufgrund von Verfügbarkeit, Leistungsfähigkeit und Stabilität sowie dem Vorhandensein des Quellcodes von Kernel und Programmen.

- Programmiersprache: C++ mit GNU C/C++ als Compiler, wegen Mächtigkeit und Performanz.
- Verwendung des Werkzeuges *kdoc 2.022* für die Dokumentation der Klassen (siehe [RTP Audio]).
- Verwendung von Threads zur Trennung unterschiedlicher Workflows und Unterstützung von Parallelität.
- Unicast-Übertragung: Für den Benutzer sollte es möglich sein, seine Position innerhalb des Audio-Mediums beliebig zu ändern, einen Pause-Modus einzuschalten, die Qualität zu ändern, ... Eine Multicast-Übertragung an mehrere Benutzer gleichzeitig war deshalb ausgeschlossen. Stattdessen werden beliebig viele Clients an einem Server unterstützt, wobei jedoch jeder Client einzeln bedient wird. Deren Anzahl ist nur durch die Ressourcen CPU-Zeit, Hauptspeicher und Sockets beschränkt ist.

Begonnen am 26. November 1999, besteht das RTP AUDIO-System mittlerweile aus mehr als 46500 Zeilen Code, welcher sich auf etwa 100 Klassen verteilt.

Systemkonfigurations-Einstellungen

Die zentrale Datei für die Systemkonfiguration der Implementations-Plattform ist *system.h*. Hier werden unter anderem Compiler-Variablen für die Anzahl der CPU-Bits (32 für Intel x86) und die Byte-Order (z.B. Little Endian für Intel x86, Big Endian für Motorola 680x0) gesetzt. Zudem werden per Typedef die Datentypen *card8*, *card16*, *card32* und *card64* für 8- bis 64-Bit vorzeichenlose Zahlen; *int8*, *int16*, *int32* und *int64* für 8- bis 64-Bit Zahlen mit Vorzeichen sowie *cardinal* und *integer* für die CPU-spezifische Default-Bitlänge (mindestens jedoch 32 Bit) definiert. Durch die konsequente Verwendung dieser Datentypen und Definitionen für alle weiteren Programmteile wird eine problemlose Portierbarkeit auf neue Hardware gewährleistet.

Die Socket-Klassen

Die *Socket*-Klasse stellt die Verbindung zu den Socket-Funktionen des Betriebssystems dar. Zur Verwaltung von Socket-Adressen gibt es das Interface *SocketAddress*, von welchem die Klassen für die Adreßtypen der verschiedenen Netzwerk-Protokolle (IPv4/IPv6 und Unix-Sockets) abgeleitet sind: *InternetAddress* und *UnixAddress*. Klassendiagramme für diese und alle weiteren Klassen sind im Anhang B zu finden, die Dokumentationen auf der Homepage.

Ein Beispiel für die Verwendung der *Socket*-Klasse:

```
InternetAddress address("odin:1234");
Socket socket(Socket::IP,Socket::Datagram,Socket::Default);
if(socket.connect(address,0x2e)) {
   socket.send("TEST",4,0,0x0a);
}
```

Hier werden ein *InternetAddress*-Objekt mit der IP-Adresse des Hosts *odin* sowie UDP-Port-Nummer 1234 und ein UDP/IP-Socket erzeugt. *InternetAddress* unterstützt dabei auch geklammerte Adreßangaben gemäß [RFC 2732] vom Dezember 1999 wie z.B. "`[1234::200:100]:7500`" -

dies ist insbesondere bei IPv6-Adressen wesentlich übersichtlicher. Das Socket wird ein IPv6-Socket, falls der Rechner dies unterstützt. Die IP-Adresse wird intern grundsätzlich immer im IPv6-Format gespeichert und gegebenenfalls für den *connect()*-Aufruf konvertiert (IPv4-mapped IPv6 $\leftrightarrow$ IPv4, falls das Socket nur ein IPv4-Socket ist). Der Parameter 0x2e bei *connect()* gibt den Default Traffic-Class-Wert für den UDP-Versand an (hier: EF). Für den *send()*-Aufruf wird der Traffic Class-Wert 0x0a (AF11) benutzt.

Ebenfalls in die *Socket*-Klasse integriert ist die Unterstützung von Flowlabels bei IPv6. Unter Linux sind Flowlabels mittels des Flowlabel Managers im IPv6-Modul des Kernels realisiert, welcher für die Verteilung und Verwaltung der Flowlabels zuständig ist. Ein Programm, welches ein Flowlabel verwenden möchte, muß dieses zuerst mit der gewünschten Ziel-Adresse beim Flowlabel Manager anfordern. War die Anforderung erfolgreich, kann nun über dieses Flowlabel zu der vorhin angegebenen Ziel-IPv6-Adresse (verschiedene Ports auf Transportebene (z.B. UDP oder TCP) sind natürlich möglich) gesendet werden. Nach erfolgter Benutzung ist das Flowlabel wieder über den Manager freizugeben.

Linux kennt für Flowlabels vier verschiedene Sharelevels, welche es ermöglichen, ein Flowlabel mit mehreren Sockets gleichzeitig zu benutzen[1]: Exclusive, Process, User und Any. Je nach Stufe ist die Verwendung auf das Socket, den Prozeß, den Benutzer oder gar nicht beschränkt. So ist es dann z. B. möglich, daß mehrere Sender eine einzige, mittels RSVP reservierte Strecke zum Zielrechner nutzen.

Ein reserviertes Flowlabel muß in regelmäßigen Abständen beim Manager erneuert werden (renew). Dabei gibt es zwei Parameter: *linger* und *expires*. Ersterer ist die Zeit in Sekunden, die nach Freigabe des Labels gewartet wird, bevor es neu vergeben werden kann (Default: 6 Sekunden). Der *expires*-Parameter gibt die Zeit in Sekunden an, die das Flowlabel noch gültig ist. Da Linux Flowlabels nicht persistent speichert, kann es bei zu hohem expires-Parameter vorkommen, daß nach einem Reboot das Flowlabel für eine andere Ziel-Adresse neu vergeben wird und somit im Netzwerk keine Eindeutigkeit von Flowlabel + Quelladresse mehr vorliegt! Ein sinnvoller Wert ist daher z.B. 10 Sekunden.

Ein Beispiel für die Verwendung von Flowlabels mit der *Socket*-Klasse:

```
Socket socket(Socket::IPv6,Socket::Datagram,Socket::Default);
InternetAddress address("ipv6-odin:1234");
InternetFlow flow = socket.allocFlow(address);
if(flow.getFlowLabel() != 0x00000) {
   if(socket.connect(flow,0x2e)) {
      socket.send("TEST",4);
   }
   socket.freeFlow(flow);
}
```

Es wird ein Flowlabel im Kernel reserviert, die Verbindung mit Traffic-Class 0x2e (EF) hergestellt und ein Test-String gesendet.

Für die komplette Dokumentation der Klassen sei auf die [RTP Audio] Homepage verwiesen.

[1] Natürlich muß auch hier für alle Sockets die gleiche Ziel-IPv6-Adresse verwendet werden. Nur die Ports dürfen sich unterscheiden.

Die Thread-Klassen

Die Thread-Klassen *Synchronizable* und *Thread* sind Wrapper-Klassen für die C-Funktionen der libpthread-Bibliothek. Ihre Aufgabe ist es, die Thread-Funktionen für die objektorientierte Verwendung nach einem Java-ähnlichen Muster bereitzustellen und ihre Anwendung zu vereinfachen.

Die Klasse *Synchronizable* enthält Funktionen für die Thread-Synchronisation durch ein PThread-Mutex, welches eine exklusive Sperre darstellt, welche zur gleichen Zeit nur von einem einzigen Thread gehalten werden kann. Dieses wird in den Methoden *synchronized()* und *unsynchronized()* realisiert. Für gemeinsam benutzte Programmteile muß dann nur noch sichergestellt werden, daß alle Zugriffe in *synchronized() / unsynchronized()*-Blöcken durchgeführt werden.

In der abstrakten *Thread*-Klasse, welche von *Synchonizable* abgeleitet ist, wird die eigentliche Thread-Funktionalität implementiert. Mittels *start()* und *stop()*-Methoden kann der Thread gestartet und beendet werden. Das eigentliche Programm des Threads liegt in der virtuellen Methode *run()*, welche von Unterklassen zu implementieren ist.

Beispiel zur Verwendung von *Synchronizable* und *Thread*:

```
class TestThread :  public class Thread
{
   ...
   void run();
   void test();
};
void TestThread::run()
{
   ...  // Thread-Programm
   synchronized();
   ...  // Synchronisierter Zugriff auf die Klasse
   unsynchronized();
   ...
}
void TestThread::test()
{
   synchronized();
   ...  // Synchronisierter Zugriff auf die Klasse
   unsynchronized();
}
```

Thread-Funktionen, welche in regelmäßigen Abständen aufgerufen werden sollen, sind mit der abstrakten Klasse *TimedThread* implementiert, welche von *Thread* abgeleitet ist. In dieser ist die virtuelle Methode *timerEvent()* vom Benutzer zu implementieren. Diese wird anschließend in durch die Methode *setInterval()* gesetzten Abständen ausgeführt.

Aufgrund der Ungenauigkeit des Scheduling-Timers von einigen Millisekunden[2] war es notwendig, eine optionale Korrektur zu implementieren: Nach jedem Aufruf von *timerEvent()* werden die

[2] Die Ungenauigkeit hängt stark von der Hardware ab, z.B. ca. 5 bis 10 ms auf Intel x86-Systemen oder etwa 2 ms auf Alpha-Systemen.

TransportInfoLayer	
card64 BytesPerSecond	Bytes/Sekunde in diesem Layer
card32 PacketsPerSecond	Pakete/Sekunde in diesem Layer
card32 FrameSize	Frame-Größe in diesem Layer
card8 MaxLossRate	Maximaler akzept. Verlust in $\frac{1}{255}$

Tabelle 3.1: Die *TransportInfoLayer*-Klasse

Soll- und Ist-Anzahl der Aufrufe seit dem Start verglichen. Ist die Abweichung kleiner als eine Konstante, so wird *timerEvent*() sooft direkt hintereinander aufgerufen, bis Ist- und Sollanzahl wieder übereinstimmen. Andernfalls - bei zu großer Abweichung - wird keine Korrektur mehr durchgeführt, um das System nicht zu überlasten.

Für das Debugging - insbesondere von Multithread-Programmen - hat sich die C-Bibliothek *libefence* als sehr nützlich erwiesen. Diese stellt Funktionen zur Verfügung, welche illegale Speicherzugriffe sehr einfach aufdecken können. Die Lokalisierung von Fehlern wird somit erheblich erleichtert, denn Multithread-Programme lassen sich nicht mit einem Debugger verarbeiten (zur Begründung siehe das [ThreadFAQ] der glibc 2.1).

3.2.2 Die RTP/RTCP-Implementation

Die RTP und RTCP-Klassen (siehe Abbildung 3.12) sind die Implementation des RTP-Protokoll aus [RFC 1889]. Zudem wird die Aufteilung der Daten in mehrere Layer realisiert, welche jeweils über verschiedene Traffic Classes und/oder Flowlabels gesendet werden können und somit beim Transport über das Netzwerk mit unterschiedlicher Dienstgüte behandelt werden, bzw. verschiedene reservierte Strecken benutzen.

Die RTP/RTCP Implementation besteht aus drei Teilen:

1. QoS-Beschreibung durch die Klassen *TransportInfo* und *ExtendedTransportInfo*,
2. der Kodierung in Form von Implementationen der Interfaces *EncoderInterface* und *DecoderInterface* sowie
3. den eigentlichen RTP/RTCP-Klassen.

Die Klassen sind dabei allgemein gehalten, so daß eine Wiederverwendung, z.B. für den Transport von Videos, problemlos möglich ist.

Die Klassen *TransportInfo* und *ExtendedTransportInfo*

Diese Klassen enthalten die komplette QoS-Beschreibung einer Kodierung sowie die aktuellen und gewünschten Einstellungen. Sie sind von zentraler Bedeutung für das gesamte QoS-Management.

TransportInfoLayer (siehe Tabelle 3.1) und *TransportInfoLevel* (siehe Tabelle 3.2) enthalten dabei die QoS-Beschreibungen eines Layers bzw. eines Levels in Form von Framegrößen, Byte[3]-

[3] Die Byteraten-Werte geben hier jeweils die Bytes pro Sekunde von Payload + kompletten Headern an.

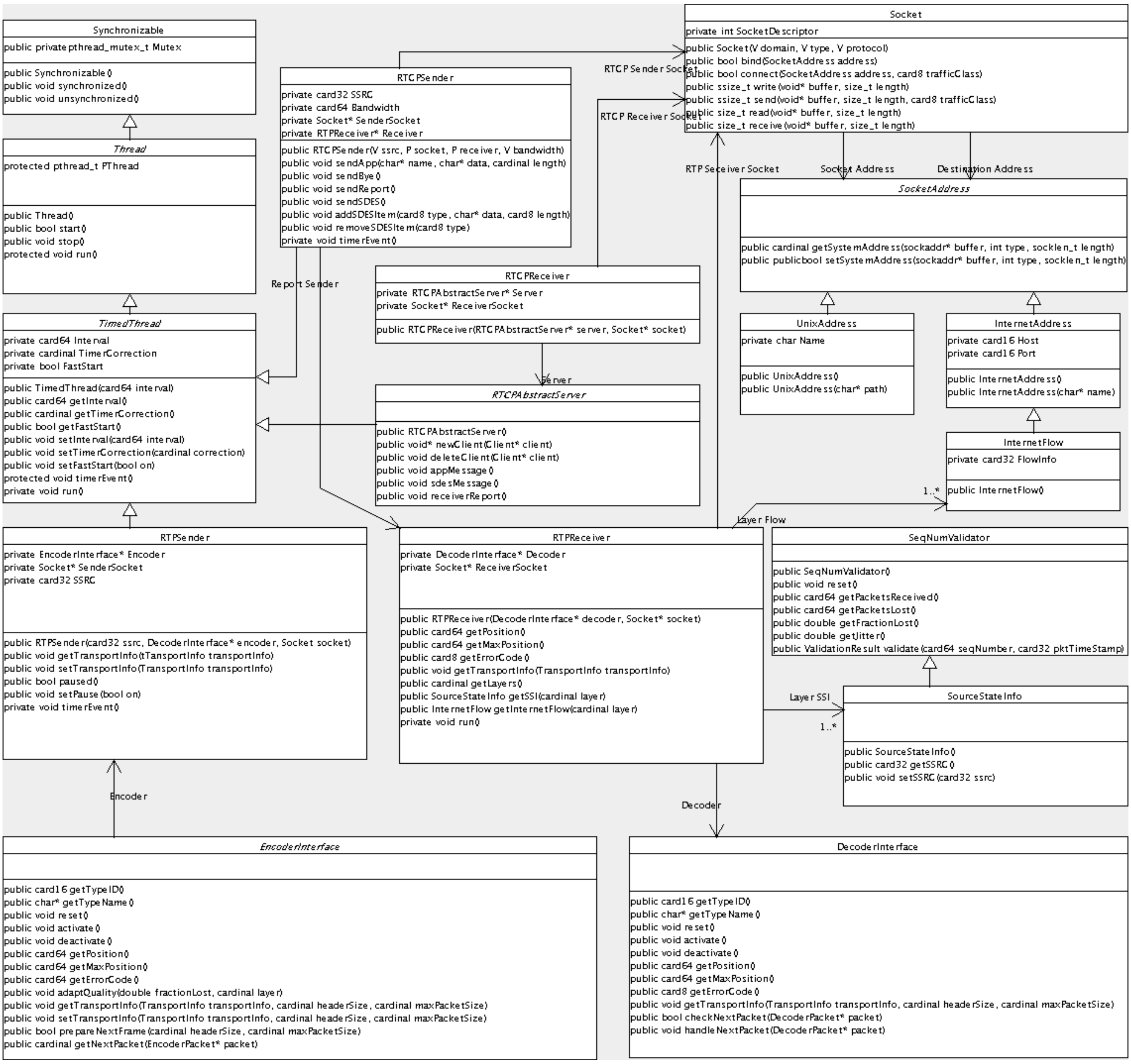


Abbildung 3.12: UML Diagramm zu den RTP/RTCP-Klassen.

TransportInfoLevel	
card32 BytesPerSecondScale	Skalierungswert * 65535 für Bytes/Sek.
card32 PacketsPerSecondScale	Skalierungswert * 65535 für Pakete/Sek.
card32 FramesPerSecondScale	Skalierungswert * 65535 für Frames/Sek.
card32 MaxTransferDelay	Maximale Verzögerung in $\frac{1}{16}$Millisek.
card8 Quality	Qualität in $\frac{1}{255}$
card8 LevelUp	Nächsthöheres Level oder 255
card8 LevelDown	Nächstniedrigeres Level oder 255
card8 QualityLayers	Anzahl der Layers dieses Levels
TransportInfoLayer QualityLayer[n]	Beschreibung der Layers dieses Levels

Tabelle 3.2: Die *TransportInfoLevel*-Klasse

TransportInfo	
Range<card64> WantedBytesPerSecond[n]	Pakete/Sek. gewünscht je Layer
Range<card32> WantedPacketsPerSecond[n]	Pakete/Sek. gewünscht je Layer
Range<card32> WantedFramesPerSecond	Frames/Sek. gewünscht
card32 WantedMaxTransferDelay	Max. Verzögerung gewünscht
card32 WantedMaxLossRate[n]	Max. Verlust gewünscht je Layer
card64 TotalBytesPerSecondLimit	Gesamtlimit Bytes/Sek.
card32 TotalPacketsPerSecondLimit	Gesamtlimit Pakete/Sek.
card32 TotalFramesPerSecondLimit	Gesamtlimit Frames/Sek.
card8 QualityLevels	Anzahl der Levels
card8 QualityLayers	Maximalzahl Layers eines Levels
card32 StartFramesPerSecond	Startwert für Frames/Sek.
TransportInfoLevel QualityLevel[m]	Beschreibung der Levels
TransportInfoLevel CurrentSetting	Aktuelle Einstellungen

Tabelle 3.3: Die *TransportInfo*-Klasse

und Paketrate, Skalierungsfaktoren, maximalem akzeptablen Paketverlust, maximaler Verzögerung usw.. *TransportInfo* (siehe Tabelle 3.3) enthält die komplette Beschreibung aller Levels der Kodierung.

Bei *QualityLevels*, *QualityLayers*, *StartFramesPerSecond* und *QualityLevel*[] handelt es sich um Konstanten der Kodierung, bei *Total**Limit* um globale Limits, *CurrentSetting* gibt die aktuelle Einstellung wieder.

Die *Wanted**-Werte sind von der Kodierung gewünschte Werte, also in der Regel die für die Qualitätseinstellung des Benutzers erforderlichen Werte. Anstatt hierfür eine der vorhandenen Level-Nummern anzugeben, werden diese Werte hier bewußt einzeln aufgeführt. Damit wird es möglich, von den festen Levels unabhängige Werte zu benutzen. Dies wird z.B. für die Audiokodierung notwendig, wenn der Benutzer eine "ausgefallene" Einstellung wählt, für die kein Level vorhanden ist (Beispiel: 4410 Hz, 16 Bit, Stereo - also niedrigste Sampling Rate, aber höchste Bit- und Kanalanzahl).

ExtendedTransportInfo : public *TransportInfo*	
StreamSrcDest SrcDest[n]	Quell- und Zieladresse, Tr.Cl. und Flowlabel je Layer

Tabelle 3.4: Die *ExtendedTransportInfo*-Klasse

ExtendedTransportInfo (siehe Tabelle 3.4) erweitert TransportInfo noch um die Quell- und Zieladressen sowie um einen Traffic Class- und Flowlabel-Wert für jedes Layer. Der Grund für die Trennung ist, daß diese zusätzlichen Werte nur für die RTP-Transport-Ebene relevant sind, nicht jedoch für die Kodierungs-Ebene.

Das Interface *EncoderInterface*

EncoderInterface ist ein Interface für einen Kodierer, welcher dieses implementiert. Die Kodierung wird duch einen 16-Bit-Wert (*getTypeID()*) identifiziert, von welchem die untersten 7 Bit für den Payload Type des RTP Paketes verwendet werden. *prepareNextFrame()* bereitet einen Frame für den Versand vor, *getNextPacket()* kopiert - solange vorhanden - Daten des Frames in ein Paket mit vorgegebener Maximallänge und setzt die Layer-Nummer des Paketes sowie die maximale Layer-Nummer für das aktuelle Level. Mit diesen Informationen kann nun der Sender das Paket entsprechend der Beschreibungen in der *ExtendedTransportInfo*-Klasse einer Traffic Class und/oder einem Flowlabel zuordnen (siehe Abschnitt zu *RTPSender*).

Die Methode *getTransportInfo()* gibt in der Klasse *TransportInfo* sämtliche für Transport und QoS-Management relevanten Daten zurück. Umgekehrt setzt *setTransportInfo*() Einstellungen der Kodierung, wobei natürlich die Konstanten Levelbeschreibungen unverändert bleiben. Zur Berechnung der Konstanten werden *getTransportInfo()* und *setTransportInfo()* jeweils die maximale Paketlänge und die Länge des Paket-Headers (z.B. IPv6 + UDP + RTP = 60) als Parameter übergeben.

Zur Unterstützung von Kodierern als Threads sind die Methoden *activate()* und *deactivate()* vorhanden. Innerhalb dieser kann der eigentliche Kodierer-Thread gestartet und gestoppt werden.

Das Interface *DecoderInterface*

Analog zur Kodierung ist dies ein Interface für einen Dekodierer. Die Identifizierung erfolgt analog zum EncoderInterface über *getTypeID()*. Ebenfalls vorhanden sind wieder die Methoden *activate()* und *deactivate()*, um z.B. einen Dekodier-Thread zu starteten oder zu stoppen.

Die eigentliche Dekodierung eines Paketes läuft in zwei Schritten ab:

1. *checkNextPacket()* überprüft ein Paket auf Gültigkeit und ordnet es einem Layer zu. Dies ist notwendig, da die untergeordnete Schicht (RTP-Transport-Ebene) keine Kenntnisse über Layers besitzt, jedoch für Rückmeldungen (RTCP Receiver Reports) die Pakete den einzelnen Layern zuordnen muß.

2. *handleNextPacket()* übernimmt die eigentliche Dekodierung der Daten.

Die Interfaces *EncoderRepository* und *DecoderRepository*

Um ein einfaches Wechseln der Kodierungen während des Sende- und Empfangsvorganges zu ermöglichen, wurden die Interfaces *EncoderRepositoryInterface* und *DecoderRepositoryInterface* definiert. Encoder- bzw. Decoder-Repositories verwalten Listen von *EncoderInterface*s bzw. *DecoderInterface*s, wobei mittels der Methoden *selectEncoderForTypeID()* bzw. *selectDecoderForTypeID()* der En- bzw. Dekoder mit der gegebenen TypeID als aktuell eingestellt wird. Dabei werden auch automatisch die notwendigen *deactivate()* und *activate()*-Aufrufe durchgeführt.

Das Repository implementiert selbst *EncoderInterface* bzw. *DecoderInterface* und leitet alle Aufrufe zum aktuellen En- bzw. Decoder um. Somit kann einfach ein Repository anstelle eines eigentlichen En- bzw. Dekoders in den RTP-Klassen verwendet werden.

Die Klasse *RTPSender*

Diese von *TimedThread* abgeleitete Klasse stellt den RTP-Sender dar, welcher Daten paketweise von einem Kodierer (Implementation von *EncoderInterface*) übernimmt und mit gegebenen Adressen (IP-Adresse, Port, Traffic Class und Flowlabel) für jedes Layer den Versand über ein Socket übernimmt.

Die eigentliche Sendeschleife ist mittels des *TimedThread* implementiert. Durch die Methode *prepareNextFrame()* des *EncoderInterface* wird der nächste Frame zum Versand vorbereitet. Anschließend werden solange Pakete erzeugt und mittels *getNextPacket()* mit Daten gefüllt bis entweder

- der Frame komplett gesendet ist,
- ein neuer *timerEvent()*-Aufruf ansteht (Senden des Frames dauerte zu lange) oder
- das Bytes pro Sekunde oder Pakete pro Sekunde-Limit aus *TransportInfo* überschritten ist (Fehler in Kodierung oder QoS-Beschreibung).

Die Pakete werden über die von *getNextPacket()* zurückgegebenen Layer gesendet, wobei Zieladresse, Traffic Class und Flowlabel aus der *ExtendedTransportInfo*-Klasse verwendet werden. Dies entspricht dem Layering Control im *Coral*-Konzept.

Um zu verhindern, daß das IP-Protokoll - zumindest im lokalen Ethernet-Netzwerk - die einzelnen Pakete fragmentieren muß, um diese dann über das Netzwerk zu übertragen, wird die Länge des RTP-Payloads auf 1376 beschränkt. Dies sind mit RTP-Header (12 Bytes), maximal 16 möglichen CSRCs (16 * 4 Bytes), UDP-Header (8 Bytes) und IPv6 -Header (40 Bytes) maximal 1500 Bytes, was dem maximalen Payload eines Ethernet-Frames entspricht.

Wie schon *EncoderInterface*, so besitzt auch *RTPSender* die Methoden *getTransportInfo()* und *setTransportInfo()*, wobei hier jedoch die erweiterte Klasse *ExtendedTransportInfo* zum Einsatz kommt. Somit ist es möglich, nicht nur die Einstellungen der Kodierung selbst, sondern auch die Adreß-Werte für den Transport - also insbesondere Traffic Classes und Flowlabels - für die einzelnen Layers während der Übertragung zu wechseln.

Die Klasse *RTPReceiver*

Abgeleitet von *Thread* wartet diese Klasse an einem Socket auf RTP-Pakete. Diese werden mit der *checkNextPacket()*-Methode eines von *DecoderInterface* abgeleiteten Dekoders auf Gültigkeit überprüft, einem Layer zugeordnet und mit der Methode *handleNextPacket()* des Dekoders dekodiert. Zudem wird mit der Sequenznummer und dem Zeitstempel des RTP-Headers die Berechnung von Paketverlusten und des Jitters durchgeführt, welche mittels der Klasse *RTCPSender* in Rückmeldungen wieder an den Sender geschickt werden.

Die Klasse *RTCPSender*

Diese von *TimedThread* abgeleitete Klasse sendet RTCP-APP und RTCP-BYE-Pakete zu einem Sender. Zusätzlich verwaltet sie eine Liste von RTCP-SDES-Einträgen, welche zusammen mit den von einem *RTPReceiver* erzeugten RTCP Receiver Reports in zufälligen Abständen übertragen werden.[4]

Die Klasse *RTCPReceiver*

Hier ist der RTCP-Empfänger - von *Thread* abgeleitet - implementiert, welcher RTCP-Pakete empfängt und dekodiert. Die dekodierten Inhalte werden an ein Objekt der Klasse *RTCPAbstractServer* weitergegeben.

Die Klasse *RTCPAbstractServer*

Diese von *TimedThread* abgeleitete, abstrakte Klasse ist zentraler Bestandteil eines auf den RTP/RTCP-Klassen aufbauenden Servers und übernimmt grundlegene Server-Aufgaben wie die An- und Abmeldung der Clients und das Entfernen von Clients mit Timeout.

Die Anmeldung läuft über RTCP-SDES-CNAME Nachrichten. Wird eine solche Nachricht mit einem unbekannten neuen Teilnehmer empfangen, so wird dieser als neuer Client aufgenommen. Bei Empfang von RTCP-BYE wird der Client wieder entfernt. Clients haben einen Timeout, wenn die verstrichene Zeit seit dem letzten Receiver Report größer als eine Konstante ist. In diesem Fall wird der Client ebenfalls entfernt.

Der eigentliche Server ist von *RTCPAbstractServer* abgeleitet. Es implementiert Funktionen zum Initialisieren und Entfernen eines Clients sowie zur Behandlung von RTCP-APP Nachrichten und RTCP Reports

3.2.3 Die Kodierung der Audiodaten

Qualitätsstufen für Audiodaten

Die Qualität eines Audio-Stromes ist durch die Sampling Rate, die Anzahl der Bits pro Sample und die Anzahl der Kanäle (Mono bzw. Stereo) gegeben, wobei in RTP AUDIO folgende Einstellungsmöglichkeiten vorhanden sind:

- 19 Sampling Rate-Einstellungen: 4410 Hz bis 44100 Hz in Schritten von 2205 Hz,

[4] Zur Berechnung der zufälligen Abstände wird der Algorithmus aus [RFC 1889] benutzt.

RTP AUDIO Qualitäten					
Level	Sampling Rate	Bits	Kanäle	Bytes/Sekunde (Payload)	Layers (AAE)
0	4410 Hz	4	Mono	2205	1
1	6615 Hz	4	Mono	3307	1
2	8820 Hz	4	Mono	4410	1
3	8820 Hz	8	Mono	8820	1
4	11025 Hz	8	Mono	11025	1
5	11025 Hz	8	Stereo	22050	2
6	13230 Hz	8	Stereo	26460	2
7	15435 Hz	8	Stereo	30870	2
8	17640 Hz	8	Stereo	35280	2
9	19845 Hz	8	Stereo	39690	2
10	22050 Hz	8	Stereo	44100	2
11	22050 Hz	12	Stereo	66150	3
12	24255 Hz	12	Stereo	72765	3
13	26460 Hz	12	Stereo	79380	3
14	28665 Hz	12	Stereo	85995	3
15	30870 Hz	12	Stereo	92610	3
16	33075 Hz	12	Stereo	99225	3
17	35280 Hz	12	Stereo	105840	3
18	35280 Hz	16	Stereo	141120	3
19	37485 Hz	16	Stereo	149940	3
20	39690 Hz	16	Stereo	158760	3
21	41895 Hz	16	Stereo	167580	3
22	44100 Hz	16	Stereo	176400	3

Tabelle 3.5: Die Qualitäts-Levels von RTP AUDIO

- 4 Bits-Einstellungen 4 Bit, 8 Bit, 12 Bit, 16 Bit,
- 2 Kanäle-Einstellungen: Mono (1) und Stereo (2).

Aus diesen insgesamt 152 Möglichkeiten wurden 23 Qualitäts-Levels ausgewählt, welche eine sinnvolle Sortierreihenfolge besitzen (siehe Tabelle 3.5. Zur Beschreibung der Layers-Spalte siehe Abschnitt 3.2.3 zu Advanced Audio Encoding).

Wie bereits im Abschnitt 3.2.2 zu *TransportInfo* erwähnt, ist es in RTP AUDIO auch möglich, Qualitäten zu benutzen, welche nicht in der Tabelle vorkommen. Lassen Beschränkungen bzgl. der Bandbreite diese gewünschte Einstellung jedoch nicht zu, so wird der bestmögliche passende Wert aus der Tabelle verwendet. Die QoS-Beschreibungen für die 23 Levels der Tabelle befinden sich in Anhang C, für die restlichen Einstellungen siehe die Beschreibung zum Programm EncoderInfo (Abschnitt A.1.2).

Anzahl der Bits	Mono	Stereo
4	[L1|L2]	[L1|L2] [R1|R2]
8	[L1]	[L1] [R1]
12	[La1] [La2] [Lb1|Lb2]	[La1] [La2] [Lb1|Lb2] [Ra1] [Ra2] [Rb1|Rb2]
16 (Big End.)	[La|Lb]	[La] [Lb] [Ra] [Rb]
16 (Little End.)	[Lb|La]	[Lb] [La] [Rb] [Ra]

Tabelle 3.6: Die Audioformate in RTP AUDIO

SimpleAudioPacket	
card32 FormatID	Identifikationsnummer: 0x74661234
card16 SamplingRate	Sampling Rate
card8 Channels	Anzahl der Kanäle
card8 Bits	Anzahl der Bits
card64 Position	Aktuelle Position in Nanosekunden
card64 MaxPosition	Maximals Position in Nanosekunden
card8 ErrorCode	Fehlernummer (z.B. Datei nicht gefunden)
card8 Flags	Simple Audio Flags
char Data[]	Payload (AudioDaten bzw. *MediaInfo*)

Tabelle 3.7: Der Header eines Simple Audio Paketes.

Das Format der Audiodaten

Die Audiodaten haben dabei das in der Tabelle 3.6 angegebene Format, wobei

- Ln/Rn den n-ten Sample-Wert des linken bzw. rechten Kanals und
- a/b die oberen 8 bzw. die unteren 8 oder 4 Bits bezeichnen.
- Ganze Bytes sind jeweils mit Indexklammern [ ] gekennzeichnet.

Zu beachten ist, daß bei 4-Bit- und 12-Bit-Kodierungen immer zwei Samples zusammengefaßt werden müssen, um eine ganze Anzahl von Bytes zu erreichen! Diese beiden Kodierungen sind reine Software-Lösungen. Vor der Ausgabe müssen diese wieder in 16- bzw. 8-Bit-Format durch Erweitern mit Null-Bits konvertiert werden.

Simple Audio Encoding

Dies ist die einfachste Audiokodierung. Die Daten werden mit 15 Frames pro Sekunde in einem einzigen Layer übertragen. Eine Fehlerkorrekturmöglichkeit besteht daher nicht. Zusätzlich zu den Audiodaten wird zweimal pro Sekunde die Struktur *MediaInfo* (siehe Tabelle 3.8) übertragen, welche Informationen wie Titel und Interpret der aktuellen Übertragung enthält. Der Paket-Header für Simple Audio Encoding hat das in der Tabelle 3.7 angegebene Format.

MediaInfo	
card64 StartTimeStamp	Zeitstempel für Beginn des Mediums
card64 EndTimeStamp	Zeitstempel für Ende der Mediums
char Title[64]	Titel
char Author[64]	Autor
char Comment[64]	Kommentar

Tabelle 3.8: Die MediaInfo-Klasse

Bits und Kanäle	Layer #0	Layer #1	Layer #2
4, Mono	4 Bit links (Paare)	-	-
8, Mono	8 Bit links	-	-
12, Mono	obere 8 Bit links	untere 4 Bit	-
16, Mono	obere 8 Bit links	untere 8 Bit	-
4, Stereo	4 Bit links (Paare)	4 Bit rechts (Paare)	-
8, Stereo	8 Bit links	8 Bit rechts	-
12, Stereo	obere 8 Bit links	obere 8 Bit rechts	untere 4 Bit (l+r!)
16, Stereo	obere 8 Bit links	obere 8 Bit rechts	untere 8 Bit (l+r!)

Tabelle 3.9: Die Verwendung der Layer bei Advanced Audio Encoding.

Die Positionen innerhalb des Mediums werden grundsätzlich in Nanosekunden angegeben. Dies ermöglicht sowohl eine ausreichende Länge ($>$ 300 Jahre) als auch eine extrem exakte Positionierug. Diese kann dann z.B. zur Synchronisation mit einem Video verwendet werden, z.B. eine Langzeitaufnahme einer Überwachungskamera oder Hochgeschwindigkeitsaufnahmen für extreme Zeitlupe-Darstellung. Der *Flags*-Wert gibt an, ob es sich beim Payload um Daten (SAF_Data) oder eine *MediaInfo*-Struktur (SAF_MediaInfo) handelt.

Ein Simple Audio Paket enthält nur Daten eines einzigen Frames; ein Frame wird dabei ggf. auf mehrere Pakete aufgeteilt, falls die Framegröße die Größe des Payloads (1376 für RTP, siehe Abschnitt 3.2.2) überschreitet.

Advanced Audio Encoding

Diese Kodierung benutzt je nach Level ein bis drei Layer (siehe dazu den Layer-Abschnitt der Qualitäten-Tabelle 3.5). Es werden 35 Frames pro Sekunde gesendet, wobei zweimal pro Sekunde ein MediaInfo-Paket über Layer #0 verschickt wird. Es wird die in Tabelle 3.9 dargestellte Einteilung in die verschiedenen Layer verwendet.

Der Paket-Header für Advanced Audio Encoding ist bis auf die Unterschiede in Tabelle 3.10 identisch mit dem aus Simple Audio Encoding: Der *Flags*-Wert gibt hier an, worum es sich beim Payload handelt: *MediaInfo* oder linker bzw. rechter Kanal mit oberen bzw. unteren Bits.

Mit Advanced Audio Encoding ist nun auch eine Korrektur bei Paketverlusten möglich:

- Die Daten eines fehlenden rechten Kanals können durch die des linken ersetzt werden und

AdvancedAudioPacket	
card32 FormatID	Identifikationsnummer: 0x74662909
...	...
card8 Flags	Advanced Audio Flags
card16 Fragment	Nummer des Fragmentes eines Frames

Tabelle 3.10: Die Erweiterung des Simple- zum Advanced Audio Header

umgekehrt. Man erhält damit eine Mono-Ausgabe.

- Fehlende untere Bit-Blöcke (4- oder 8-Bit) können auf Null gesetzt werden bzw. falls die oberen 8 Bit ersetzt wurden (und nur dann!) durch die unteren des anderen Kanals ersetzt werden. Man beachte: Die unteren Bit-Blöcke werden zwar in einem Layer gesendet (aufgrund der gleichen Priorität), jedoch getrennt nach Kanal in unterschiedlichen Paketen!

Die Verteilung auf mehrere Layers und die Fehlerkorrektur machen jedoch eine Pufferung notwendig. Es war daher erforderlich, die Dekodierung in einem eigenen Thread zu implementieren, der in regelmäßigen Abständen die Liste der empfangenen Fragmente durchläuft, ggf. Korrekturen durchführt und die Daten schließlich ausgibt.

Ein Advanced Audio Paket enthält nur Daten eines einzigen Frames und Layers. Analog zu Simple Audio Encoding ist auch hier eine Aufteilung der Daten auf mehrere Pakete ggf. notwendig.

Der RTP Audio Server

Die Klasse *AudioServer* basiert wie bereits erwähnt auf der Klasse *RTCPAbstractServer* und startet für jeden neuen Client einen eigenen *RTPSender*-Thread für die Unicast-Übertragung der Daten. Die Audiodateien sind in Audiolisten zusammengefaßt, welche ein Client beim Server anfordern kann. Diese Audiolisten können MP3[5]-, WAV- oder weitere Audiolisten enthalten. Für den Client stellt sich eine solche Liste als ein einziges, zusammenhängendes Audio-Medium dar. Für eine genauere Beschreibung sei auf die [RTP Audio] Homepage verwiesen.

Zustandsänderungen des Clients, wie das Ändern von Position, Qualität, Medium usw. werden über RTCP-APP Nachrichten an den Server übertragen. Diese haben folgendes Format: Mittels der Sequenznummer wird verhindert, daß durch Paketvertauschungen veraltete Nachrichten verarbeitet werden. In den weiteren Feldern wird der **komplette** Zustand des Clients mit Wiedergabe-Modus, Qualität, Kodierung, Bandbreite-Limit und Medium festgehalten. Positionsänderungen werden durch *StartPosition* gesetzt; um bei Paketverlusten und -vertauschungen die Gültigkeit der neuen Position zu überprüfen, existiert hierfür eine weitere Sequenznummer: *PosChgSeqNumber*.

Da beim Versand über UDP Paketverluste möglich sind, wird das komplette *AudioClientAppPacket*-Paket als RTCP SDES-PRIV Nachricht zusammen mit den anderen RTCP SDES Nachrichten regelmäßig[6] an den Server gesendet. Zusammen mit dem Eintrag *RestartPosition*, welcher die jeweils letzte vom Client empfangene Position enthält, wird es nun für den Server einfach möglich, den Client nach einem Server-Neustart bzw. Client-Timeout[7] mit den alten Einstellungen

[5] Die MP3-Dateien werden vom Server mittels libmpegsound entpackt.

[6] In zufälligen Abständen gemäß dem Algorithmus aus RFC 1889. RTP Audio: ca. 1 bis 2 Sekunden.

[7] Siehe dazu den Abschnitt 3.2.2 über Client-Timeouts in *RTCPAbstractServer*.

AudioClientAppPacket	
card32 FormatID	Identifikationsnummer: 0x75003388
card16 SequenceNumber	Sequenznummer
card16 PosChgSeqNumber	Sequenznummer für Pos.-wechsel
card16 Mode	Modus (Wiedergabe bzw. Pause)
card16 SamplingRate	Sampling Rate
card8 Channels	Anzahl der Kanäle
card8 Bits	Anzahl der Bits
card16 Encoding	TypeID der Kodierung
card32 BandwidthLimit	Obere Grenze für Bytes/Sekunde
card64 StartPosition	Startposition in Nanosek.
card64 RestartPosition	Wiederanlauf-Position in Nanosek.
char MediaName[128]	Name des Audio-Mediums

Tabelle 3.11: Das APP-Paketformat von RTP AUDIO

und ab der alten Position wieder anlaufen zu lassen.

Der Server besitzt eine Anbindung an den QoS-Manager (siehe Kapitel **??**), bei welchem alle Ströme registriert werden. Der QoS-Manager teilt dabei den einzelnen Strömen anhand der Qualitätsberechreibungen in Form der Klasse *ExtendedTransportInfo* Bandbreite sowie Traffic Class zu. Mit dem QoS-Manager wurde die ursprüngliche Idee eines Congestion Management auf Basis der Beobachtung des Netzwerkverkehres mittels *libpcap* ersetzt. Siehe dazu Abschnitt A.4.

Die RTP AUDIO Clients

Für RTP AUDIO existieren vier verschiedene Clients: Ein Textmodus-Client, ein Textmodus-Client zu Verifikationszwecken (*vclient*) und ein graphischer Client (*QClient*), welche in C++ geschrieben wurden, sowie ein Java-Client. Alle Clients basieren auf der Klasse *AudioClient*, welche die komplette Client-Funktionalität basierend auf den RTP/RTCP-Klassen implementiert.

Die beiden Textmodus-Clients dienen hauptsächlich Test- und Verifikationszwecken, wobei *vclient* eine gegebene Anzahl von Threads mit *AudioClient*-Objekten startet und zufällig Aktionen wie Positions- und Qualitätsänderungen sowie Medienwechsel mit gegebenen Wahrscheinlichkeiten durchführt. Ihre genaue Beschreibung - sowie die der anderen Programme auch - ist in der Benutzerdokumentation im Anhang A zu finden.

Für den graphischen *QClient* (siehe Abbildung 3.13) wurde die C++-GUI-Bibliothek *Qt* (siehe [Qt] Homepage) verwendet - aufgrund ihrer Stabilität, Leistungsfähigkeit, Performanz und guten Dokumentation. Es können mehrere Ausgabegeräte gleichzeitig für die Ausgabe der Audiodaten verwendet werden. Zur Verfügung stehen neben dem Audio-Device selbst noch ein Spectrum Analyzer, ein Debug-Device zur Ausgabe von Verzögerungszeiten und ein Null-Device[8]. Somit ist auch die Verwendung des Clients auf Rechnern ohne Soundhardware möglich. Der Client besitzt zudem eine Vielzahl von Statusanzeigen, welche - aggregiert oder für jedes Layer einzeln - den Übertra-

[8]Die Audio-Daten werden einfach verworfen.

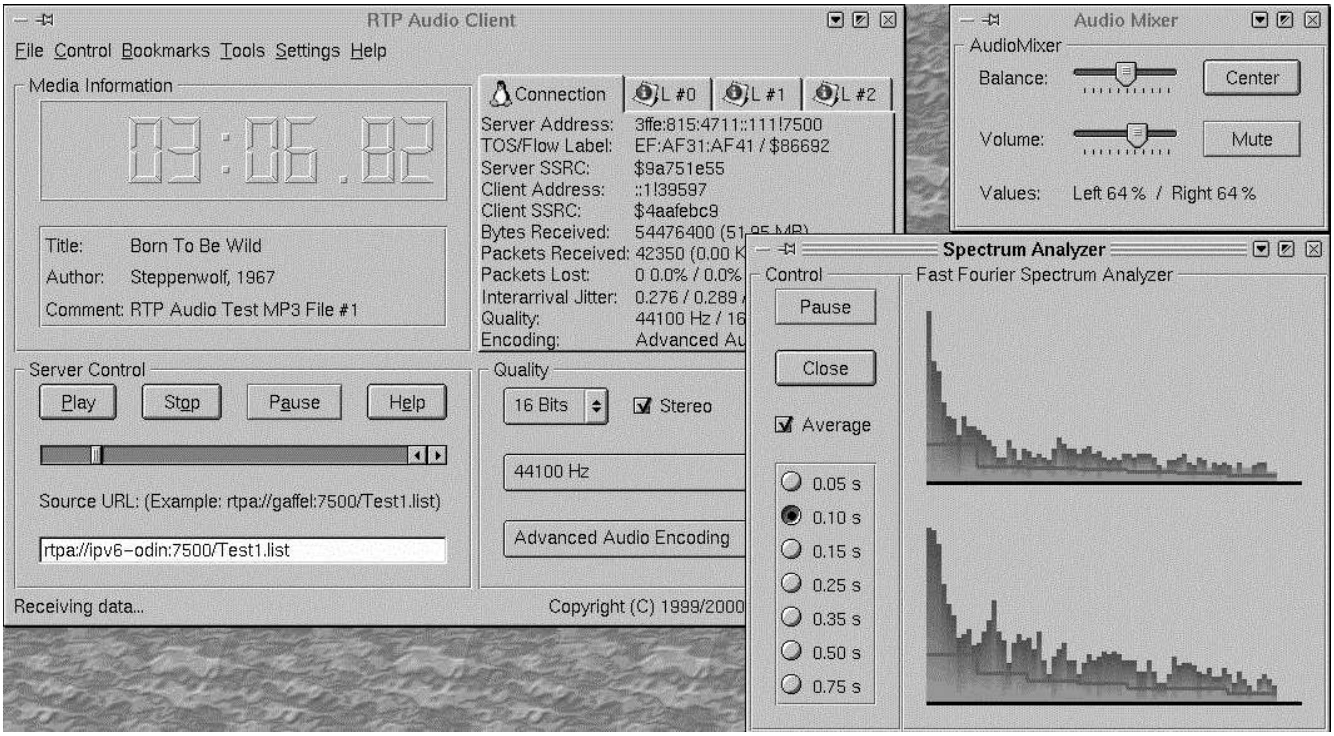


Abbildung 3.13: Der QClient mit Spectrum Analyzer und Audio-Mixer

gungsstatus in Form von Adressen, Traffic Classes, Flowlabels, Bytes/Sekunde, Pakete/Sekunde, Verluste, Jitter, usw. darstellen. Für eine genaue Beschreibung sei auf die Benutzerdokumentation im Anhang A bzw. die Online-Hilfe verwiesen.

Der Java Client und die PROG4D-Anbindung

Zur Realisierung eines gemeinsamen PROG4D/RTP AUDIO-Clients war es aufgrund der Java-Implementation des PROG4D-Clients notwendig, einen RTP AUDIO Client in Java ([Blackdown] Version 1.2.2-RC4) zu implementieren. PROG4D ist ein System zum Übertragen von interaktiven 3D-Videos über RTP.

Da in Java momentan keine Klasse für den Zugriff auf das Audio-Device[9] existiert, war von Anfang an geplant, diesen Audio-Device-Zugriff in C/C++ zu implementieren und diesen dann mittels JNI an den Java-Code anzubinden. Bei JNI handelt es sich um das Java Native Interface, welches eine Dynamic Linked Library mit in C geschriebenen Funktionen lädt, welche dann vom Java-Interpreter ausgeführt werden können.

Aufgrund der Tatsache, daß Java weder IPv6 noch die Verwendung von Traffic Classes unterstützt und sich die Programmierung von JNI nach ersten Tests als äußerst aufwendig, fehlerträchtig und ineffizient erwies und zudem kleinere Inkompatibilitäten zwischen den einzelnen Java-Revisionen (RC1, RC2, RC3 und RC4; die verschiedenen Java-Versionen sind bei der JNI-Verwendung zudem völlig inkompatibel) immer wieder für langwieriges Suchen nach Umgehungsmöglichkeiten sorgten, wurde nur die Klasse *AudioClient* über JNI für Java zugänglich gemacht.

[9] Java kann zwar verschiedene Audioformate abspielen, aber direktes Schreiben von "rohen" Audiodaten ist - zumindest bis zur Version 1.2.2-RC4 (Blackdown) - nicht möglich.

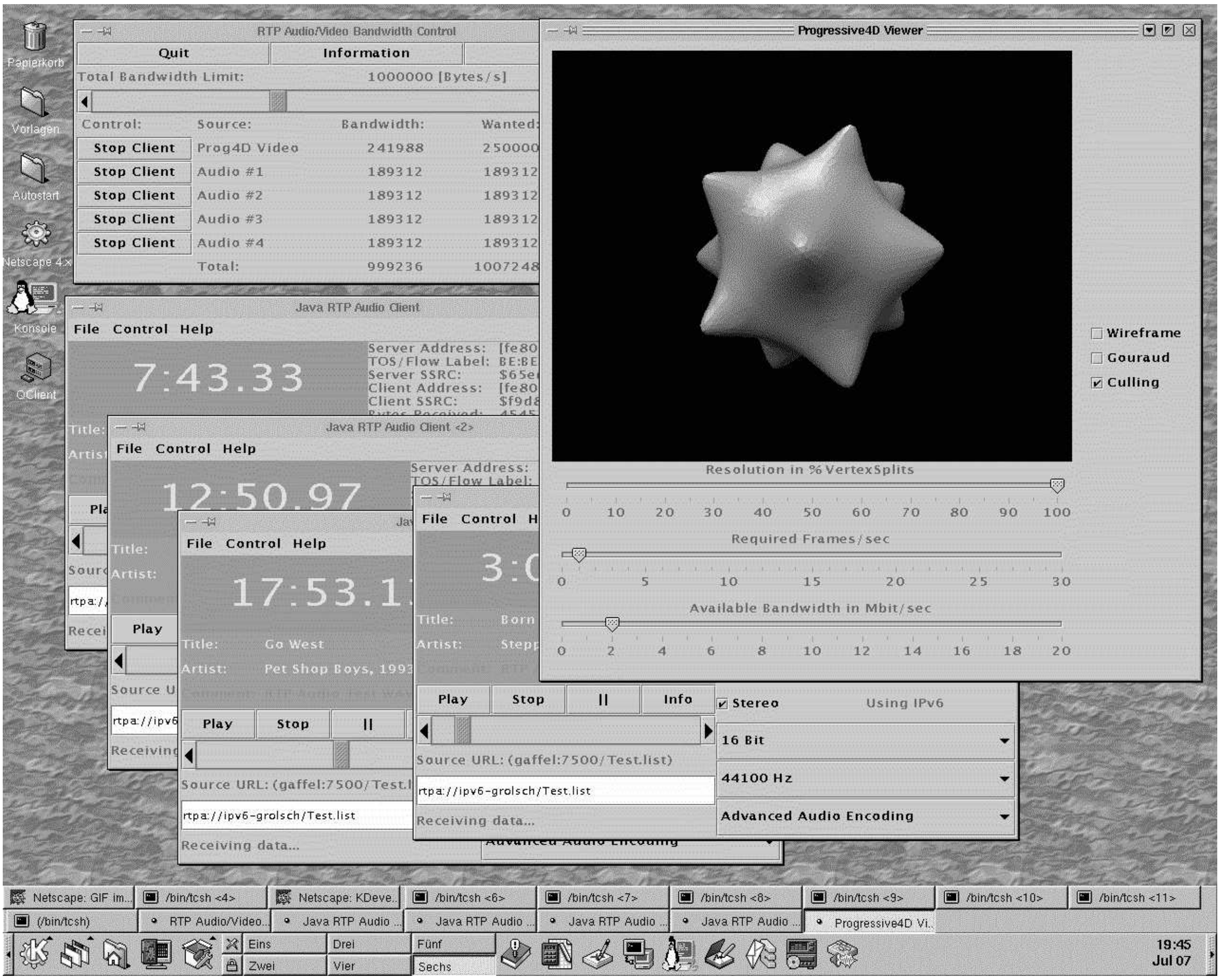


Abbildung 3.14: MainControl mit vier RTP Audio- und dem Prog4D-Client

Der Java-Code des Audio-Clients beschränkt sich somit auf die graphische Oberfläche mit Swing - der Empfang, die Dekodierung und die Ausgabe der Audiodaten werden vom C++-Code übernommen. Somit entfielen die meisten Java-Probleme und es wird zusätzlich noch ein erheblicher Gewinn an Rechenzeit erreicht (Optimierter Maschinencode im Gegensatz zu interpretiertem Java Bytecode).

Mit dem so implementierten Java-Client war es nun möglich, einen gemeinsamen Prog4D- und RTP Audio-Client (*MainControl*) zu realisieren. Der Prog4D/RTP Audio-Client kann mehrere RTP Audio-Verbindungen und eine Prog4D-Verbindung gleichzeitig starten. Dabei kann die Gesamt-Bandbreite mittels eines Scrollbalkens beschränkt werden. Die somit eingestellte Bandbreite wird dann unter den Clients verteilt. Siehe dazu die Abbildung 3.14.

3.3 Die Meßwerkzeuge

VON THOMAS DREIBHOLZ

Geeignete Software zur Durchführung von Messungen mit Unterstützung von IPv6, Traffic Classes und Flowlabels war aufgrund der Neuheit noch nicht verfügbar. Es war daher notwendig, diese vollständig selbst zu implementieren. Als Basis wurde die Bibliothek *libpcap* verwendet, welche auch in *tcpdump* zum Einsatz kommt und ein Mitschneiden des gesamten Netzverkehres über ein Netzwerk-Device wie *eth0*[10] ermöglicht. Bei Netzen mit Broadcast-Medium wie Ethernet (CSMA/CD, nicht Switched Ethernet!) oder Token-Ring läßt sich das Device in einen Überwachungsmodus (promiscuous mode) schalten, wobei dann sämtliche über das Netz gesendete Pakete mitgeschnitten werden). Für die Meßzwecke war es schon ausreichend, nur die Paket-Header mitzuschneiden, was den Rechenzeitaufwand gering hält.
Das Meßwerkzeugpaket besteht aus drei Programmen:

1. *NetLogger* schneidet mittels *libpcap* die Paketheader eines Netzwerk-Devices mit, analysiert diese und schreibt in einstellbaren Abständen Summen von übertragenen Bytes und Paketen gruppiert nach Protokoll und Traffic Class in eine Logdatei. Zusätzlich kann dies noch für einzelne, auswählbare Ströme durchgeführt werden. *NetLogger* erfordert zur Ausführung *root*-Rechte, da das Mitschneiden des Netzwerkverkehres sicherheitskritisch ist (z.B. Übertragung unverschlüsselter Passwörter u.ä..).

2. *NetAnalyzer* liest die Logdatei von *NetLogger* ein und erzeugt für ausgewählte Daten ein GNU PLOT Skript sowie Plotdateien. Diese können dann mittels des Tools GNU PLOT graphisch dargestellt werden.

3. *RTTP* (Round Trip Time Pinger) führt Messungen der Round Trip Time zu einer Liste von Rechnern mit gegebenen Traffic Classes durch. Dies geschieht mittels ICMPv4 bzw. ICMPv6 Echo Requests und Echo Replies für IPv4 bzw. IPv6. Dazu wird beim Versand der aktuelle Zeitstempel in das Echo Request-Paket geschrieben. Bei Empfang des Echo Replys - der Paket-Payload ist eine Kopie des Echo Request-Payload - wird der Empfangs-Zeitstempel mit dem Zeitstempel des Antwort-Paketes verglichen.
   Mit der so ermittelten Round Trip Time ϑ wird dann - wie bei TCP - die geglättete Round Trip Time errechnet:
   $$RTT_{Neu} := \alpha * RTT_{Alt} + (1 - \alpha) * \vartheta$$
   Der Default-Wert für α ist $\frac{7}{8}$ (wie bei TCP).

4. Der Versand der Echo Requests wird in zufälligen Intervallen durchgeführt, um Meßfehler durch Bursts zu vermeiden. Gehen ICMP-Pakete verloren.[11], so wird die Differenz zwischen Systemzeit und Zeit des letzten **beantworteten** Echo Requests in regelmäßigen Abständen als neue "gemessene" Roundtripzeit benutzt. Somit wird erreicht, daß die Roundtripzeit ansteigt, wenn keine Replies empfangen werden.

[10] eth0 bezeichnet unter Linux die erste Ethernet-Karte.

[11] Ein Echo Request wird als verloren angenommen, falls die Zeit seit dem letzten Echo Request größer als Max{2.5 Sekunden, maximale wirklich gemessene Roundtripzeit} beträgt.

Zusätzlich zu den Meßwerkzeugen wurden noch Testsender und Testempfänger für UDP und TCP entwickelt.

3.4 Der QoS-Manager

von Simon Vey

Die Aufgabe des QoS-Managers ist es, die verfügbaren Ressourcen fair unter allen Strömen aufzuteilen. Jeder Strom hat seine eigenen statischen QoS-Anforderungen, die bei der Verteilung in die QoS-Klassen berücksichtigt werden müssen. Die QoS-Parameter der einzelnen Klassen wiederum können aber über die Zeit variieren. Alle N Sekunden werden die Ströme neu auf die QoS-Klassen verteilt. Zwischen zwei solchen Neuverteilungen werden die Ströme nur innerhalb ihrer Klassen skaliert, da dies einen geringeren Verwaltungsaufwand bedeutet. Insgesamt soll keine optimale, sondern eine schnelle Verteilung erreicht werden, um auch bei sehr vielen zu verwaltenden Strömen noch schnell reagieren zu können. Der hier vorgestellte Verteilungsalgorithmus versucht, die Bandbreite fair unter allen Strömen zu verteilen (strombasierte Fairneß).

3.4.1 Die Stromhierarchie

Im QoS-Manager können mehrere Sessions gleichzeitig verwaltet werden. Dabei steht eine Session hier für einen Macroflow, d.h. sie beinhaltet alle Anwendungsströme, die zu demselben Zielrechner gesendet werden. So kann man sich zum Beispiel vorstellen, daß in einer MPEG-4 Szene verschiedene Audio- und Videoströme zusammengefaßt werden. Im Prinzip wären auch mehrere Sessions zu einem Zielrechner vorstellbar, was aber in der aktuellen Version noch nicht implementiert ist. Jede Session kann im Prinzip beliebig viele Anwendungsströme beinhalten, die in verschiedenen Levels senden können. Jeder Level wiederum beinhaltet mindestens einen Layer (Transportstrom). Durch Hoch- oder Herunterskalieren eines Anwendungsstroms, also durch einen Levelwechsel, können so Anzahl und Größe der Layer des Stroms variiert werden. Abbildung 3.15 gibt einen schematischen Überblick über die Stromhierarchie.

3.4.2 Statische QoS-Beschreibung

Um die Bandbreite fair unter den einzelnen Strömen aufteilen zu können, braucht der QoS-Manager Informationen über die Anforderungen und die Beschaffenheit der Anwendungsströme. Hierzu übergibt jeder Strom bei seiner Anmeldung eine statische QoS-Beschreibung an den Manager, bestehend aus Level- und Layerbeschreibungen. Ein Beispiel einer solchen Beschreibung ist in Abbildung 3.16 zu sehen. Jeder Strom besitzt mindestens einen Level. In der Levelbeschreibung wird festgelegt, mit wie vielen Frames pro Sekunde in diesem Level gesendet werden soll (fps_start * fps_scale). Des weiteren enthält die Levelbeschreibung die obere Grenze für die akzeptable Ende-zu-Ende-Verzögerung, die mit diesem Level erreichte Qualität und die Informationen, in welche Level von diesem Level aus hoch- und herunterskaliert werden soll.
Jeder Level enthält mindestens einen Layer. In der Layerbeschreibung wird die Frame-Größe und die maximal tolerierbare Verlustrate für den jeweiligen Layer festgehalten.
Aus den Frames pro Sekunde eines Levels und der Frame-Größe aller Layer dieses Levels kann dann die Bandbreite errechnet werden, mit der ein Strom in einem bestimmten Level sendet.

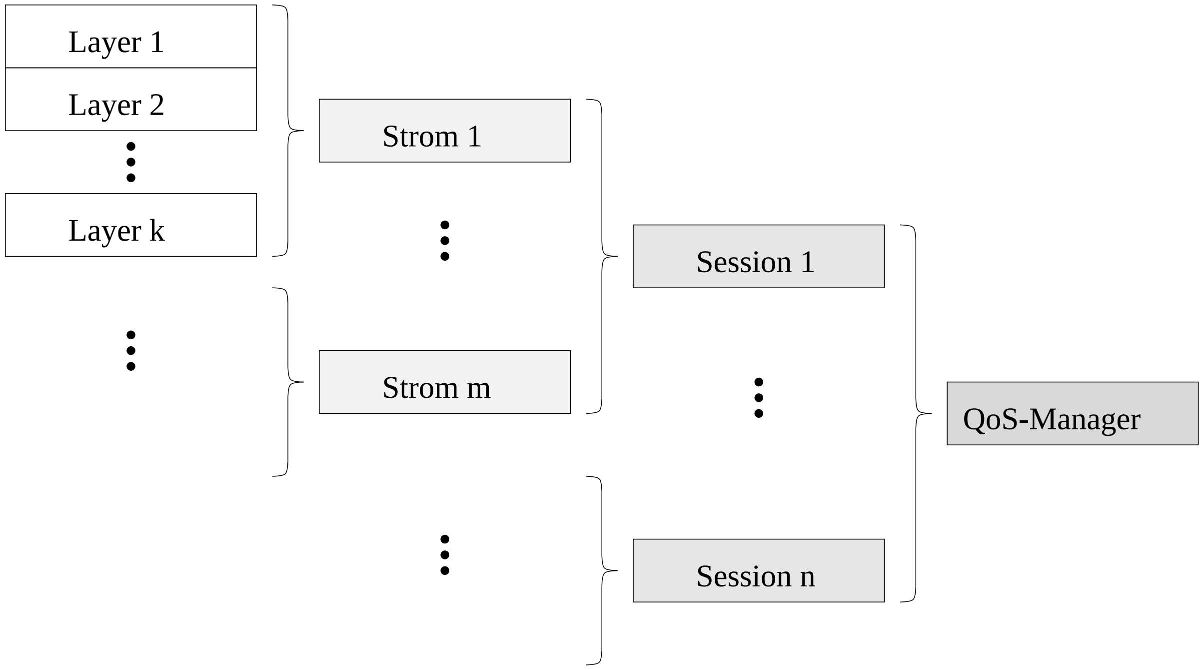


Abbildung 3.15: Hierarchische Strombeschreibung im QoS-Manager

3.4.3 Hinzufügen eines Stroms

In Abbildung 3.17 wird dargestellt, wie ein neuer Strom in das QoS-Management aufgenommen wird. Zuerst wird überprüft, ob es bereits eine Session zu der vom Strom gewünschten Adresse gibt. Wenn das der Fall ist, wird der Strom dieser Session hinzugefügt. Andernfalls wird eine neue Session erzeugt, die dann zu Anfang nur den neuen Strom enthält. Prinzipiell wäre auch denkbar, daß es mehrere Sessions zur selben Internet-Adresse geben kann (verschiedene Anwendungen auf demselben Rechner). Da aber noch kein entsprechendes Session-Management existiert, wurde der Session-Gedanke zunächst wie oben beschrieben verwirklicht.
Der nächste Schritt ist es zu ermitteln, mit welchem Level der neue Strom integriert werden soll. Ausgehend vom Grundgedanken der strombasierten Fairneß wird versucht, den Strom mit einer Qualität einzuordnen, die der durchschnittlichen Qualität der schon existierenden Ströme entspricht. Existieren noch keine anderen Ströme im Management, so wird der neue Strom nach Möglichkeit mit maximaler Qualität, also mit seinem höchsten Level, eingestuft. Ansonsten wird die Qualität nach folgender Formel ermittelt:

$$QoS_{neu} = \frac{1}{n+1} * \sum_{i=1}^{n} QoS_i$$

QoS$_{neu}$ ist hierbei der QoS-Wert des neuen Stroms, n die Anzahl der bereits existierenden Ströme und QoS$_i$ der QoS-Wert des i-ten Stroms im Management. Es wird durch n+1 anstatt durch n geteilt, um den neuen Strom schon bei der Durchschnittsbildung zu berücksichtigen. Dadurch wird der neue Strom eher zu vorsichtig eingestuft als zu aggressiv. Für den Fall, daß eigentlich genügend Bandbreite für eine bessere Qualität zur Verfügung stehen würde, wird ganz am Ende des Einfüge-Algorithmus versucht, den neuen Strom noch hochzuskalieren (s.u.). Der gewünschte

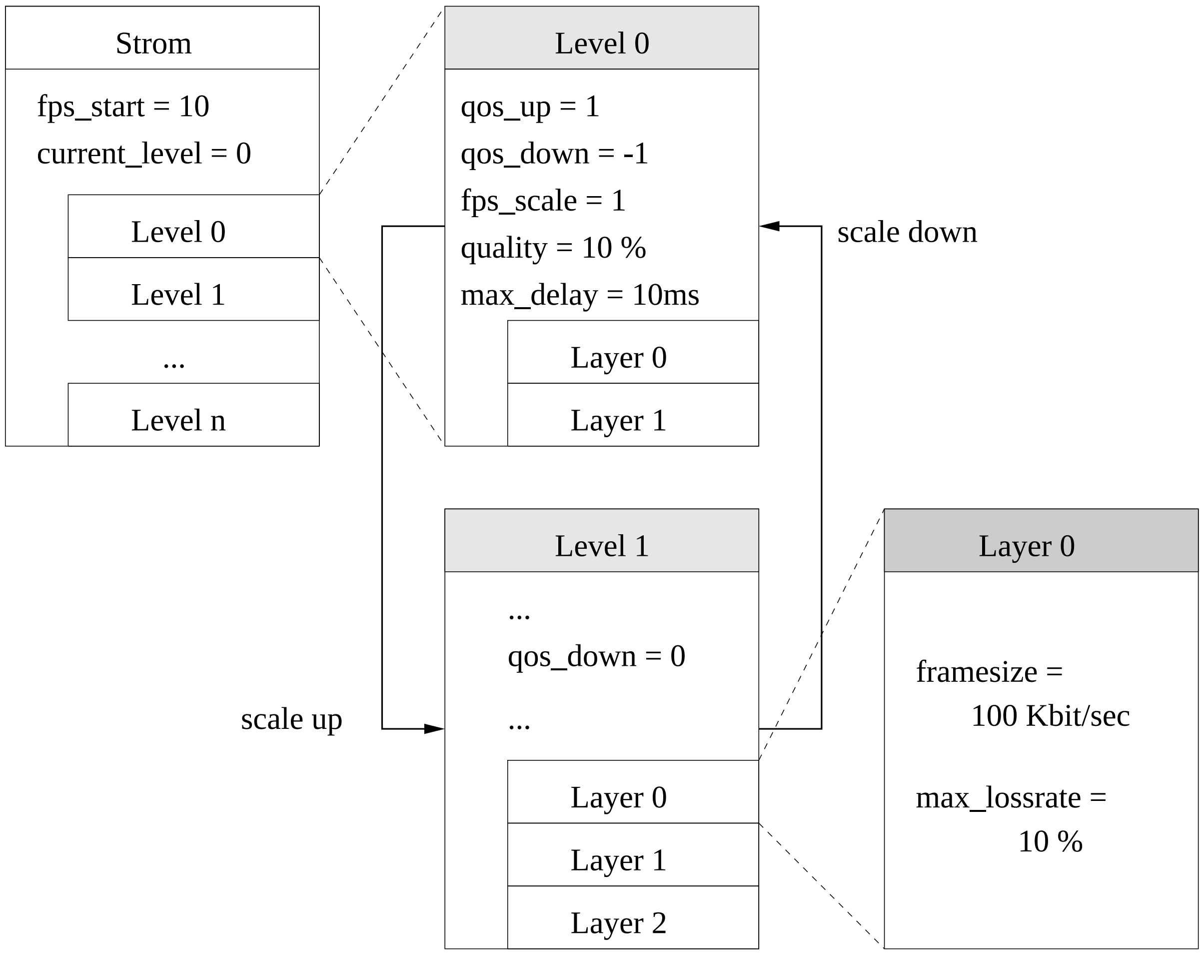


Abbildung 3.16: Statische QoS-Beschreibung eines Stroms

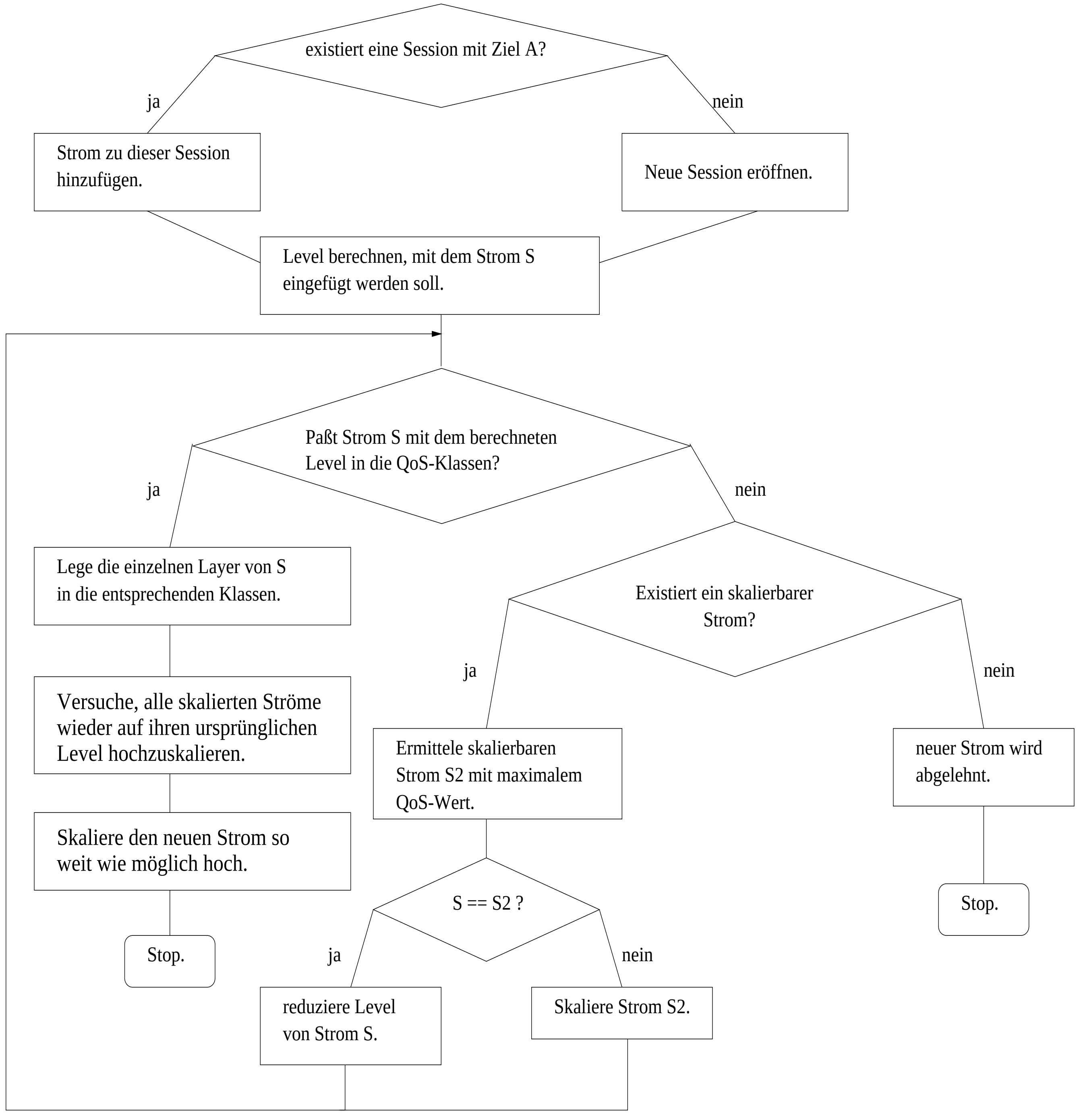


Abbildung 3.17: Neuen Strom S mit Zieladresse A hinzufügen

Level des neuen Stroms ist nun direkt aus der berechneten Qualität abzuleiten. Es wird der niedrigste Level gewählt, der mindestens diese Qualität gewährleistet.
Paßt der Strom nun auf Anhieb in die QoS-Klassen, d.h. daß für jeden Layer des Levels eine Klasse gefunden werden kann, die sowohl genügend freie Bandbreite als auch Delay-Zeiten und Verlustraten bietet, die vom entsprechenden Layer gefordert werden, dann ist der neue Strom erfolgreich in die QoS-Klassen integriert, und der Algorithmus ist beendet. Ansonsten wird versucht, Ressourcen für den neuen Strom frei zu geben, indem bereits bestehende Ströme herunterskaliert werden. Und zwar wird immer der Strom mit dem höchsten aktuellen QoS-Wert skaliert, sofern diese Skalierung überhaupt sinnvoll ist. Nicht sinnvoll ist eine Skalierung dann, wenn der zu skalierende Strom nur Layer in Klassen besitzt, die für den neuen Strom gar nicht in Frage kommen, weil sie die QoS-Anforderungen des neuen Stroms nicht erfüllen. Skaliert wird also ein in diesem Sinne sinnvoll skalierbarer Strom mit der höchsten Qualität. Besitzt der neu einzufügende Strom in dem oben berechneten Level den höchsten Qualitätswert, wird der Level dieses Stroms reduziert. Das führt dann nicht zu mehr verfügbarer Bandbreite, sondern zu weniger benötigter Bandbreite. Würde der neue Strom nicht beim Skalieren berücksichtigt, könnte es passieren, daß andere Ströme in unfairer Art und Weise zurückgedrängt oder sogar ganz verdrängt werden.
Es wird nun solange skaliert, bis entweder der neue Strom eingefügt werden kann oder keine Skalierung mehr möglich ist. Im letzten Fall heißt das, daß der Strom abgelehnt wird.
Es kann nun aber vorkommen, daß Ströme skaliert wurden, deren Skalierung zwar eigentlich sinnvoll aber unnötig war, weil der neue Strom am Ende doch in andere Klassen aufgenommen wurde. Aus diesem Grund wird nach der Skalierungsphase versucht, alle Ströme wieder auf ihren ursprünglichen Level hochzuskalieren. Besonders deutlich wird die Notwendigkeit dieser Maßnahme im Falle der Ablehnung des neuen Stroms. Es wurde dann ja solange skaliert wie möglich, mit dem Ergebnis, daß der neue Strom gar nicht aufgenommen wird. Es ist offensichtlich sinnvoll, in diesem Fall wieder den Ausgangszustand herzustellen.
Ganz am Ende, nach erfolgreicher Aufnahme des neuen Stroms, wird dieser noch so weit wie möglich hochskaliert, um nicht eventuell mit einer unnötig niedrigen Qualität zu senden.

3.4.4 Beenden eines Stroms

Wird ein Strom von der Anwendung beendet, werden seine Layer aus dem QoS-Klassen entfernt. Es wird nicht versucht, andere Ströme hochzuskalieren. Dies hat folgende Gründe: Zum einen ist es natürlich weniger aufwendig, die Layer einfach aus den Klassen zu entfernen, zumal ja spätestens bei der nächsten Neuverteilung das Beenden des Stroms berücksichtigt wird. Die frei werdende Bandbreite kann danach auch gut für neue Ströme verwendet werden. Würden beim Beenden eines Stroms die anderen Ströme sofort hochskaliert und so die freie Bandbreite direkt wieder genutzt, so müßten diese Ströme eventuell wieder herunterskaliert werden, sobald ein neuer Strom angemeldet wird. Dieses Vorgehen würde also einen recht hohen Aufwand bedeuten. Zum anderen könnte es zu einem ständigen Hin- und Her-Skalieren führen, was aber vermieden werden soll. Es scheint bei multimedialen Datenströmen sinnvoller zu sein, eine Weile in niedriger Qualität zu senden, als ständig die Qualität zu ändern. Man stelle sich nur eine Videoübertragung mit ständig wechselnder Auflösung vor.

3.4.5 Hochskalieren eines Stroms

Soll ein Strom um einen Level hochskaliert werden, muß zuerst bestimmt werden, in welchen Level eigentlich skaliert werden soll. Ein Strom, der noch nicht sendet, wird beim Hochskalieren in Level 0 versetzt. Ansonsten wird der neue Level aus der statischen QoS-Beschreibung des Stroms ermittelt (QoS-Up-Wert des alten Level). Sendet der Strom bereits mit dem höchst möglichen Level, schlägt der Skalierungsvorgang fehl. Andernfalls werden alle Layer des alten Levels aus ihren Klassen genommen, und es wird versucht, die Layer des neuen Layers in den Klassen unterzubringen. Wie auch beim Herunterskalieren spielen hier nicht nur die noch freie Bandbreite der Klassen, sondern auch deren Eigenschaften in Bezug auf Paketverluste und Delay-Zeiten eine Rolle. Ein Layer paßt nur dann in eine QoS-Klasse, wenn diese Klasse alle seine QoS-Anforderungen erfüllt. Delay-Zeiten und Paketverluste sind aber keine globalen Eigenschaften einer Klasse, sondern sie sind auch an die jeweilige Session gebunden, da diese Eigenschaften stark von der Route durch das Internet und somit auch von der Zieladresse abhängen. In der in Abschnitt 4.7 beschriebenen Messung erkennt man, wie der QoS-Manager auf eine zu hohe Ende-zu-Ende-Verzögerung reagiert: Wenn das Delay für einen Transportstrom zu groß wird, wird dieser Strom bei der nächsten Neuverteilung aus seiner Klasse herausgenommen, und es wird eine andere Klasse gesucht, die den Anforderungen des Stroms gerecht wird.
Findet nun jeder Layer des neuen Levels eine Klasse, die seinen Anforderungen genügt, so wird das Skalieren erfolgreich abgeschlossen. Paßt aber nur ein einziger Layer in keine der Klassen, ist das Hochskalieren nicht möglich, und es müssen alle anderen Layer des neuen Levels wieder entfernt werden. Die Layer des alten Levels werden dann in ihre ursprünglichen Klassen zurückgelegt, womit der Ausgangszustand wieder hergestellt ist.
Abbildung 3.18 veranschaulicht diesen Vorgang anhand eines Flußdiagramms.

3.4.6 Herunterskalieren eines Stroms

Im Gegensatz zum Hochskalieren werden beim Herunterskalieren eines Stroms alle Layer in ihren Klassen belassen. Zuerst wird der Level zu dem skaliert werden soll, also der QoS-Down Wert des aktuellen Levels, aus der statischen QoS-Beschreibung des Stroms gelesen. Wenn der Strom herunterskaliert werden kann (Qos-Down$\neq$ -1), werden die Layer des alten Levels aus ihren Klassen entfernt und die Layer des neuen Levels in dieselben Klassen gelegt. Es darf nicht vorkommen, daß ein niedrigerer Level mehr Layer besitzt als der höhere oder die einzelnen Layer beim Runterskalieren größer werden, da sonst das Skalieren fehlschlagen könnte. Wenn sich die Anzahl der Layer erhöht, ist unklar, in welche Klasse neue Layer eingefügt werden sollen und ob diese Layer überhaupt in eine Klasse eingefügt werden können. Erhöht sich für einen Layer beim Herunterskalieren die Bandbreite, paßt dieser eventuell nicht mehr in seine vorherige QoS-Klasse, und es kann eventuell auch keine andere Klasse für ihn gefunden werden.
Das Herunterskalieren eines Stroms soll aber immer möglich sein, sofern der Strom nicht schon mit niedrigster Qualität sendet, da so auf Netzüberlastung und die damit verbundenen Paketverluste reagiert wird, und außerdem ein neu hinzukommender Strom ein Skalieren notwendig machen kann (s. Abschnitt "Hinzufügen eines Stroms").

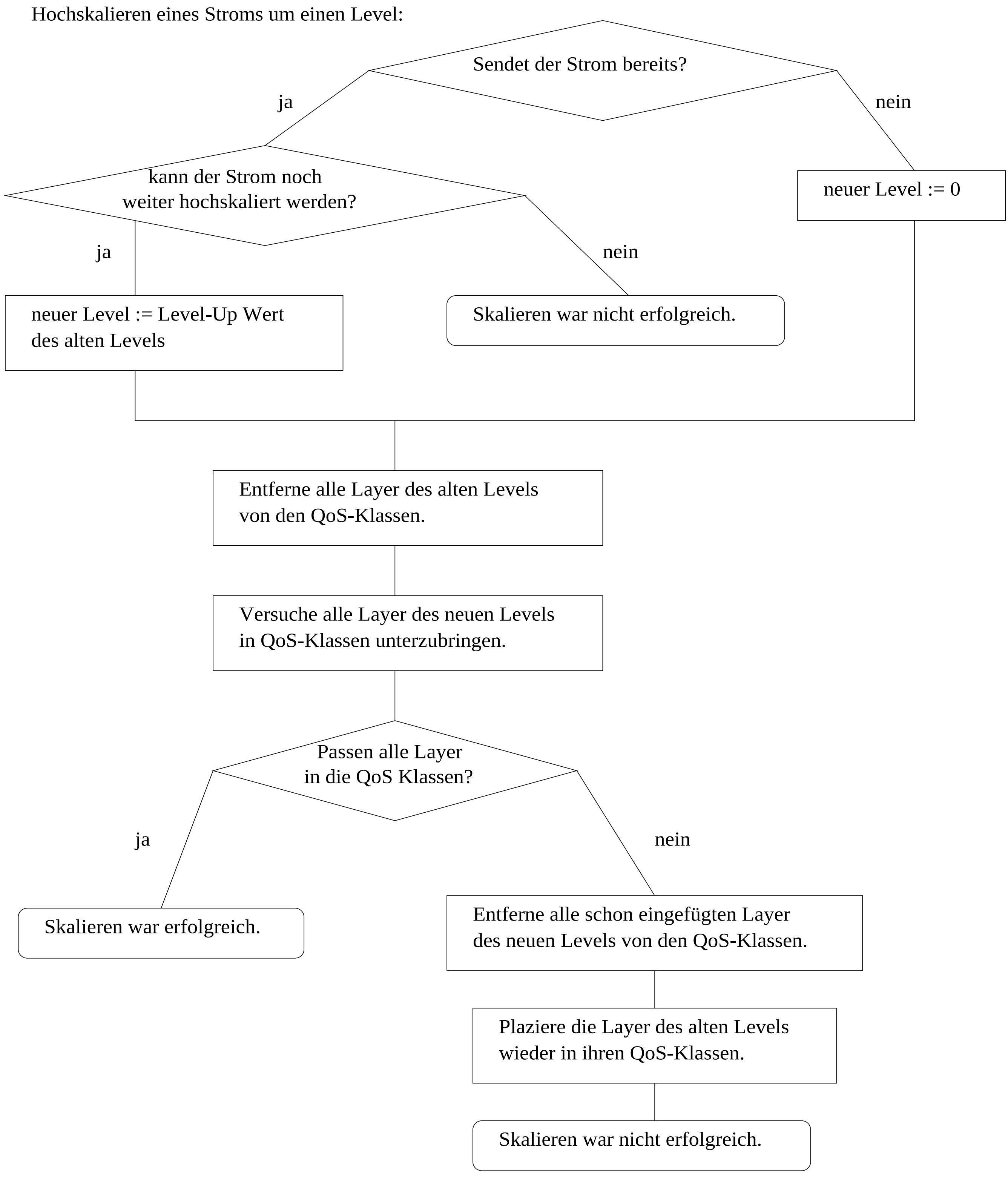


Abbildung 3.18: Hochskalieren um einen Level

3.4.7 Die Neuverteilung

Bei der Neuverteilung alle N Sekunden (momentan ist N=10) werden anfangs alle Ströme aus ihren Klassen genommen. Die Neuverteilung läuft dann in zwei Phasen ab: zuerst werden alle Ströme mit ihrem niedrigsten Level auf die Klassen verteilt, um die Minimalanforderungen jedes Stroms zu gewährleisten. Danach werden die Ströme solange wie möglich hochskaliert, wobei versucht wird, immer den Strom mit der momentan geringsten Qualität zu skalieren. Die Verteilung ist dann beendet, wenn entweder alle Ströme mit maximalem Level senden, oder kein weiterer Strom mehr hochskaliert werden kann. Ein Strom kann dann nicht weiter hochskaliert werden, wenn keine der Klassen seine QoS-Anforderungen erfüllt. Das kann abgesehen von fehlender Bandbreite in der Klasse zu hohe Verlustraten oder zu hohes Ende-zu-Ende-Delay der Klasse bedeuten. Erfüllen mehrere Klassen die Anforderungen, so wird von diesen die niedrigste ausgewählt.

3.4.8 Reaktion auf Paketverluste und variierende Ende-zu-Ende-Verzögerungen

Paketverluste und Delay-Zeiten sind keine globale Eigenschaften einer QoS-Klasse, sondern sie unterscheiden sich auch innerhalb einer Klasse zwischen den verschiedenen Sessions, die Ströme in dieser Klasse besitzen. Es handelt sich also um Eigenschaften auf Macroflow- und Klassen-Ebene. Bei Transportströmen, also Layern, die innerhalb einer Klasse liegen und zu derselben Session gehören, geht man davon aus, daß sie ähnliche Bedingungen im Netz vorfinden.
Der QoS-Manager reagiert auf zweierlei Weisen auf Paketverluste: Zum einen wird ein Strom herunterskaliert, für den vom Transportmodul Paketverluste gemeldet werden, die mindestens die maximal aktzeptierbare Verlustrate eines seiner Layer überschreiten. Bei jeder Neuverteilung wird dann wieder versucht, einen Level hochzuskalieren. Zum anderen geht jede gemeldete Verlustrate in einen Alterungswert ein, der bei der nächsten Neuverteilung berücksichtigt wird. Liegt dann dieser Wert über dem maximal aktzeptierbaren Wert eines Layers, wird dieser Layer auf keinen Fall in die entsprechende Klasse eingefügt. Wird eine neue Verlustrate gemessen, dann wird der Alterungswert folgendermaßen berechnet:

$$Verlustrate_{neu} = (1-\alpha) * Verlustrate_{alt} + \alpha * Verlustrate_{gemessen}\ , \alpha = \frac{1}{2}$$

Wenn dieser Alterungswert aber einmal einen Wert annimmt, der keinem Strom einer bestimmten Session mehr erlaubt, einen seiner Layer in dieser Klasse zu plazieren, können für diese Session auch keine neuen Verlustraten mehr gemessen werden. Um zu verhindern, daß der Alterungswert dann dauerhaft auf diesem hohen Niveau bleibt, wird er nach jeder Neuverteilung auf Null zurückgesetzt. Der erste gemessene Wert nach einer Neuverteilung muß dann aber natürlich voll berücksichtigt werden. D.h. in diesem speziellen Fall nimmt α dann den Wert 1 an.
Ebenfalls über eine Alterungsfunktion gehen die Delay-Zeiten in die Neuverteilung ein. Für jede Session schickt der QoS-Manager ICMP-Pings über jede QoS-Klasse an den entsprechenden Zielrechner und mißt die Zeit zwischen Senden des ICMP-Packetes und dem Empfang der Antwort des Client. Diese Round-Trip-Zeit ergibt dann durch zwei geteilt die geschätzte Ende-zu-Ende Verzögerung zum Empfänger.
Geht man davon aus, daß die ICMP-Pakete nur auf ihrem Weg zum Client über die entsprechenden Klassen gesendet werden und die Antwort-Pakete des Empfängers zum Beispiel alle über

Best-Effort, so sollte man diese Asymmetrie in der Berechnung berücksichtigen, indem man die gemessene RTT nicht einfach durch zwei teilt sondern nach folgender Formel berechnet:

$$d_{Klasse_i} = RTT_{Klasse_i} - \frac{RTT_{BE}}{2},$$

wobei d_{Klasse_i} für das geschätzte Ende-zu-Ende-Delay der Klasse i in der entsprechenden Session, RTT_{Klasse_i} für die in dieser Klasse gemessene Round-Trip-Zeit und RTT_{BE} für die gemessene Best-Effort Round-Trip-Zeit steht. Dies wird allerdings in der aktuellen Version des QoS-Managers noch nicht berücksichtigt.

Kapitel 4

Die Messungen

In diesem Kapitel werden die Messungen beschrieben. Das DiffServ-Szenario ist in Abbildung 3.10 zu sehen. Im ersten Teil wird die Funktionalität der DiffServ-Router durch Messungen von Datenraten, Roundtripzeiten und Jitter dargestellt, darauf folgen Messungen zur Funktionalität und TCP-Freundlichkeit des QoS-Managers.

Anmerkung: Das SLA bei allen Messungen wurde mit Hilfe des TC-Programmes eingestellt. Die mit diesem Programm allokierten Bandbreiten unterscheiden sich geringfügig von den Bandbreiten des SLA. Der Grund dafür ist, daß das TC-Programm für die Allokation der Bandbreite die internen Tabellen aus dem Kernel verwendet. Deshalb ist eine bitgenaue Allokation unmöglich. Dieser Fehler ist aber relativ klein.

4.1 Borderrouter-Funktionalität

von Jan Selzer

In diesem Test wurde die Funktionalität des Border Routers geprüft. Es wurden die Ströme durch alle DS-Klassen vom Server (*corona*) zum Client 1 (*detmolder*) gesendet. Das SLA des Borderrouters und die Senderaten kann man der Tabelle 4.2 entnehmen. Man will mit dem Test das Verwerfen der Pakete für alle Klassen im Falle der Überschreitung der jeweiligen Bandbreite zeigen. Die Abbildung 4.1 zeigt die gemessenen Senderaten der Ströme auf dem Server corona (wie in Tabelle 4.2).

Die Abbildung 4.2 zeigt die Empfangsraten auf dem Client 1 (*detmolder*) Diese stimmen mit dem SLA des Borderoruters überein. Dies bedeutet, daß alle Pakete, welche über der im SLA

DiffServ-Klasse	SLA für Borderrouter (MBit, Byte)	Senderaten auf *corona* (Mbit, Byte)
EF	2 MBit (250000)	3 MBit (375000)
AF21	2 MBit (250000)	3 MBit (375000)
AF11	2 MBit (250000)	3 MBit (375000)
BE	4 MBit (500000)	5 MBit (625000)

Tabelle 4.1: Konfiguration für Messung 1

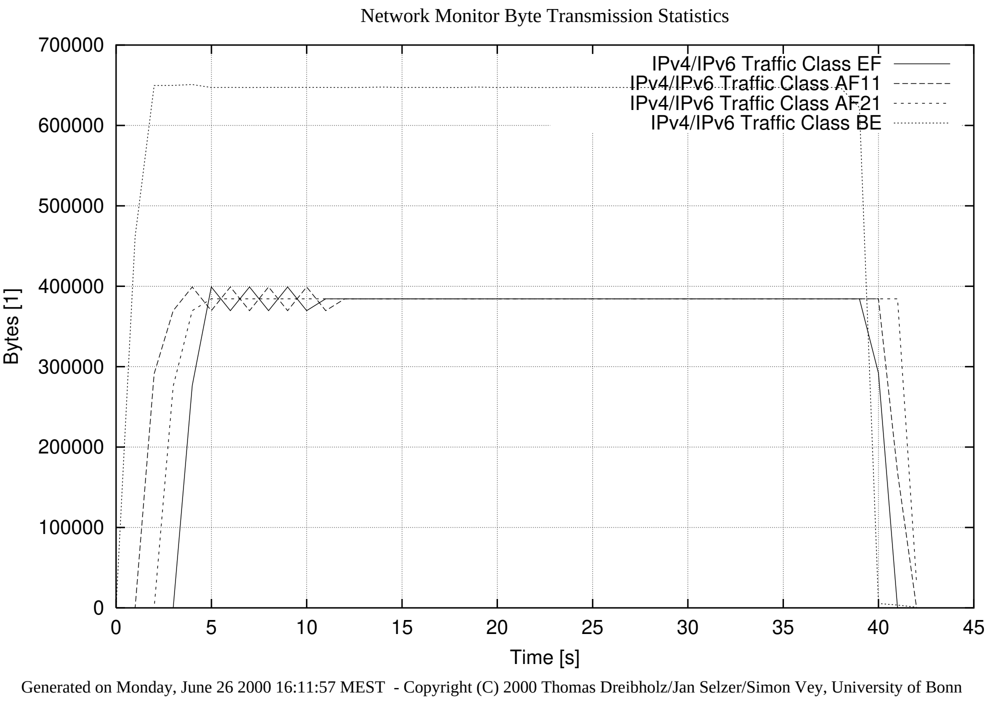


Abbildung 4.1: Borderrouter-Funktionalität, Messung auf corona

DiffServ-Klasse	SLA für Core-Router (MBit, Byte)	Senderaten auf *amstel* (Mbit, Byte)
AF21	3 MBit (375000)	4 MBit (500000)
BE	5 MBit (625000)	6 MBit (750000)

Tabelle 4.2: Konfiguration für Messung 2

eingestellen Bandbreite verschickt wurden, auf dem Borderrouter (*holsten*) verworfen wurden. Dies beweist die Funktionsfähigkeit des Borderrouters.

4.2 Roundtripmessung für zwei Klassen

VON JAN SELZER

Diese Messung zeigt den Unterschied der Rount-Trip-Werte zwischen BE und AF-Klassen. Es wurden dafür 2 Ströme vom Hintergrundlast-Sender (*amstel*) zum Hintergrundlast-Empfänger (*gaffel*) über den Core-Router (*holsten*) gesendet. Die Senderaten sowie das SLA auf dem Core-Router kann man der Tabelle 4.2entnehmen.

Man kann den deutlichen Unterschied zwischen den beiden Round-Trip-Werten in der Abbildung 4.3 sehen. Die Roundtripzeiten für BE sind wesentlich höher und instabiler als für die AF-Klasse. Daraus ergibt sich eine bessere Zuverlässigkeit der AF-Klasse, was der Definition der AF-Klasse entspricht.

Es wird jeweils 1 MBit über die allozierten Bandbreiten gesendet. In der Abbildung 4.5 sieht man, wie der Empfangsraten schwanken. Dies hängt mit der Implementierung der BE- und AF-Queues zusammen: Es werden mehr Daten als im SLA vereinbart durchgelassen. Die Größe der

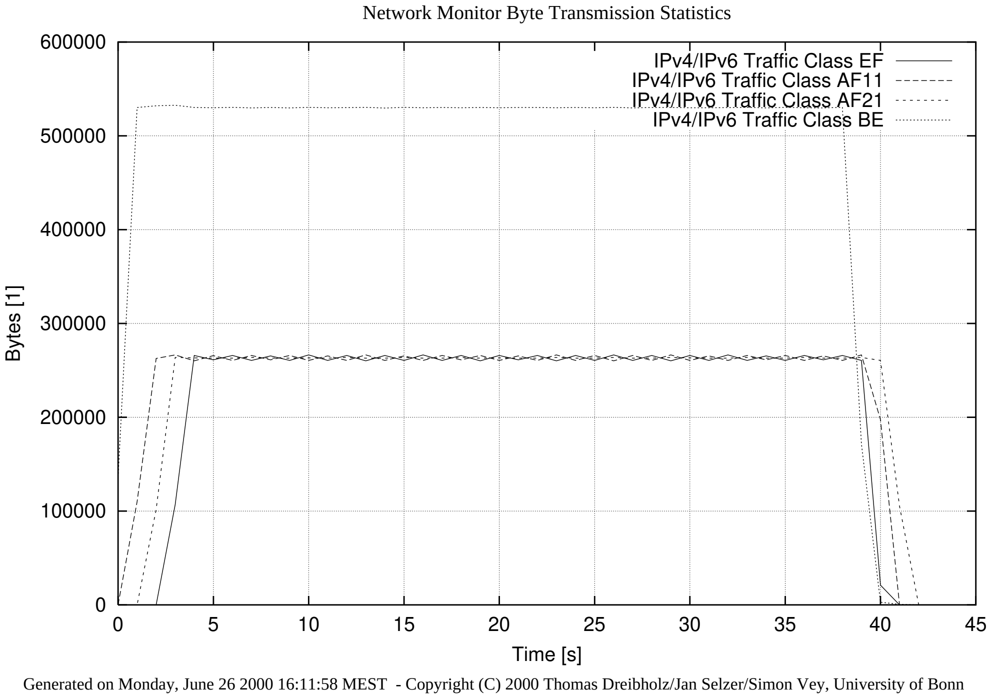


Abbildung 4.2: Borderrouter-Funktionalität, Messung auf detmolder

DiffServ-Klasse	SLA für Core-Router (MBit, Byte)	Senderaten auf *amstel* (Mbit, Byte)
EF	3 MBit (375000)	2.5 MBit (312500)
AF21	4 MBit (500000)	3.5 MBit (437500)
AF11	4 MBit (500000)	3.5 MBit (437500)
BE	5 MBit (625000)	8 MBit (1000000)

Tabelle 4.3: Konfiguration für Messung 3

Überschreitung der allozierten Bandbreite hängt vom Limit-Parameter des RED-Mechanismus ab. In unserem Beispiel war es 60 KByte. Die Schwankungen in der Abbildung 4.5 ergeben sich aus den Paketverlusten, da wir mehr als im SLA vereinbart gesendet haben.

4.3 Roundtripmessung für alle Klassen in einem realistischem Szenario

von Jan Selzer

In der folgenden Messung wird das Verhalten der DS-Ströme in der DS-Domain untersucht. Hier wird der Strom vom Hintergrundlast-Sender (*amstel*) zum Hintergrundlast-Empfänger (*gaffel*) über den Core-Router (*holsten*) gesendet. Abbldung 4.6 zeigt die gemessenen Roundtripzeiten für alle DS-Klassen. Das SLA des Core-Routers und die Senderaten kann man der Tabelle 4.4 entnehmen. Nur BE überschreitet die Bandbreite des SLA.

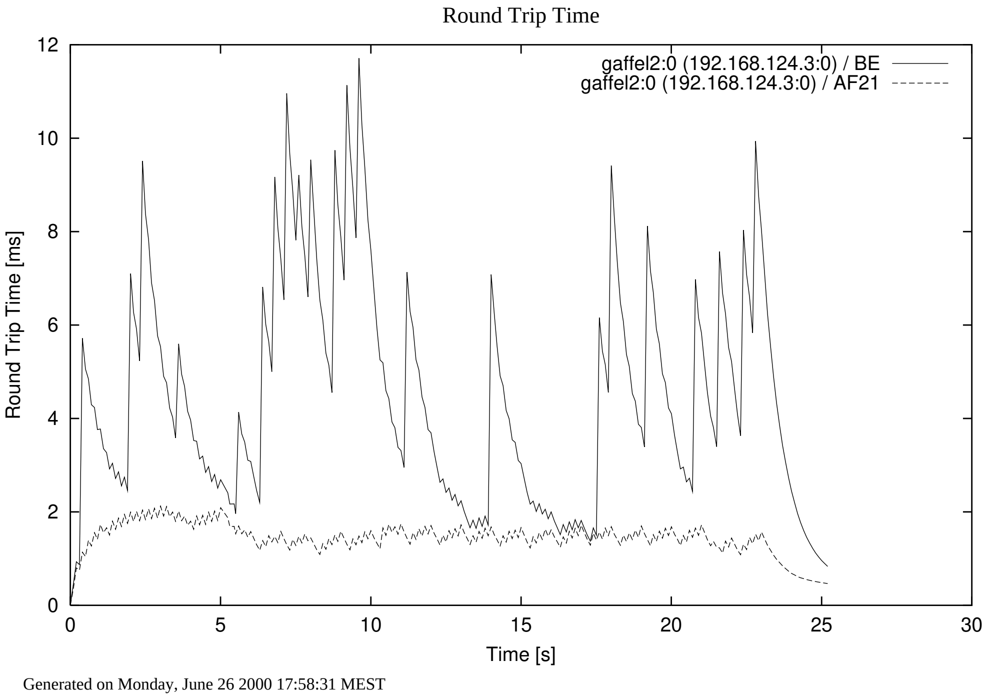


Abbildung 4.3: Roundtripzeiten

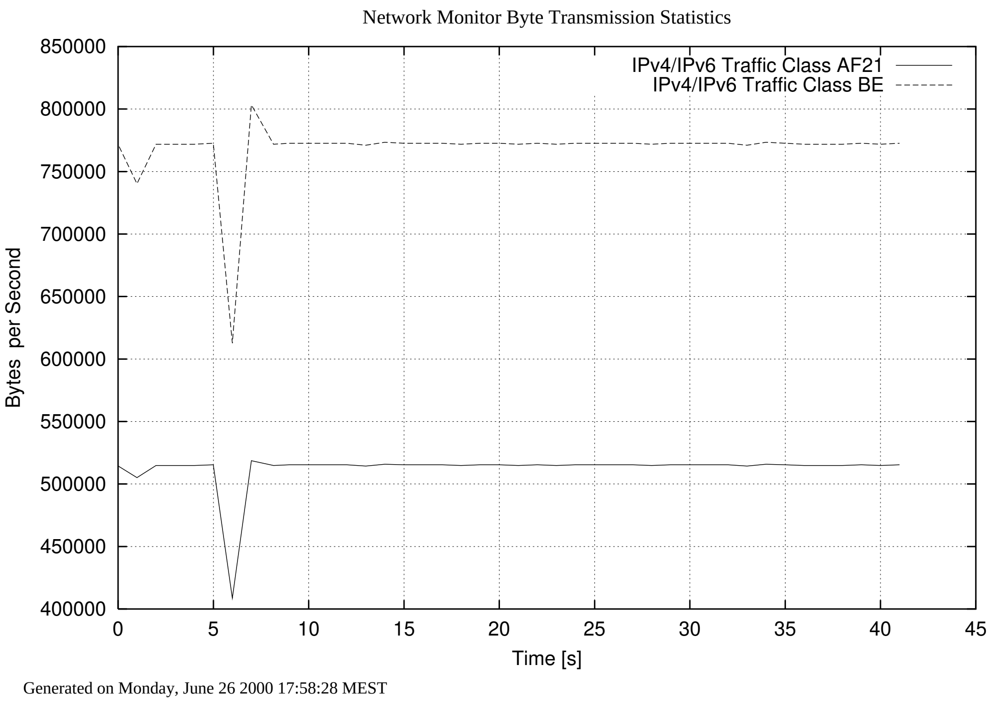


Abbildung 4.4: Hintergrundlast-Sender

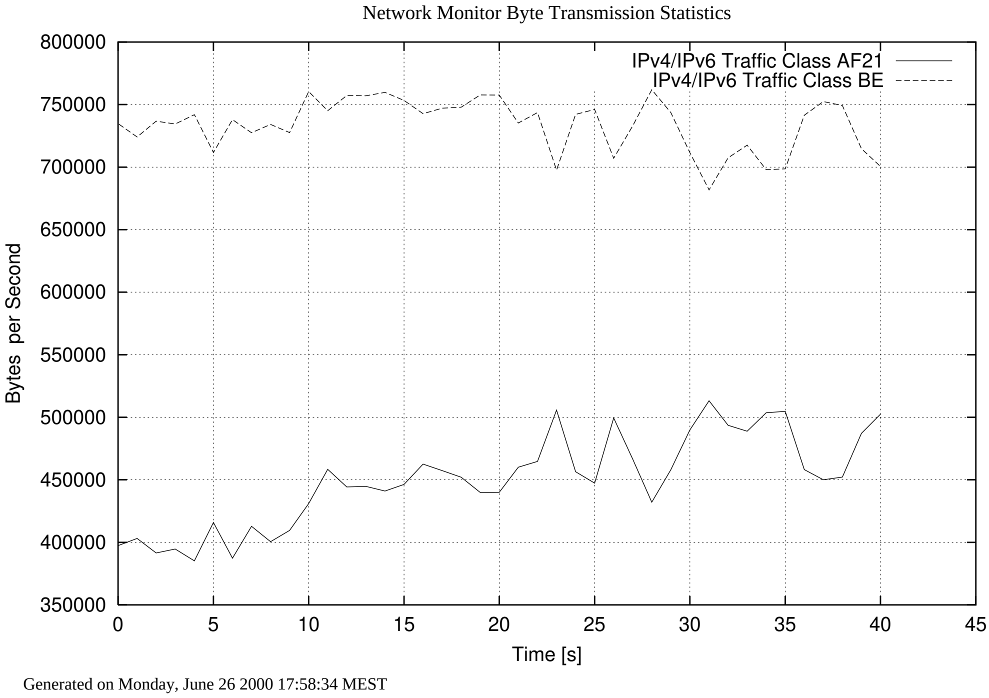


Abbildung 4.5: Hintergrundlast-Empfänger

Man sieht, daß die EF-Pakete die besten Roundtripzeiten haben. Dann kommen die AF Klassen und anschließend die BE-Klasse mit den längsten Roundtripzeiten. Dieses Verhalten der verschiedenen DS-Ströme wird erwartet, wenn die Anwendung ihre Ströme durch verschiedene Klassen, abhängig von der Wichtigkeit der Daten, sendet.

4.4 Jitter-Messung

von Jan Selzer

Die Messung des Jitters vergleicht alle DS-Klassen in einem für eine DiffServ-Domain realistischen Szenario: Es werden vom Störsender zum Störempfänger 3 Ströme über EF, AF11, AF21 Klassen gesendet, und über die BE-Klasse wird TCP verschickt. Die Einstellungen für diese Messung sind in Tabelle 4.4 zu finden. Nur BE überschreitet die Bandbreite des SLA.

Die Jitter-Werte kann man den Abbildungen 4.7, 4.8, 4.9 und 4.10 entnehmen. Aus den Bildern kann man den Schluß ziehen, daß die AF und EF Jitter relativ gleich um 250 Mikrosek. liegen. Im Gegensatz dazu liegt der Jitter von BE bei 10000 Mikrosek.. Also wird bei diesem Szenario, wenn die Benutzer von EF und AF Klassen ihre Bandbreiten nicht überschreiten, eine gute und zuverlässige Verbindung hergestellt. EF und AF haben kleine Jitter-Werte und werden ohne Verluste geliefert.

Die Schwankungen können dadurch erklärt werden, daß die Pakete verworfen werden. Bei Verlusten steigt dann der Jitter.

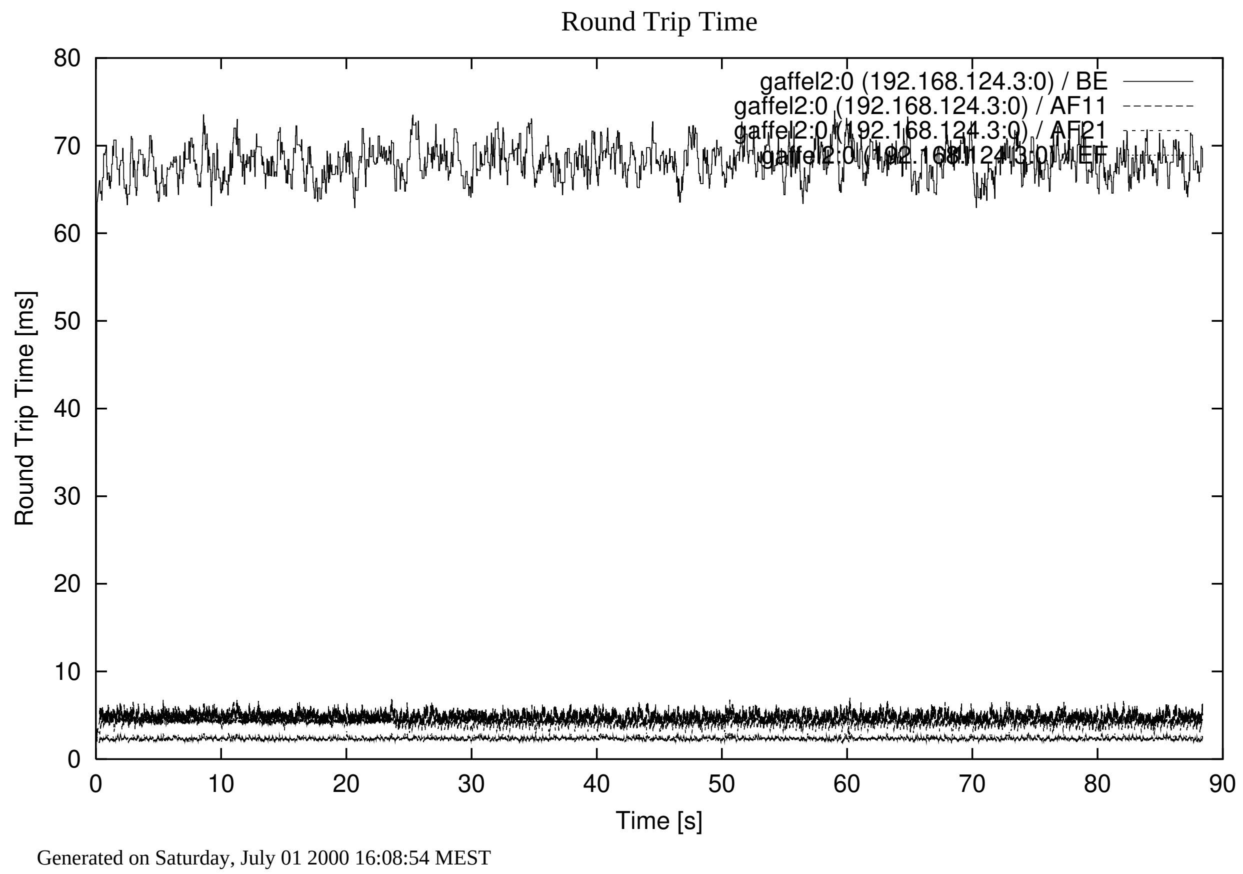


Abbildung 4.6: Roundtripzeiten

DiffServ-Klasse	SLA für Core-Router (MBit, Byte)	Senderaten auf *amstel* (Mbit, Byte)
EF	2.8 MBit (350000)	1.5 MBit (187500)
AF21	1.8 MBit (225000)	1.5 MBit (187500)
AF11	1.8 MBit (225000)	1.5 MBit (187500)
BE	1.8 MBit (225000)	TCP-Strom (max. Bandbreite)

Tabelle 4.4: Konfiguration für Messung 4

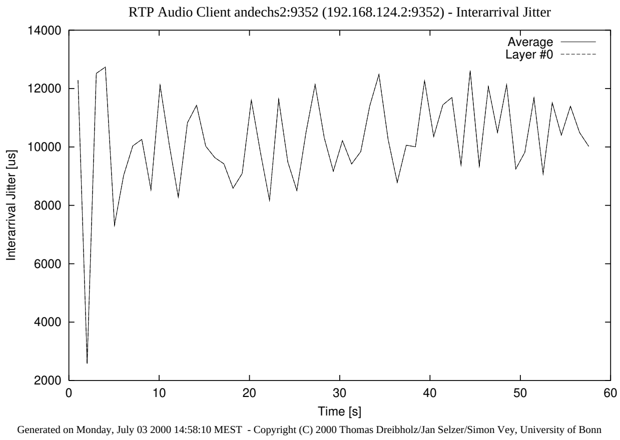


Abbildung 4.7: Client 1: BE

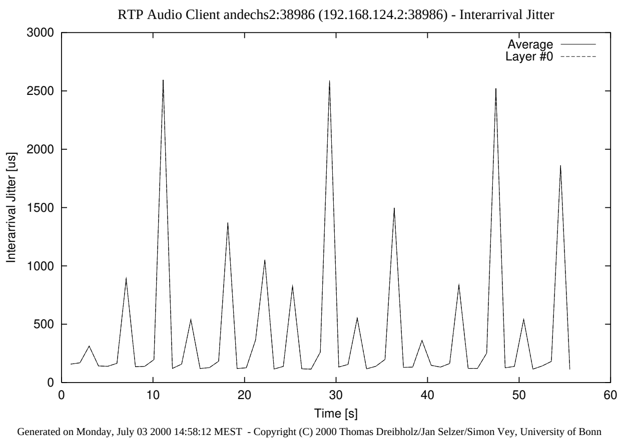


Abbildung 4.8: Client 2: EF

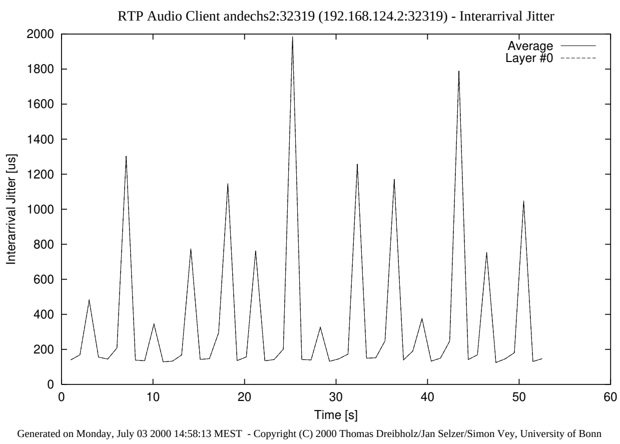


Abbildung 4.9: Client 3: AF11

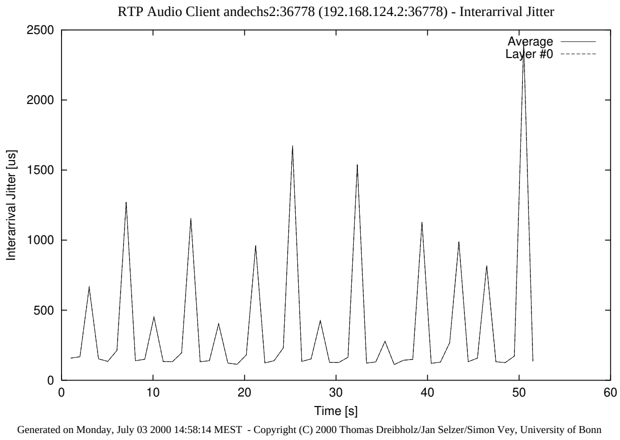


Abbildung 4.10: Client 4: AF21

4.5 Skalierungen und Neuverteilung

von Simon Vey

Bei dieser Messung wurde dem QoS-Manager ein SLA von je zwei Mbit/s in den Klassen EF, AF21 und AF11 übergeben. Best Effort wurde also bei der Verteilung der Bandbreite nicht berücksichtigt. Es wurden nacheinander vier Audioströme gestartet. Die Datenraten der einzelnen Ströme wurden direkt hinter dem Server gemessen. In Abbildung 4.11 sieht man die Meßergebnisse der Ströme, wobei für jeden Strom dessen Durchsätze in den einzelnen Service-Klassen dargestellt werden.

Da ein Audiostrom bei höchster Qualität eine Datenrate von ca. 1,5 Mbit/s beansprucht, paßt der erste Strom anfangs komplett in AF11. Der Strom wird in AF11 plaziert, da dies die niedrigste Klasse ist, die die Anforderungen des Stroms erfüllt (Paketverluste und zu hohe Round-Trip-Zeiten spielten bei dieser Messung keine Rolle). Nach ca. 18 Sekunden wird der zweite Audiostrom gestartet. Ein Layer dieses Stroms paßt mit knapp 0,4 Mbit noch in AF11, die anderen beiden Layer werden anfangs über AF21 gesendet. Nach etwa 25 Sekunden findet eine Neuverteilung statt, und man erkennt, daß nun beide Ströme ihre Bandbreite jeweils gleichmäßig auf AF11 und AF21 aufteilen. Zum Zeitpunkt t=30 Sekunden wird der dritte Strom gestartet. In den beiden AF-Klassen ist jeweils noch Bandbreite für einen kleinen Layer des neuen Stroms übrig. Der große Layer (ca. 0,75 Mbit/s) des neuen Stroms wird über EF gesendet. Strom vier, der nach ca. 42 Sekunden gestartet wird, kann nun nicht mehr mit höchster Qualität in die Klassen aufgenommen werden, da nicht mehr genügend Bandbreite in den Service-Klassen vorhanden ist. Dieser Strom sendet also mit niedrigerer Qualität mit einer Bandbreite von ca. 1,2 Mbit in EF. Bei der nächsten Neuverteilung (bei 45 Sekunden) wird die Bandbreite nun fairer zwischen den Strömen aufgeteilt: Die Ströme 1 und 2 werden in ihren Klassen herunterskaliert, Strom 3 nimmt seinen Layer aus AF11 heraus und sendet nun über EF und AF21. Auf diese Weise wurde in AF11 und EF genug Bandbreite freigemacht, so daß Strom 4 jeweils mit knapp 0,7 Mbit in diesen Klassen senden kann. Nachdem Strom 1 nach ca. 85 Sekunden beendet wurde, ist wieder genügend Bandbreite für die volle Qualität der verbleibenden drei Ströme vorhanden. Dementsprechend bekommen die drei Ströme bei der nächsten Neuverteilung wieder ihre volle Bandbreite zugewiesen.

Die Messung verdeutlicht, wie die zur Verfügung stehende Bandbreite unter den Strömen aufgeteilt

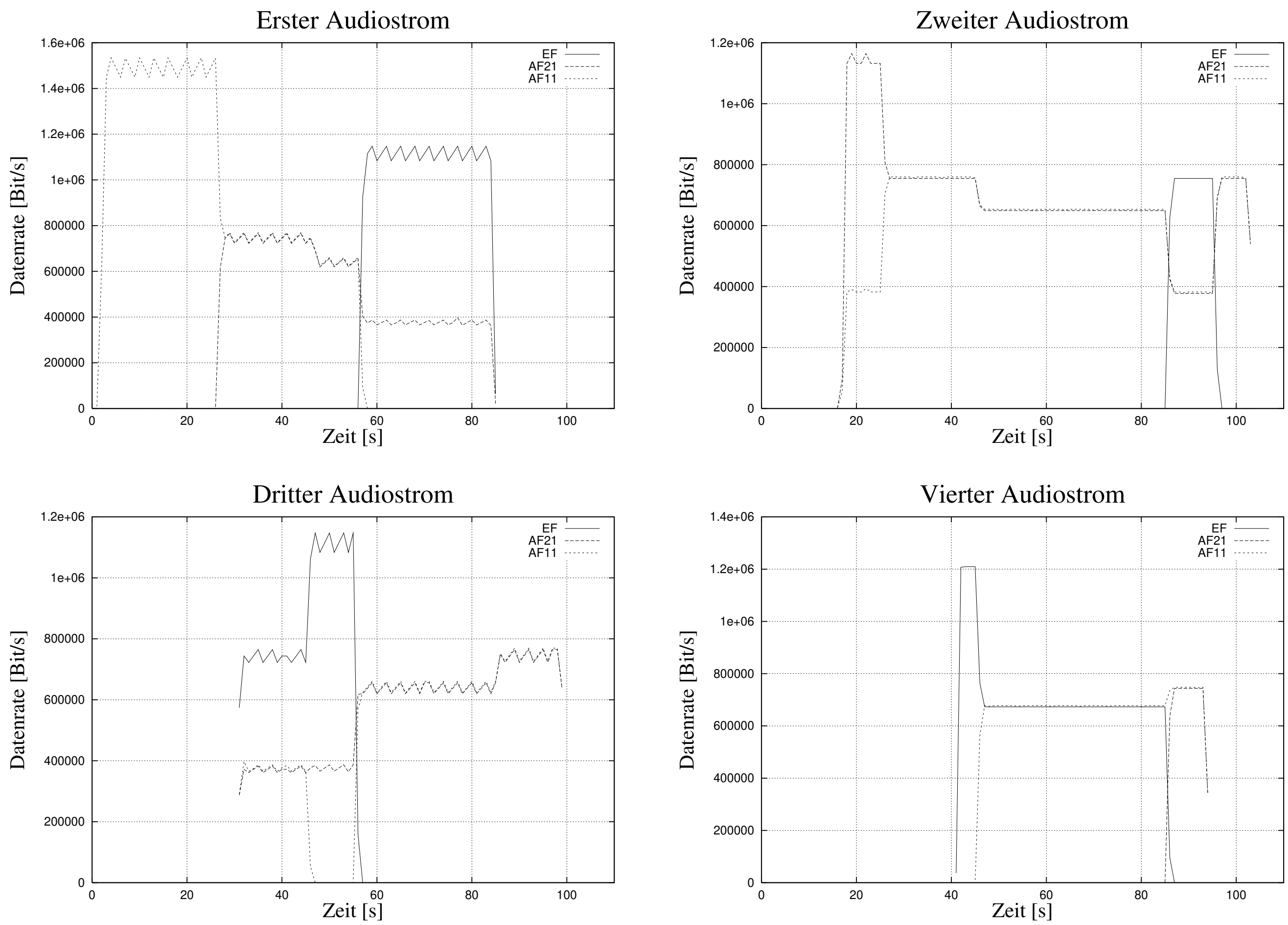


Abbildung 4.11: Vier Audioströme, direkt hinter dem Server gemessen.

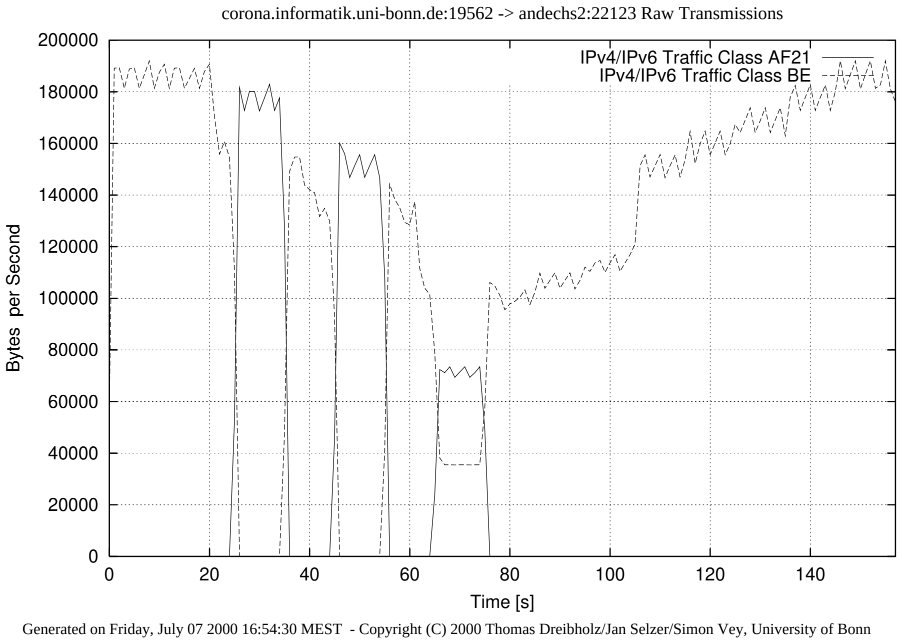


Abbildung 4.12: Skalierungen und Klassenwechsel aufgrund von Paketverlusten (gemessen hinter dem Core-Router).

wird. Wenn nicht genügend Bandbreite für die optimale Qualität aller Ströme vorhanden ist, wird durch Skalierungen und Neuverteilungen eine faire Aufteilung erreicht.

4.6 Paketverluste

von Simon Vey

Um festzustellen, wie der QoS-Manager auf Paketverluste reagiert, wurde ein Audiostrom von dem Server Corona zum Client Andechs gestartet, der anfangs vollständig über Best Effort sendet. Das SLA im Border-Router ist auf jeweils 2 Mbit/s für EF, AF21 und AF11 und auf 4 MBit/s für Best Effort eingestellt. Die Reaktion auf zu hohe Ende-zu-Ende-Verzögerungen wurde bei dieser Messung deaktiviert. Nach ca. 20 Sekunden wurde ein UDP-Störsender mit 4 Mbit/s aktiviert. Da im internen Router Holsten ein SLA von 5 Mbit/s für Best Effort eingestellt ist und Audiostrom und Störsender zusammen eine Bandbreite von ca. 5,5 Mbit/s haben, werden Pakete im Router verworfen. Es treten also Paketverluste auf. In Abbildung 4.12 erkennt man, daß unmittelbar nach Starten des Störsenders (Abbildung 4.13) der Durchsatz des Audiostroms durch Holsten geringer wird. Der Audiostrom ist so eingestellt, daß er nach Paketverlusten von mindestens 5% in Layer 0 herunterskaliert. Bei der nächsten Neuverteilung (t=25s) versucht der QoS-Manager Service-Klassen zu finden, die den Anforderungen des Audiostroms gerecht werden. Da in Best-Effort aber starke Paketverluste aufgetreten sind, ist diese Klasse für den Strom nicht mehr ausreichend. Der Strom wird jetzt über AF21 gesendet, und sofort hat der Störsender einen höheren Durchsatz in BE. Bei der nächsten Neuverteilung wird der Audiostrom wieder nach BE gelegt, da hier im letzten Neuverteilungsintervall keine Verluste gemessen wurden (es sendete ja kein Strom über

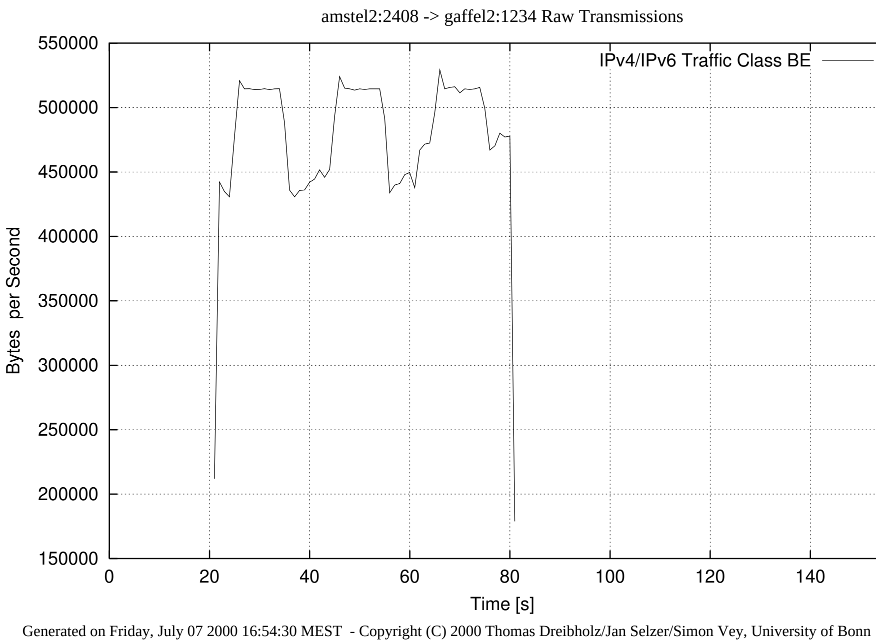


Abbildung 4.13: Durchsatz des Störsenders.

BE). Auf diese Art und Weise wechselt der Strom immer zwischen diesen beiden Klassen und wird, während er über BE sendet, immer weiter herunterskaliert. Nach ca. 65 Sekunden tritt der Fall auf, daß die Verlustraten für Layer 0 in BE wieder zu hoch sind, nicht aber für einen anderen Layer des Stroms, da dieser höhere Verlustraten akzeptiert. Für ein Neuverteilungsintervall wird der Strom über zwei Klassen gesendet.
Nach ca. 80 Sekunden wird der Störsender abgeschaltet. Der Audiostrom muß nun nicht mehr in andere Klassen ausweichen, und er skaliert langsam wieder hoch (ein Level nach jeder Neuverteilung).
Diese Messung zeigt also, daß der QoS-Manager auf Paketverluste reagiert, indem er die betroffenen Ströme herunterskaliert und bei einer Neuverteilung gegebenenfalls die Layer in andere Klassen legt. Nachdem der Störsender abgeschaltet wurde, erholt sich der Audiostrom jedoch nur langsam, da er bei jeder Neuverteilung nur um einen Level hochskaliert wird. Hier wäre eventuell eine andere Vorgehensweise sinnvoll, wobei man jedoch auch nicht zu schnell hochskalieren sollte, um ein ständiges Hin- und Herskalieren zu vermeiden.

4.7 Reaktion auf schwankende Ende-zu-Ende-Verzögerungen

VON SIMON VEY

Der QoS-Manager achtet nicht nur auf Paketverluste, sondern auch auf die aktuellen Delay-Zeiten in den Klassen. Wie bei der letzten Messung, wird auch bei dieser Messung ein Audiostrom über BE von Corona nach Andechs gesendet und dann ein Störsender mit 4 Mbit/s über BE gestartet. Die SLA's entsprechen denen aus der vorherigen Messung. Während bei der letzten Messung aber die Reaktion auf zu hohe Delay-Zeiten im QoS-Manager deaktiviert war, so ist hier

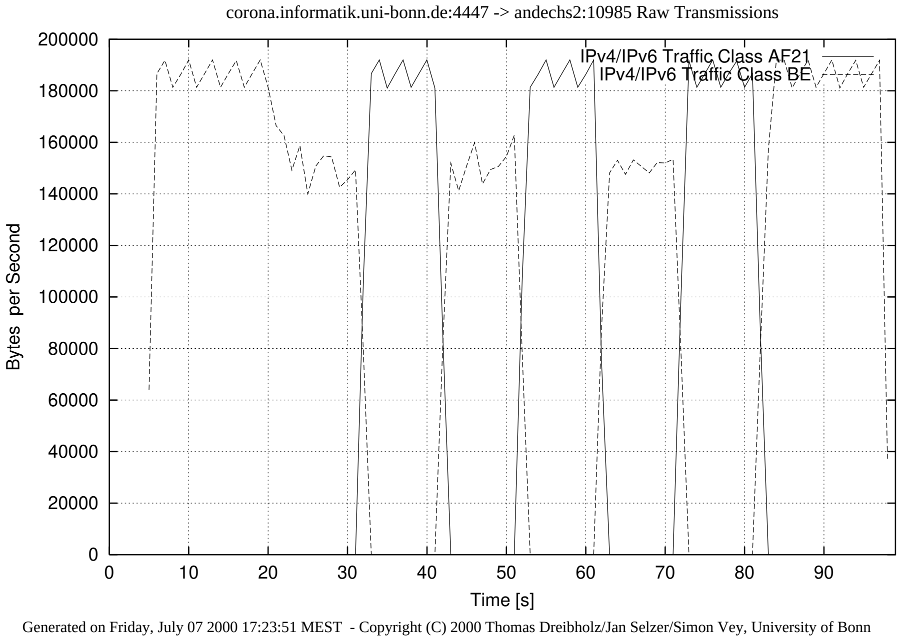


Abbildung 4.14: Klassenwechsel aufgrund hoher Delays.

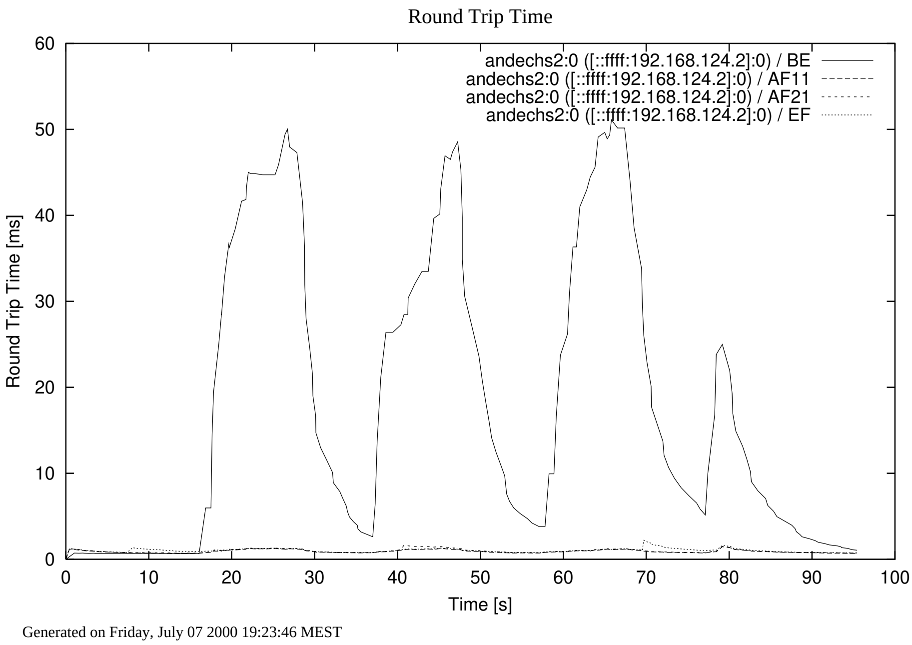


Abbildung 4.15: Round-Trip-Zeiten zwischen Server und Client.

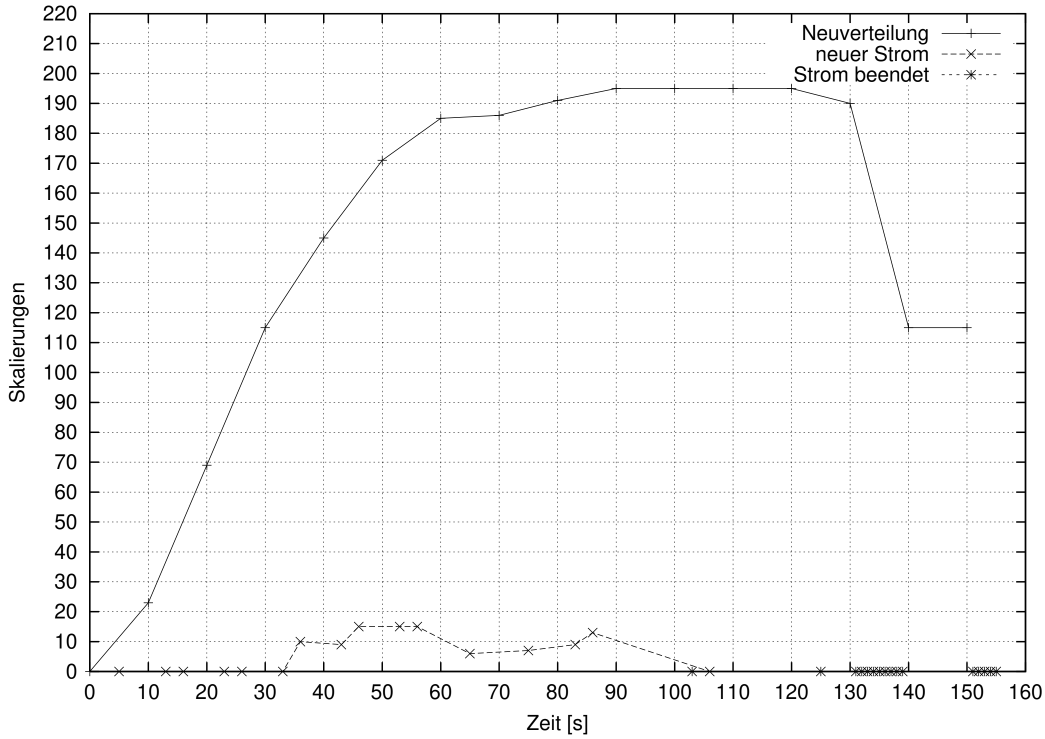


Abbildung 4.16: Anzahl erforderlicher Skalierungen bei Neuverteilungen und hinzukommenden Strömen.

die Reaktion auf Paketverluste nicht aktiv. Wie zu erwarten war, tritt hier ein ähnlicher Effekt ein: Nach Starten des Störsenders (t=17s) steigen die Round-Trip-Zeiten zwischen Server und Client (Abbildung 4.15) und der Audiostrom, der maximal 10 ms Ende-zu-Ende-Verzögerung akzeptiert (also eine Round-Trip-Time von 20 ms), wechselt die Service-Klasse (Abbildung 4.14). Danach ist Best Effort wieder weniger belastet und die Round-Trip-Zeiten fallen wieder, woraufhin der Strom wieder nach BE wechselt. Nach Beenden des Störsenders bleibt der Strom in BE.
Es ist also zu sehen, daß der QoS-Manager Delay-Zeiten berücksichtigt, allerdings erst bei einer Neuverteilung Konsequenzen daraus ziehen kann, da ja zwischen zwei Neuverteilungen die Ströme nicht die Klassen wechseln dürfen.

4.8 Fünfzehn Audioströme

von Simon Vey

Ziel dieser Messung war es, das Verhalten des QoS-Managers bei vielen Audioströmen zu erfassen. Zu diesem Zweck wurden nacheinander fünfzehn Audioströme gestartet, und es wurden die Anzahl der benötigten Skalierungen beim Anmelden und Abmelden eines Stroms sowie bei den Neuverteilungen ausgegeben. Des weiteren wurden auch die durchschnittlichen Qualitäten aller zu diesen Zeitpunkten im QoS-Management enthaltenen Ströme und die durchschnittlichen Abweichnungen von diesen durchschnittlichen Qualitäten ermittelt. Die Anzahl der Skalierungen ist ein Maß für die Komplexität einer Operation im QoS-Manager, die Abweichung von der durch-

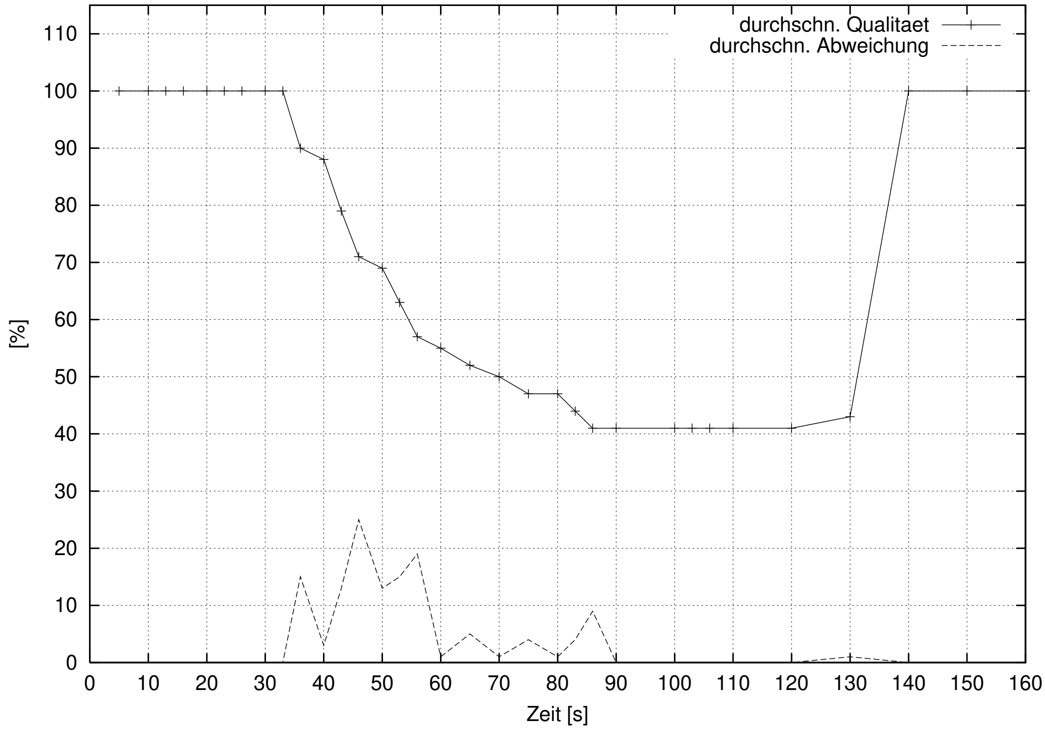


Abbildung 4.17: Durchschnittliche Qualitäten und durchschnittliche Abweichung von diesen Qualitäten.

schnittlichen Qualität ist ein Maß für die strombasierte Fairneß. Je größer die durchschnittliche Abweichung, desto unfairer ist die Verteilung der Ressourcen. In den einzelnen Service-Klassen standen folgende Bandbreiten per SLA zur Verfügung:

- EF = 2Mbit,
- AF21 = 2Mbit,
- AF11 = 2Mbit,
- BE = 4Mbit.

In Abbildung 4.16 sind die Anzahl der Skalierungen im QoS-Manager über die Zeit aufgetragen. Jeder Meßwert steht für die Anzahl der Skalierungen bei einer Neuverteilung (obere Kurve) oder beim Anmelden bzw. Abmelden eines Stroms (untere Kurve). Die ersten sechs Audioströme passen offensichtlich vollständig in die Service-Klassen. Demzufolge sind beim Anmelden dieser Ströme keine Skalierungen notwendig, wohingegen bei den Neuverteilungen alle Ströme in ihren höchsten Level (Level 22) hochskaliert werden müssen. Das führt zu einem linearen Anstieg der Skalierungen. Insgesamt erkennt man, daß beim Anmelden eines Stroms weit weniger Skalierungen notwendig sind als bei einer Neuverteilung. Einen neuen Strom in das Management aufzunehmen, ist also weniger komplex, was ja auch die Idee der Neuverteilung ist. Während bei den Neuverteilungen in diesem Beispiel bis zu 195 mal skaliert werden muß, sind bei einer Anmeldung eines neuen Stroms immer weniger als 20 Skalierungen erforderlich.

Aber auch die Neuverteilung wird nicht beliebig komplex. Da mit zunehmender Anzahl der Ströme die durchschnittliche Qualität der Ströme sinkt (siehe Abbildung 4.17), werden pro Strom auch weniger Skalierungen notwendig, um den entsprechenden Level zu erreichen.
Wird ein Strom beendet, wird seine Bandbreite nicht direkt unter den anderen Strömen aufgeteilt. D.h., ein Strom, der danach neu in das Management aufgenommen wird, kann diese Bandbreite nutzen, ohne daß Skalierungen notwendig sind. In Abbildung 4.16 wird dies deutlich. Bei t=103 Sekunden wird ein Strom beendet und bei t=106 Sekunden ein neuer gestartet. Es werden keine Skalierungen vorgenommen.
Abbildung 4.17 stellt die durchschnittlichen Qualitäten aller Ströme und die durchschnittlichen Abweichungen von diesen Qualitäten dar. Da die ersten sechs Ströme vollständig in den Klassen untergebracht werden können, senden diese am Anfang alle mit voller Qualität. Daher gibt es in diesem Zeitraum auch keine Abweichungen von der Durchschnittsqualität. Kommen dann noch weitere Ströme hinzu, sinkt die durchschnittliche Qualität mit jedem neuen Strom. Interessant ist, daß auch bei einer Neuverteilung nach Einfügen eines Stroms die durchschnittliche Qualität minimal sinkt. Das ist aber der Preis für eine fairere Verteilung. An der unteren Kurve kann man nämlich erkennen, daß mit dieser Neuverteilung die durchschnittliche Abweichung von der Durchschnittsqualität geringer wird. Die Verteilung ist also fairer. Beispielsweise sieht man bei der Neuverteilung nach 40 Sekunden, daß die durchschnittliche Qualität von 90% auf 88% fällt, dafür aber die durchschnittliche Abweichung auch von 15% auf 3% sinkt.
Nach 125 Sekunden wird ein Strom beendet. Bei der nächsten Neuverteilung wird seine Bandbreite unter den anderen Strömen aufgeteilt, was zu einem leichten Anstieg der durchschnittlichen Qualität nach der nächsten Neuverteilung führt. Nun werden weitere neun Ströme beendet. Die durchschnittliche Qualität steigt wieder auf 100% an, und die Anzahl der Skalierungen bei der nächsten Neuverteilung nimmt entsprechend ab. Ab t=151 Sekunden werden die restlichen Ströme beendet.
Diese Messung zeigt, daß das Einfügen und Entfernen von Strömen weit weniger aufwendig ist, als die Neuverteilungen, die ja in einem festen Intervall ausgeführt werden. Zu sehen ist auch, daß die Neuverteilungen zu einer faireren Aufteilung führen, da durch sie die durchschnittlichen Abweichungen von der durchschnittlichen Qualität verringert werden.

4.9 Messung der TCP-freundlichkeit für BE-Ströme

VON THOMAS DREIBHOLZ

Ein Ziel des QoS-Management ist es, ein TCP-freundliches **Best-Effort-Verhalten** für die verwalteten Multimediaströme herzustellen, d.h. eine gerechte Verteilung der BE-Bandbreite zwischen TCP- und UDP-Strömen herzustellen. Ohne eine Regulierung der Bandbreite würden ansonsten TCP-Ströme von UDP-Strömen verdrängt werden, da TCP die Bandbreite bei Paketverlusten sofort senkt.

Einführung in das Meßszenario

Diese Messung soll zeigen, daß durch das QoS-Management ein TCP-freundlichen Verhalten der RTP AUDIO-Ströme über BE erreicht wird. Zum Vergleich wird anschließend die gleiche Messung nochmals ohne Verlustskalierung durchgeführt.

Meßszenario (siehe Abbildung 3.10):

- Ein TCP-Testsender sendet vom Server *(corona)* zum 2. Client (*andechs*) über die BE-Klasse mit maximaler Bandbreite. Die maximale Bandbreite für BE wird dabei vom Core-Router *(holsten)* auf 5 MBit/s begrenzt (durch SLA: 3 MBit/s EF, 4 MBit/s AF11, 4 MBit/s AF21 sowie 5MBit/s BE). Am Border-Router (*grolsch*) ist kein SLA eingestellt.

- Vom Server *(corona)* zum 2. Client (*andechs*) werden mit einigem Abstand vier Audioströme gestartet, wobei die Clients jeweils die maximale Qualität (ca. 185 KBytes/s bei 142 Pakete/s) anfordern. Für das QoS-Management stehen 5 MBit/s BE-Bandbreite zur Verfügung; EF, AF11 und AF21 werden in diesem Szenario nicht genutzt. Der QoS-Manager muß daher spätestens beim 4. Client skalieren.

- Vom Server *(corona)* zum 2. Client (*andechs*) wird ein UDP-Testsender mit 2 MBit/s in Klasse EF gestartet. Dies ist notwendig, da der Router ansonsten nicht die vollen 5 MBit/s BE durchläßt (Der Router scheint bei Auslastung genauer zu arbeiten).

- Am Ausgang des Core-Routers *(holsten)* werden mit *NetLogger* Bandbreitemessungen durchgeführt. Jeder Client zeichnet die Anzahl der empfangenen und verlorengegangenen Pakete auf.

Messung 1

In Messung 1 wurde der beschriebene Versuchsaufbau mit eingeschalteter Verlustskalierung im QoS-Manager durchgeführt. Die Clients werden im Abstand von 60 Sekunden gestartet. Der gesamte Netzwerkverkehr ist in Abbildung 4.18 gezeigt, wobei die Clients bei 60, 120, 180 und 240 Sekunden der Reihenfolge nach gestartet und bei 330, 390, 450 und 505 Sekunden in umgekehrter Reihenfolge beendet werden. Man beachte, daß der UDP-Plot hier auch den EF-Sender (konstant 2 MBit/s) zeigt. Der TCP-Strom ist in Abbildung 4.20 zu sehen, angekommene Pakete und Anzahl der verlorengegangenen Pakete (aus Sequenznummer errechnet) an den vier Clients sind in den Abbildungen 4.22 (1. Client), 4.24 (2. Client), 4.26 (3. Client) und 4.28 (4. Client) zu finden. Man beachte hier, daß sich die Zeitangaben der einzelnen Clients jeweils auf deren Laufzeit beziehen!

Ergebnisse:

- Der TCP-Sender sendet am Anfang und am Ende mit ca. 5 MBit/s. Das eingestellte SLA funktioniert also.

- In Abbildung 4.18 ist zu erkennen, daß der QoS-Manager aufgrund von Paketverlusten hoch- und runterskaliert. Der TCP-Strom erhält dann entsprechend mehr bzw. weniger Bandbreite.

- Betrachtet man die Paketverluste der Clients, so liegen diese meistens deutlich unter 5%. Insbesondere in der Zeitspanne, in der alle vier Clients gleichzeitig laufen, liegt sie bei etwa 2 bis 4%.

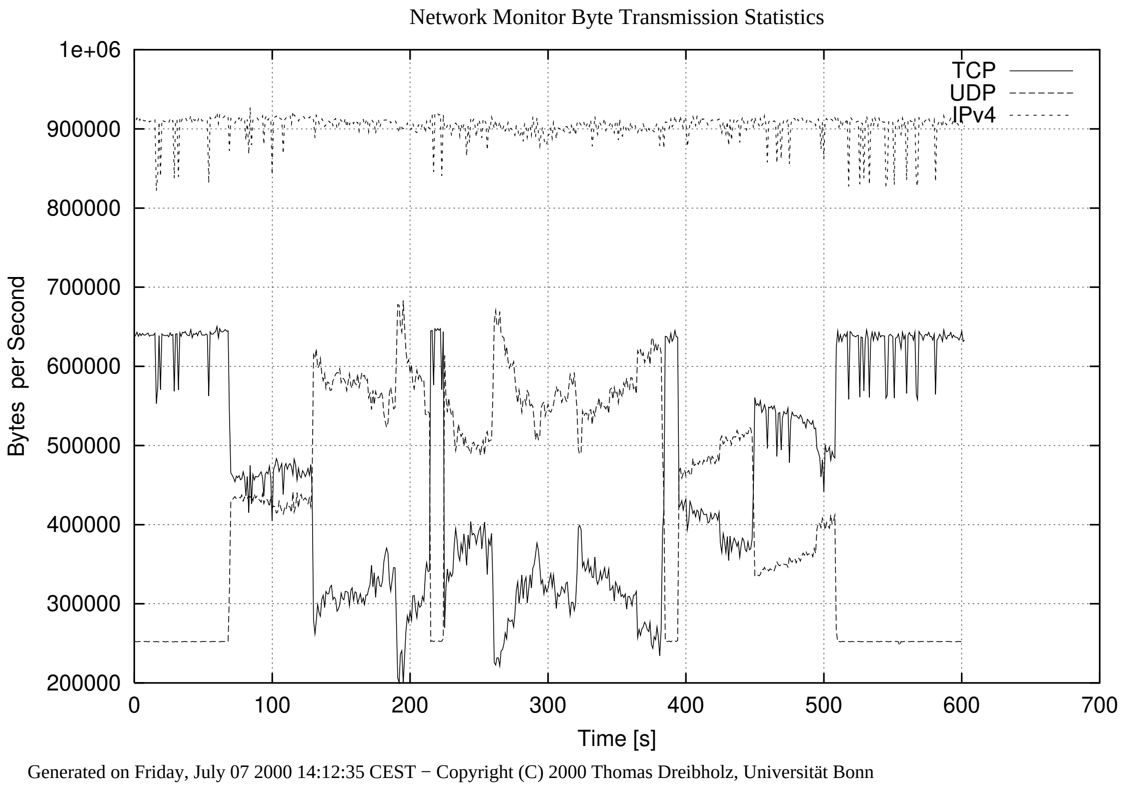


Abbildung 4.18: Der gesamte Netzwerkverkehr bei Messung 1 (mit Verlustskalierung)

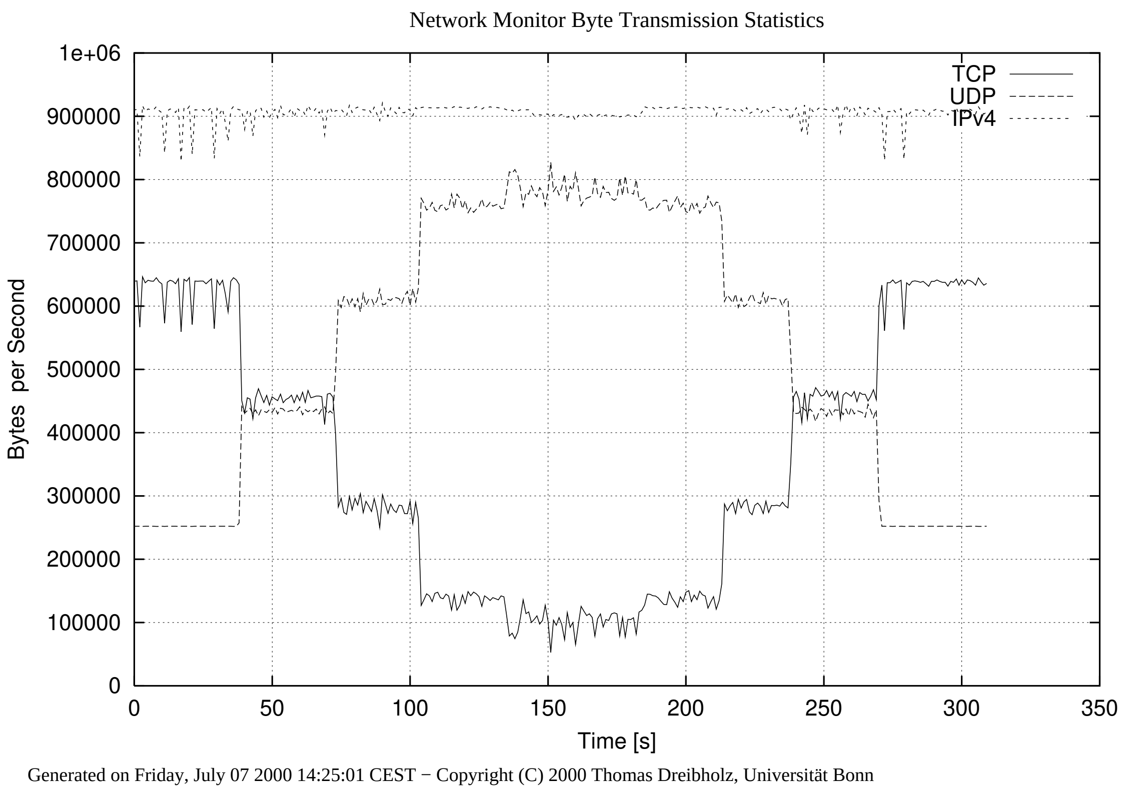


Abbildung 4.19: Der gesamte Netzwerkverkehr bei Messung 2 (ohne Verlustskalierung)

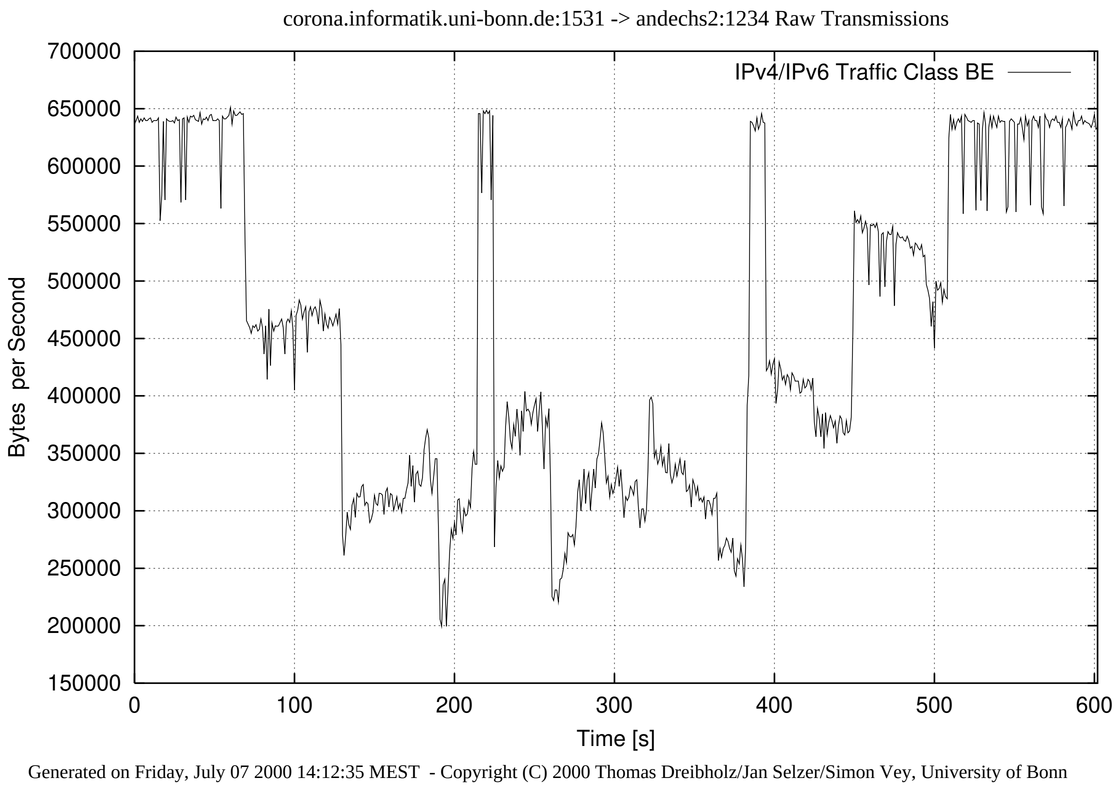


Abbildung 4.20: Der TCP-Sender in der ersten Messung (mit Verlustskalierung)

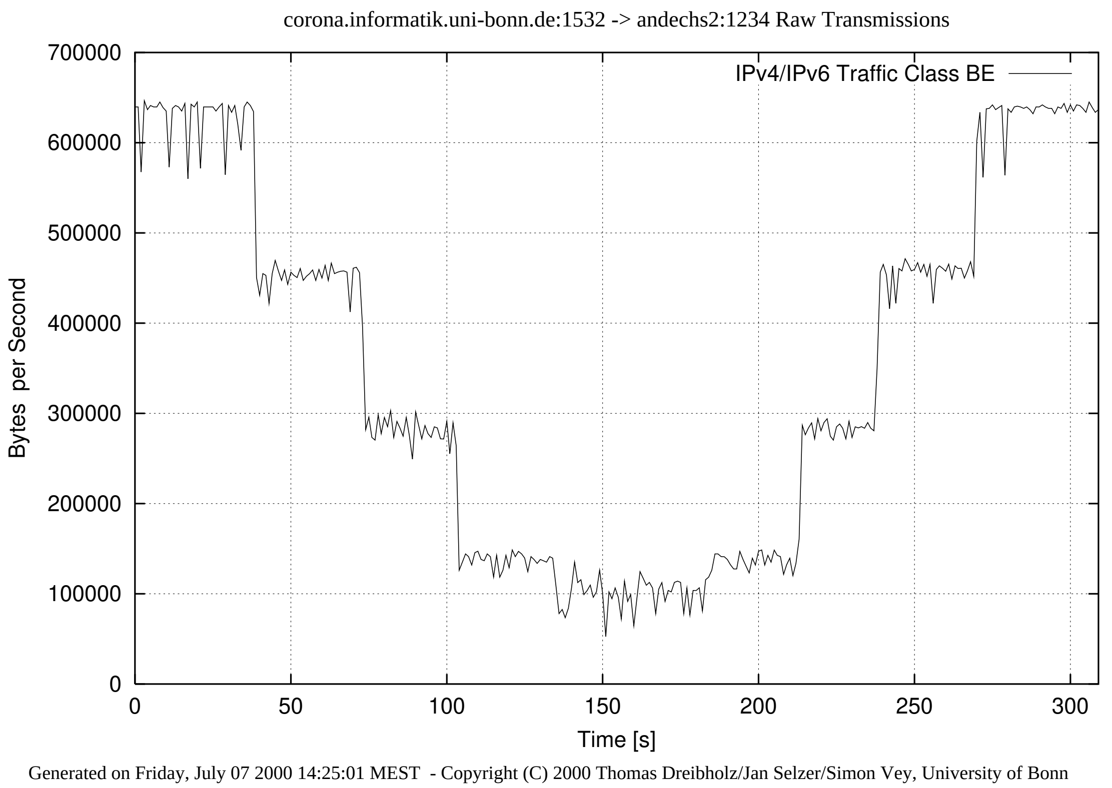


Abbildung 4.21: Der TCP-Sender in der ersten Messung (ohne Verlustskalierung)

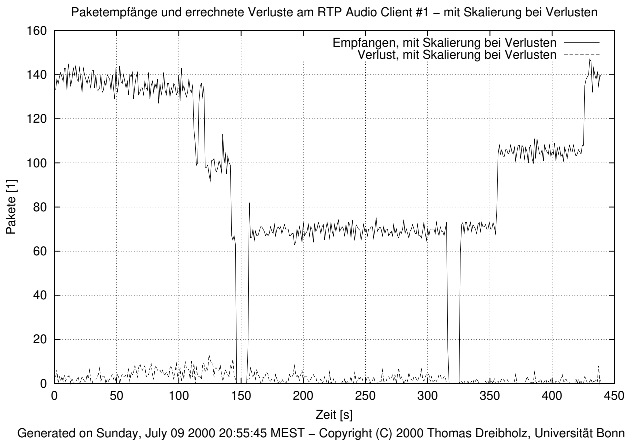


Abbildung 4.22: Client #1 - mit ...

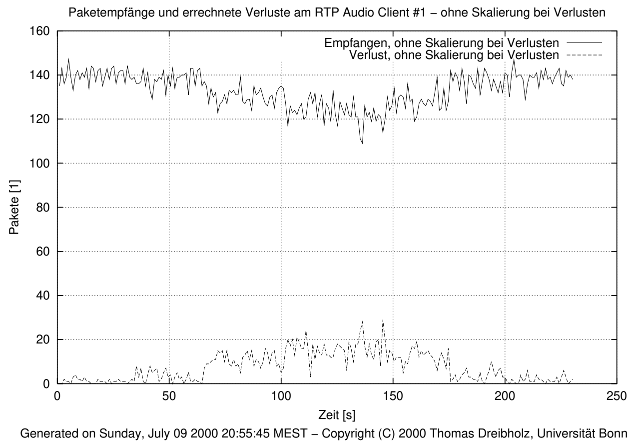


Abbildung 4.23: ... und ohne Skalierung

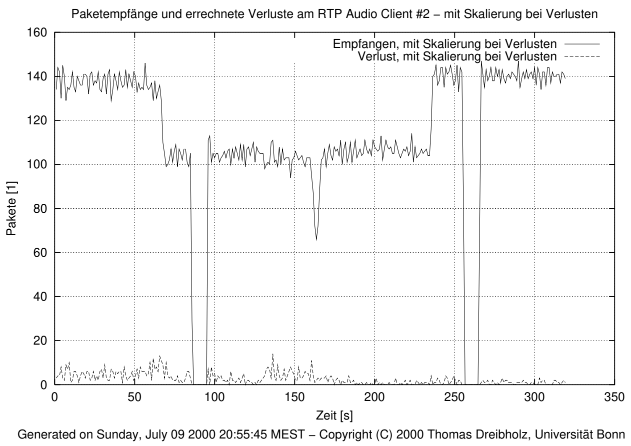


Abbildung 4.24: Client #2 - mit ...

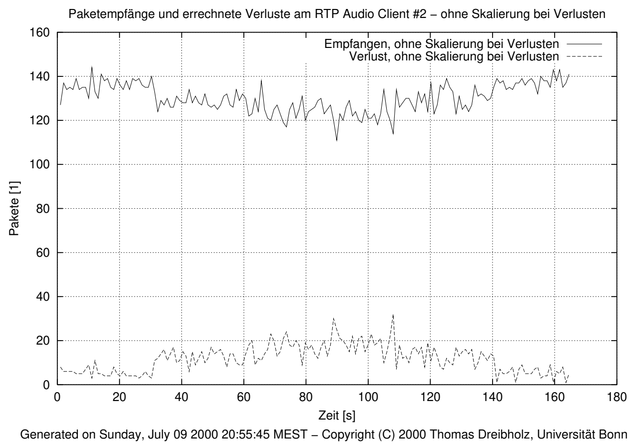


Abbildung 4.25: ... und ohne Skalierung

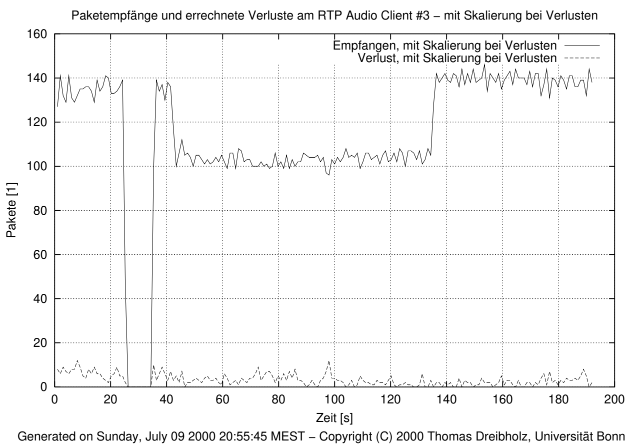


Abbildung 4.26: Client #3 - mit ...

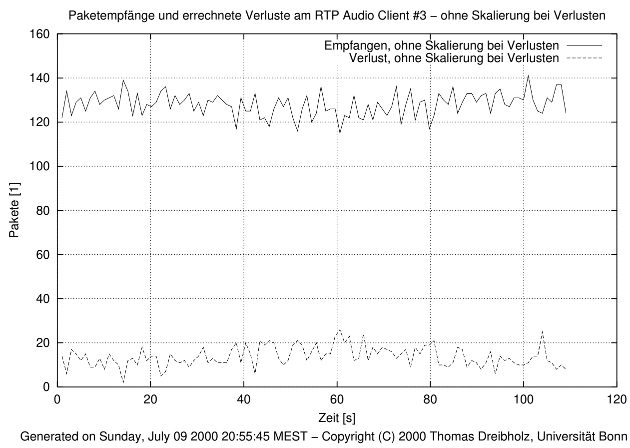


Abbildung 4.27: ... und ohne Skalierung

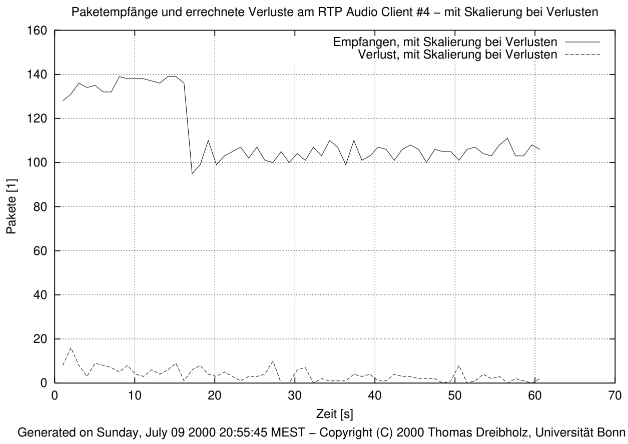


Abbildung 4.28: Client #4 - mit ...

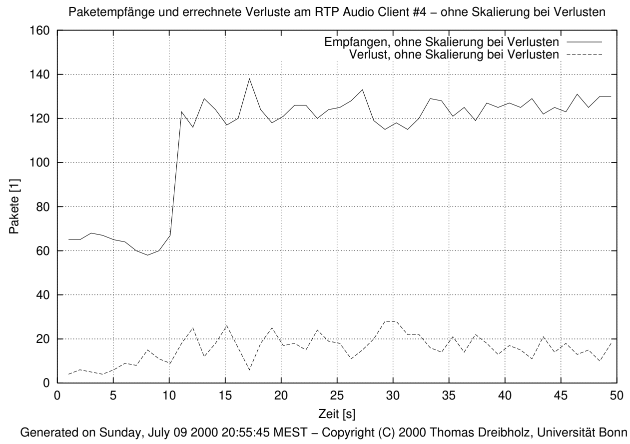


Abbildung 4.29: ... und ohne Skalierung

- Bei 220s und 380s hält der QoS-Manager alle Ströme an, da hier die Roundtripzeiten[1] zu hoch werden. Aus den Roundtripzeiten wird die Transportverzögerung berechnet (etwa die halbe Roundtripzeit, siehe dazu Abschnitt 2.1.6). Die maximale akzeptierte Verzögerung ist auf 100 ms eingestellt. Hier würde - wenn andere Klassen außer BE verfügbar wären - ein Wechsel in eine dieser Klassen versucht werden. Der TCP-Strom erhält in dieser Zeit die volle Bandbreite von 5 MBit/s.

- Selbst bei voller Belastung durch 4 RTP AUDIO Clients hält der TCP-Strom fast immer noch etwa 300000 bis 400000 Bytes/s.

Messung 2

Diese Messung wiederholt die erste Messung, wobei jetzt zum Vergleich die Skalierung bei Paketverlusten ausgeschaltet worden ist. Die Clients werden hier im Abstand von 30 Sekunden gestartet. In Abbildung 4.19 ist wieder der gesamte Netzwerkverkehr dargestellt, wobei die Clients bei 40, 70, 100 und 130 Sekunden nacheinander gestartet und bei 130, 180, 240 und 270 Sekunden in umgekehrter Reihenfolge beendet werden. In Abbildung 4.21 befindet sich der TCP-Strom und in den Abbildungen 4.23 (1. Client), 4.25 (2. Client), 4.27 (3. Client) und 4.29 (4. Client) die Paketstatistik für die vier Clients.

Ergebnisse:

- Der TCP-Sender sendet am Anfang und am Ende wieder mit ca. 5 MBit/s.

- Während nach dem Einschalten des ersten Clients die Senderate des TCP-Stromes mit 450000 Bytes/s ähnlich derer der ersten Messung ist, fällt sie beim zweiten Client schon auf unter 300000 Bytes/s ab. Beim dritten Client sind es noch ca. 130000 Bytes/s. Durch den letzten Client wird die Bandbreite dann auf Werte zwischen ca. 50000 Bytes/s und 110000 Bytes/s

[1] Die Roundtripzeitmessung geschieht mittels ICMP Echo Requests und Echo Replies, wie in Abschnitt 3.3 beschrieben. Dort wird insbesondere auch das Verhalten bei ICMP-Paketverlusten beschrieben.

eingeschränkt (Man beachte: Der QoS-Manager muß hier etwas skalieren, um die 5 MBit/s auf 4 Clients aufzuteilen).

- Die Paketverluste der Clients steigen im Vergleich zur ersten Messung erheblich an (teilweise über 20% im Gegensatz zu i.d.R. deutlich unter 5% in der ersten Messung). Insbesondere sieht man den erwartungsgemäßen Anstieg bei jedem weiteren gestarteten Client.

Bewertung der Meßergebnisse

Wie aus dem Vergleich der ersten Messung mit der zweiten zu sehen ist, wird der TCP-Strom nicht durch die vier UDP-Sender verdrängt, sondern er erhält aufgrund der Regulierung der UDP-Ströme durch das QoS-Management bis auf wenige kurze Ausnahmen deutlich mehr als 300000 Bytes/s. Dies ist bei 5 MBit/s = 625000 Bytes/s fast die Hälfte der möglichen Bandbreite. Dies hat zur Folge, daß sich nun die UDP-Ströme die verbleibende Bandbreite teilen müssen - also etwa 80000 Bytes/s pro Strom.

Diese Regelung ist daher - falls UDP- und TCP-Ströme gleichberechtigt sein sollen - schon etwas zu TCP-freundlich. Im theoretischen Idealfall sollte jeder der 5 Ströme etwa 125000 Bytes/s erhalten.

Eine Abhilfe für dieses Problem beim Best-Effort-Dienst soll das Endpoint Congestion Management (ECM) bringen, welches eine TCP-ähnliche Regelung für Best-Effort-UDP-Ströme oberhalb des QoS-Managers realisiert, wobei die TCP-freundlichkeit konfigurierbar ist.

Bei der Benutzung der reservierten Klassen (EF bzw. AF) besteht dieses Problem nicht. Der QoS-Manager schaltet - wie die vorherigen Messungen gezeigt haben - bei Überlast oder zu hohen Roundtripzeiten in BE einfach auf AF oder EF um.

4.10 Zusammenfassung der Meßergebnisse

von Jan Selzer

Insgesamt kann man die Funktionsfähigkeit des Projektes als gut bezeichnen. Durch Messungen hat man festgestellt, daß die DiffServ-Grundfunktionen vom Border-Router sowie vom Core-Router erfüllt werden. Die DiffServ-Klassen weisen ein erwartetes Verhalten auf. Das System erreicht eine faire Verteilung der Bandbreite für verschiedene Ströme. Außerdem ist hier ein TCP-freundliches Verhalten zu beobachten. Dieser Aspekt kann aber noch durch die Arbeit von Thorsten Karl zu "Endpoint Congestion Management" verbessert werden.

Kapitel 5

Erweiterungsmöglichkeiten

Hier werden Vorschläge für mögliche Erweiterungen und Verbesserungen gemacht, welche an DiffServ-Router, RTP Audio und QoS-Manager z.B. im Rahmen weiterer Praktikumsarbeiten gemacht werden können.

5.1 DiffServ-Router

von Jan Selzer

Dies sind einige Erweiterungsmöglichkeiten für den DiffServ-Router:

- Ummarkierungsmöglichkeit auf dem Border-Router, z.B. Pakete der AF-Klassen höherer Priorität werden bei Bandbreitenüberschreitung in Klassen niedrigerer Priorität ummarkiert. Ein Skript dazu existiert schon.
- Unterstützung weiterer AF-Unterklassen.
- Traffic Shaping auf dem Border-Router, anstatt Pakete zu verwerfen.
- Bandwidth-Broker-Ansatz mit dynamischem SLA. Dies würde zu einer besseren Ausnutzung der DiffServ-Klassen führen.
- IPv6-Verwendung bei DiffServ-Routern. *iproute2* unterstützt mittlerweile IPv6, es gibt jedoch noch Probleme damit.
- Unterstützung der Verwendung von RSVP. Diese Eigenschaft hängt mit der Verwendung des Bandwidth Brokers und dynamischem SLA in einem IntServ/DiffServ-Szenario zusammen. Da man für das dynamische SLA ständige Abfrage der Ressourcen des ISPs benötigt, wird hierfür ein Signalling-Protokoll verwendet.

5.2 RTP Audio

von Thomas Dreibholz

Im Folgenden wird eine Aufstellung möglicher Erweiterungen für RTP Audio gegeben:

Weitere Kodierungen

- Simple- und Advanced Audio Encoding übertragen ihre Daten unkomprimiert, für die maximale Qualität sind dies ca. 185 KBytes/Sekunde. Sinnvoll wäre hier eine Möglichkeit, die Daten komprimiert zu übertragen, um Bandbreite zu sparen. Ideen:

  1. **Die Verwendung von MP3**:
     Allerdings besteht hier das Problem, daß die Daten schon vorher komprimiert vorliegen müssen - für jede Qualitätsstufe eine eigene MP3-Datei. Eine Echtzeitkomprimierung ist - mit der heutigen Hardware - noch zu aufwendig und zudem ist momentan kein freier MP3-Encoder verfügbar (wegen MP3-Patent).
  2. **Die Implementation einer eigenen Kompression**:
     Man könnte evtl. die Audiodaten Fourier-transformieren und das Frequenzspektrum in mehrere Bereiche nach - Wichtigkeit sortiert - aufteilen. Diese Bereiche könnte man einzeln in Echtzeit komprimieren (mit irgend einem schnellen und frei verfügbaren Algorithmus) und dann in eigenen Layern übertragen.

- Verschlüsselung zur sicheren Übertragung vertraulicher Audiodaten.

- Verbesserung der Reparaturmöglichkeiten von fehlenden Paketen durch Forward Error Correction (FEC). Dies ist im Rahmen einer Praktikumsaufgabe bereits in Arbeit.

Eine Implementation neuer Kodierungen in RTP AUDIO ist problemlos möglich. Dazu sind nur die Interfaces *AudioEncoderInterface* und *AudioDecoderInterface* zu implementieren und die Objekte der neuen Kodierung analog zu Simple- und Advanced Audio Encoding in den Klassen *AudioServer* bzw. *AudioClient* dem Repository hinzuzufügen.

RSVP und Quality of Service

Das Hinzufügen von RSVP-Reservierungen sollte sehr einfach möglich sein. Unterstützung für IPv6-Flowlabels ist bereits in RTP AUDIO integriert. Die Klasse *AudioClient* besitzt Methoden zur Abfrage von Senderadresse, Flowlabel und Traffic Class für jedes Layer. Es ist daher nur noch notwendig, die Anbindung zur API des RSVP -Systems zu implementieren und darüber die entsprechenden Reservierungen vorzunehmen.

Analog zur Verwendung des DiffServ-Routers durch Setzen von entsprechenden Traffic Classes wäre dies auch mittels bestimmter, vom QoS-Manager und Reservierungs-Modul reservierter Flowlabels auf Rechner-zu-Rechner-Ebene möglich. Diese könnten dann analog zu den Traffic Classes einzelnen Strömen zugeteilt werden. Der Wechsel eines Stromes auf ein anderes Flowlabel ist bereits implementiert durch das *FlowLabel*-Feld in *ExtendedTransportInfo*.

Im Zusammenhang mit Reservierungen und Prioritäten könnte man zudem Abrechnungsfunktionalitäten einbauen: Ein Benutzer, welcher hohe Übertragungsqualitäten benutzt, bezahlt dafür mehr als für eine Best Effort-Übertragung. Dazu sind dann Benutzerauthentifizerung und weitere Verwaltungsfunkionen notwendig: Will der Benutzer für bessere Übertragung mehr bezahlen? Darf der Benutzer die bessere Qualität anfordern? ....

RTP Audio und Mobiles Internet

Der mobile Internetzugriff über GPRS und vor allem UMTS wird in den kommenden Jahren zunehmend an Bedeutung gewinnen. Es wäre daher sinnvoll zu prüfen, inwieweit RTP Audio für mobilen Zugriff erweitert werden sollte:

- Große Paketverluste bei Funkübertragung:
  - Kodierungen sollten Redundanz, Fehlerkorrektur usw. besitzen,
  - Der QoS-Manager darf hierbei - im Gegensatz zu Verlust durch Congestion - nicht herunterskalieren,
  - ...
- Große Verzögerungszeiten,
- Hohe Bandbreiten verursachen hohe Kosten → Kompression,
- Unterstützung von UMTS-Reservierungen,
- ...

Dies müßte - mangels Testmöglichkeiten - mittels einer Simulation analysiert werden. Idee: Zwischen den Server und die Clients wird ein Simulator als Proxy geschaltet, welcher die Pakete möglichst "mobilfunkähnlich" verzögert, vertauscht, verwirft, etc. und UMTS-Fähigkeiten wie QoS und Reservierungen hinzufügt. Um ein solches Proxy auf IP-Ebene zu realiseren, könnte das EtherTap-Device im Linux-Kernel sehr nützlich sein (siehe [EtherTap]).

Meßwerkzeuge

Die Meßwerkzeuge könnten um eine graphische Benuzueroberfläche (z.B. mit *Qt*) erweitert werden, die das Eingeben der Kommandozeilenoptionen erspart. Bei NetAnalyzer wären damit verbunden z.B. die folgenden Erweiterungen nützlich:

- Das Auswählen einzelner Ströme,
- das Setzen von GNU Plot-Einstellungen, einzeln für jede Seite,
- eine Bildschirmvorschau,
- das Verwenden von Schablonen (z.B. "Zeige alle IPv6-Ströme größer als 200 KBytes/s von ipv6-grolsch nach ipv6-corona mit gesetztem Flowlabel und Traffic Class EF..."),
- ...

NetLogger könnte weitere Netzwerktypen unterstützen, also außer Ethernet auch z.B. Token Ring oder ATM. Sobald entsprechende Hardware vorhanden ist, sollte natürlich auch UMTS implementiert werden.

5.3 QoS-Manager

von Simon Vey

Wie die Messung in Abschnitt 4.9 gezeigt hat, verhält sich der QoS-Manager aufgrund des schnellen Herunterskalierens und des sehr langsamen Hochskalierens sehr TCP-freundlich. Da der TCP-Strom im Durchschnitt mehr Bandbreite erhält als ein Audiostrom, könnte man sagen, daß sich das System schon zu TCP-freundlich verhält. Hier wäre auf jeden Fall ein Endpoint Congestion Management für Best Effort sinnvoll, was momentan auch schon im Rahmen einer Diplomarbeit entwickelt wird.
Bisher werden Paketverluste und Ende-zu-Ende-Verzögerungen der einzelnen Klassen vom QoS-Manager beachtet. Sinnvoll wäre hier auf jeden Fall noch die Berücksichtigung des Jitter, da auch der Jitter durchaus eine Rolle bei multimedialer Kommunikation spielt. Schwanken die Zwischenankunftszeiten der Pakete zu stark, reicht eventuell die Puffergröße beim Empfänger nicht mehr aus, um diese Schwankungen abzufangen.
Ein weiterer Aspekt, der in Zukunft in Angriff genommen werden soll, ist die dynamische QoS-Beschreibung der Ströme. Die dynamische QoS-Beschreibung soll dem QoS-Manager die Möglichkeit geben, progressive Datenströme wie z.B. bei Prog4D zu verwalten, die in der Lage sind, mit sehr vielen unterschiedlichen Datenraten zu senden. Der QoS-Manager soll dann in der Lage sein, aufgrund von Meta-Informationen über die einzelnen Ströme, diese Ströme selbständig zu skalieren, ohne daß explizit jede Skalierungsstufe in einer statischen Beschreibung festgelegt ist.

Anhang A

Die Benutzerdokumentation zu RTP Audio

von Thomas Dreibholz

Sämtliche Programme des RTP Audio-Paketes sind in diesem Anhangskapitel mit ihren Parametern dokumentiert. Zudem befinden sich hier auch noch Screenshots zu den einzelnen Programmen. Der Download des gesamten Paketes ist auf der [RTP Audio] Homepage möglich.

A.1 Der RTP Audio Server

A.1.1 server - Der RTP Audio Server

Beschreibung

Dies ist der Server von RTP Audio. Er wartet an einem gegebenen UDP-Port auf RTCP-Pakete von Clients und startet einen neuen Sender-Thread bei Eintreffen eines neuen RTCP-SDES-CNAME-Paketes.

Bildschirmfoto

Siehe Abbildung A.1.

Aufruf

./server
    {-port=port}
    {-directory=path}
    {-manager=host:port}
    {-local=host}
    {-timeout=secs}
    {-disable-qm}
    {-force-ipv4}

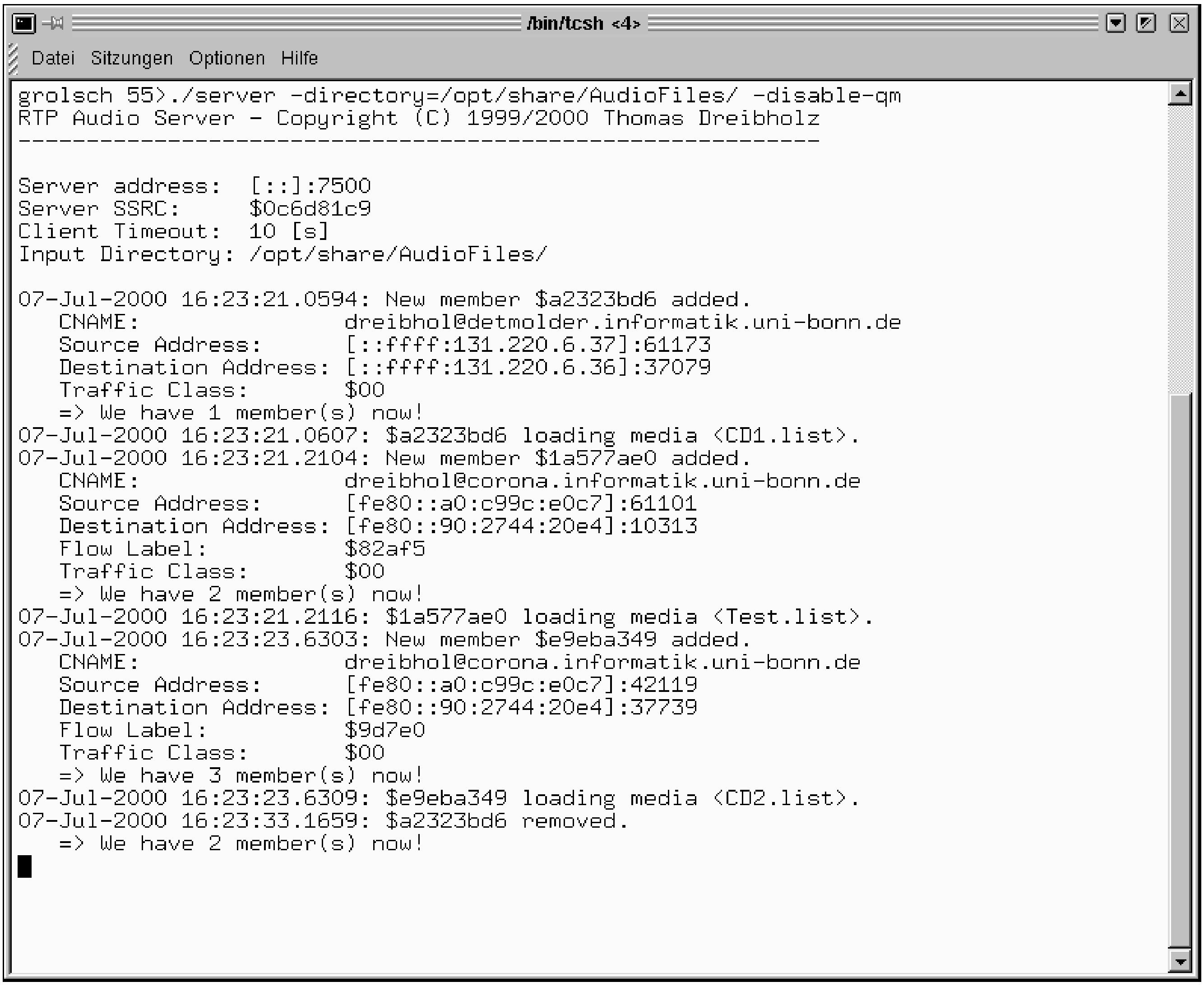


Abbildung A.1: Der RTP Audio Server

Parameter

-port=port: Dies ist der UDP-Port für den Empfang der RTCP-Pakete. Der Default-Wert ist 7500.

-directory: Mit diesem Parameter kann ein Pfad zu einem Verzeichnis mit den Eingabedateien gesetzt werden. Beispiel: -directory=../AudioFiles

-manager=host:port: Hiermit wird die Adresse des Congestion-Managers gesetzt. Wird der Congestion Manager zusammen mit dem QoS-Manager benutzt, so wird von beiden Bandbreite-Werten der minimale Wert verwendet.

-local=host: Soll das RTCP-Socket an eine bestimmte Adresse gebunden werden, so kann dies hiermit angegeben werden. Damit kann erreicht werden, daß RTP AUDIO z.B. nur Anfragen von einem bestimmen Interface erhält (eth1, eth2 o.ä.) oder ausschließlich IPv6 benutzt. Beispiel: -local=3ffe:0815:4711::111.

-timeout=secs: Der RTP AUDIO Server entfernt laufende Clients, sobald innerhalb eines gegebenen Timeouts kein RTCP Receiver Report empfangen wurde. Dieser Timeout kann hiermit gesetzt werden. Beispiel: -timeout=30.

-disable-qm: Hierdurch wird die Benutzung des QoS-Managers ausgeschaltet.

-force-ipv4: Hierdurch wird die Benutzung von IPv6 ausgeschaltet.

Das Dateiformat der Audiolisten

Audiolisten enthalten eine Liste von WAV- und/oder MP3-Dateien und/oder weiteren Audiolisten. Der Anfang einer solchen Liste beginnt immer mit der Zeile “AudioList”. Darauf folgen entweder Kommentare (#-Zeichen in 1. Spalte), Leerzeilen, Optionen (*-Zeichen in 1.Spalte) oder Angaben von Dateinamen. Die Option *directory=name setzt das Verzeichnis, in dem die folgenden Dateinamen gesucht werden, auf den angegebenen Namen. Dieser ist relativ zum aktuellen Verzeichnis. anzugeben. Die Optionen *title=Titel, *artist=Interpret und *comment=Kommentar setzen Titel, Interpret und Kommentar der nächsten Audiodatei in der Liste. Diese Optionen ersetzen bei MP3-Dateien die entsprechenden MP3-Tags (falls vorhanden).

Beispiel:

```
AudioList

# Hier wird das Verzeichnis gesetzt,
# in dem die folgenden Dateien gesucht werden.
# Die Angabe ist relativ zum aktuellen Verzeichnis.
*directory=../AudioFiles/MP3s

# Jetzt werden Titel, Interpret und Kommentar für die
# nächste Datei gesetzt.  Die MP3-Tags werden ignoriert.
```

```
*title=Born To Be Wild
*artist=Steppenwolf
*comment = RTP AUDIO Test File #1
BornToBeWild.mp3

# Hier werden wieder die MP3-Tags für diese Informationen
# verwendet.
Hellraiser.mp3
```

A.1.2 EncoderInfo - Informationen über die Audiokodierungen in RTP Audio

Beschreibung

EncoderInfo liefert Informationen über Bandbreite, Pakete pro Sekunde, Levels und Layers für die in RTP Audio eingebauten Kodierungen Advanced Audio Encoding und Simple Audio Encoding.

Aufruf

./encoderinfo
    [-ti]
    [-quality]
    [-levels]

Parameter

- -ti: Es werden die TransportInfo-Strukturen für beide Kodierungen ausgegeben.
- -quality: Es werden die Daten jeder möglichen Qualitäts-Einstellung für beide Kodierungen ausgegeben.
- -levels: Es werden die Audioqualitäts-Levels ausgegeben.

A.2 Die RTP Audio Clients

A.2.1 client - Der RTP Audio Client (Text-Version)

Beschreibung

Dies ist der Text-Client für RTP Audio. Er startet eine Verbindung zu einem gegebenen RTP Audio Server und gibt die empfangenen Daten entweder auf das Audio-Device oder den Audio-Debugger aus bzw. ignoriert diese (Audio-Null). Zusätzlich kann eine Ausgabe der Byte- und Paketrate, Verlusten sowie Jitters als GNU Plot-Datei gemacht werden.

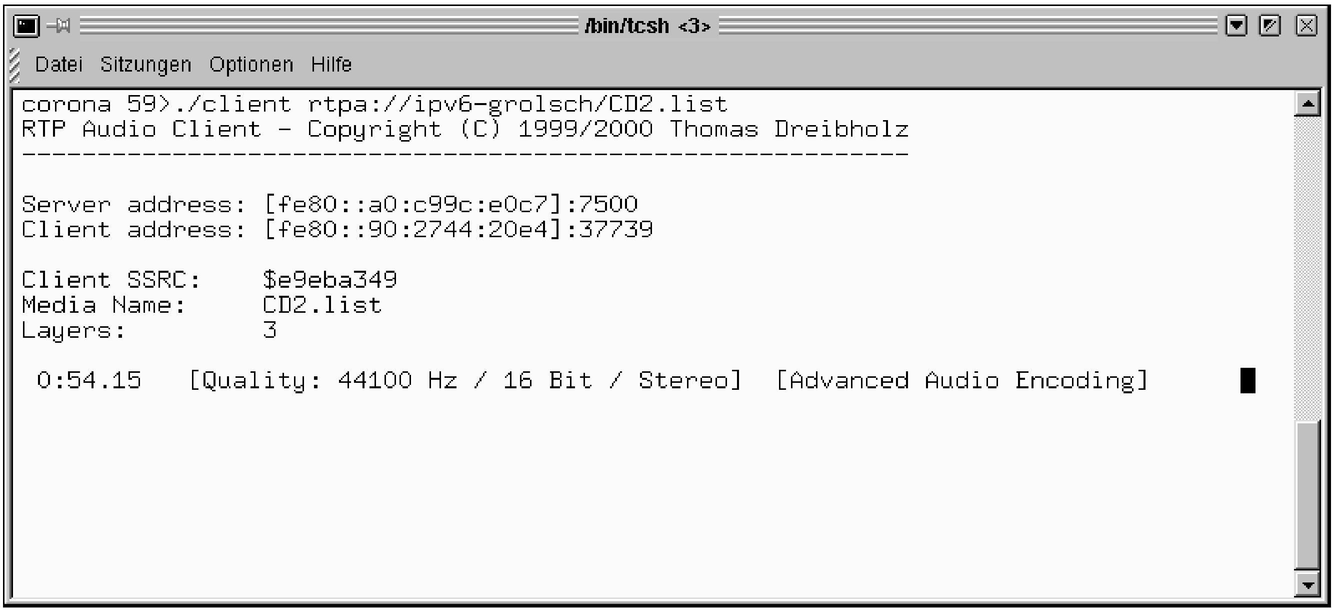


Abbildung A.2: Der Text-Client

Bildschirmfoto

Siehe Abbildung A.2.

Aufruf

./client
  [URL]
  {-local=host{:Port}}
  {-prefix=filename prefix}
  {-info=info string}
  {-debug}
  {-null}
  {-encoding=number}
  {-rate=sampling rate}
  {-bits=bits}
  {-stereo}
  {-mono}
  {-force-ipv4}

Parameter

URL: Hiermit werden Adresse und Portnummer des RTP AUDIO Servers sowie der Dateiname der abzuspielenden Audioliste angegeben.
Beispiel: rtpa://ipv6-odin:7500/Test1.list

Abbildung A.3: Der Audio-Mixer der QClients

-prefix=filename: Dies ist das Dateinamen-Präfix für die GNU Plot-Ausgabe. Das Skript erhält die Endung ".gp", die Daten ".data". Wird dieser Parameter nicht angegeben, so wird keine Ausgabe erzeugt.

-info=string: Hier kann ein optionaler Informationstext für die GNU PLOT-Ausgabe angegeben werden. Dieser erscheint dann hinter dem Datum.

-local=host:{port}: Soll das RTP-Empfänger-Socket an eine bestimmte Adresse und optional an einen bestimmten Port dieser Adresse gebunden werden, so kann dieses hiermit erreicht werden. Beispiel: -local=phoenix oder -local=gaffel:1234.

-debug: Anstatt die empfangenen Daten auf das Audio-Device auszugeben, wird der Audio-Debugger benutzt. Er zeigt die Ausgabepuffer-Belegung in regelmäßigen Abständen an, wodurch sich Probleme in den Kodierungen leichter finden lassen.

-null: Empfangene Daten werden ignoriert (Null-Device).

-encoding=index: Die Kodierung kann durch Angabe des Index gewählt werden:
0 Advanved Audio Encoding
1 Simple Audio Encoding.

-rate=sampling rate: Hier kann die Sampling-Rate für die Audioübertragung angegeben werden. Falls diese nicht verfügbar ist, so wird der nächstniedrigere Wert verwendet.

-bits=bits: Dieser Parameter stellt die Anzahl der Audio-Bits ein: 4, 8, 12 oder 16.

-stereo: Die Audio-Übertragung erfolgt in Stereo.

-mono: Die Audio-Übertragung erfolgt in Mono.

-force-ipv4: Hierdurch wird die Benutzung von IPv6 ausgeschaltet.

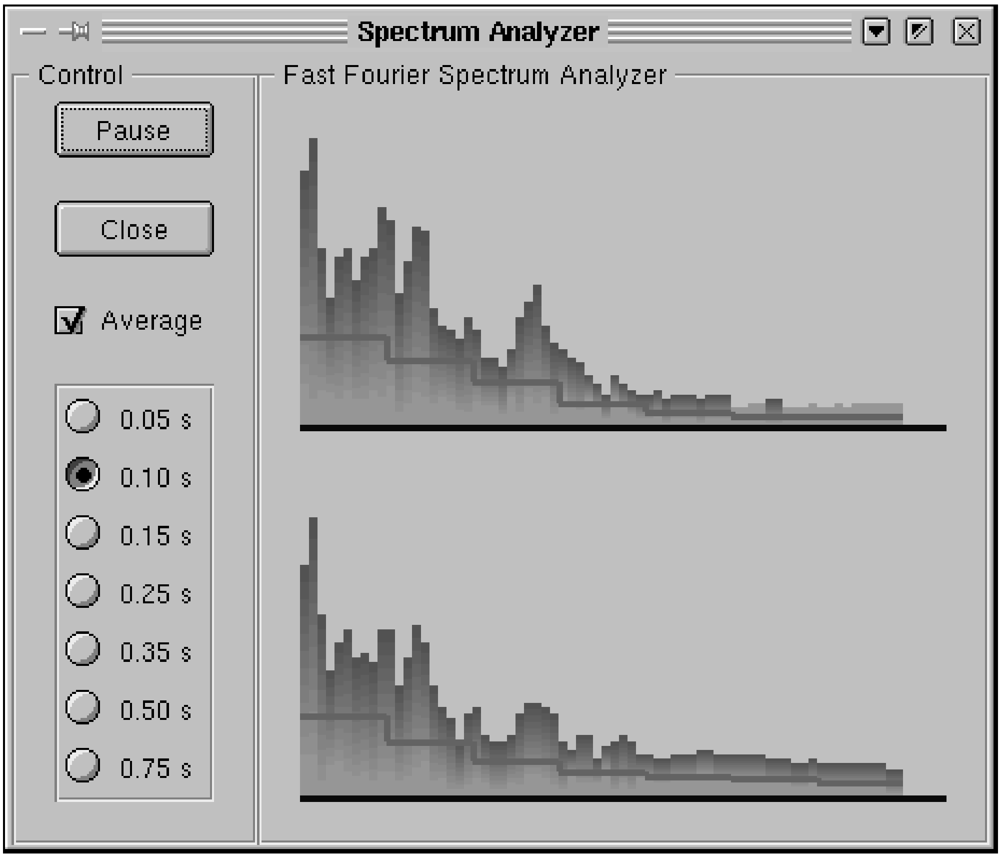


Abbildung A.4: Der Spectrum-Analyzer des QClients

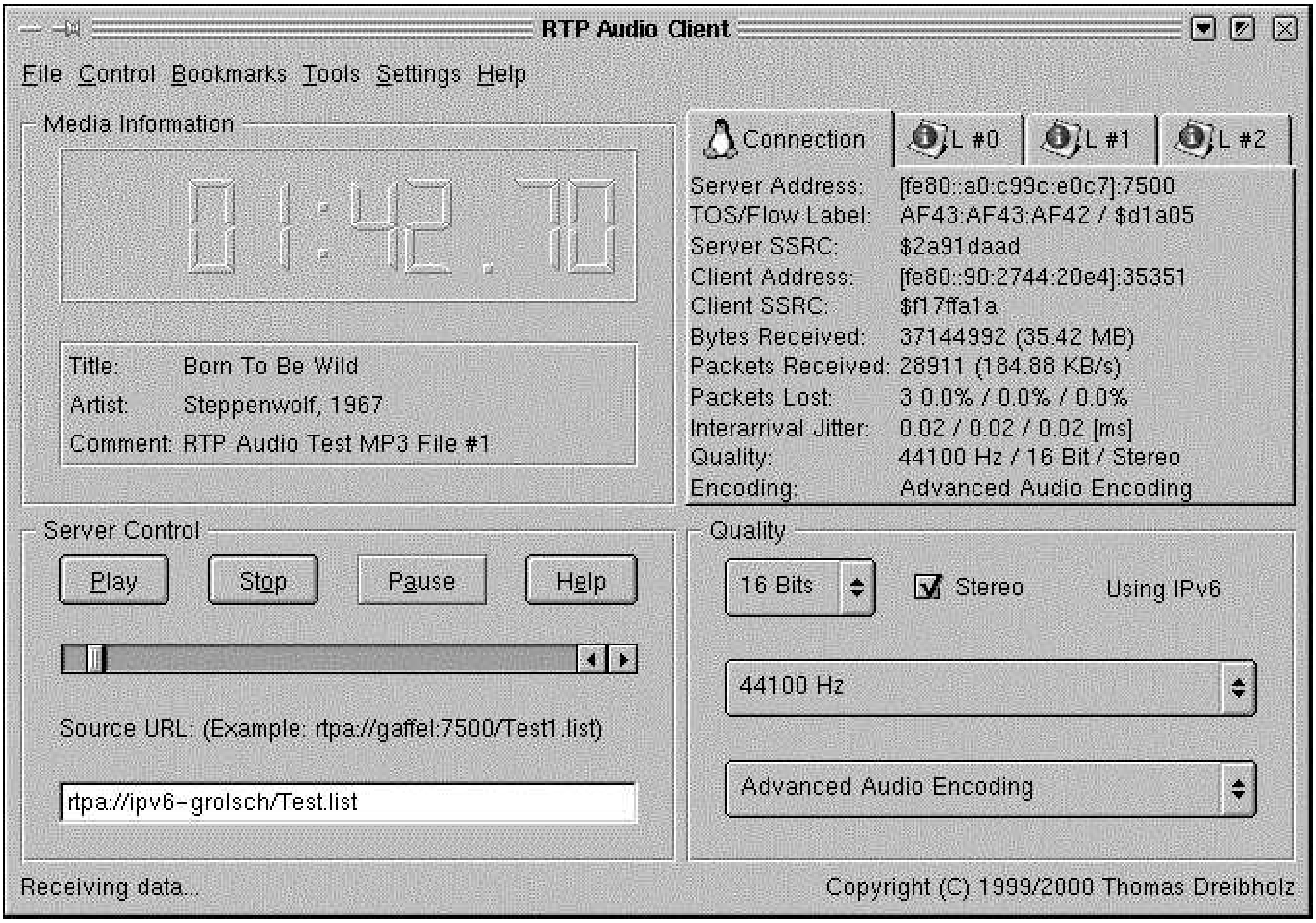


Abbildung A.5: Der QClient

A.2.2 qclient - Der RTP Audio Client (Qt-Version)

Beschreibung

Dies ist der graphische Client (Qt-Oberfläche) für RTP Audio. Er startet eine Verbindung zu einem gegebenen RTP Audio Server und gibt die empfangenen Daten entweder auf das Audio-Device und/oder Audio-Debugger und/oder Spectrum-Analyzer aus bzw. ignoriert diese (Audio-Null).

Bildschirmfotos

Siehe Abbildungen A.5 (QClient), A.4 (Spectrum Analyzer) und A.3 (Audio-Mixer).

Aufruf

./qclient
{-url=URL}
{[+/-]debug}
{[+/-]null}
{[+/-]device}
{[+/-]analyzer}
{[+/-]mixer}
{-force-ipv4}
{-local=hostname{:port}}

Parameter

-url: Hiermit können Adresse und Portnummer des RTP Audio Servers sowie der Dateiname der abzuspielenden Audioliste angegeben werden. Beispiel: rtpa://ipv6-odin:7500/Test1.list. Wird eine URL angegeben, so erscheint diese im URL-Textfield und die Ausgabe wird automatisch gestartet.

-local=host:{port}: Soll das RTP-Empfänger-Socket an eine bestimmte Adresse und optional an einen bestimmten Port dieser Adresse gebunden werden, so kann dies mit diesem Parameter erreicht werden. Beispiel: -local=phoenix oder -local=gaffel:1234.

+device: Die Ausgabe erfolgt auf das Audio-Device.

-device: Die Ausgabe erfolgt nicht auf das Audio-Device.

+debug: Die Ausgabe erfolgt auf den Audio-Debugger. Er zeigt die Ausgabepuffer-Belegung in regelmäßigen Abständen an, wodurch sich Probleme in den Kodierungen leichter finden lassen.

-debug: Die Ausgabe erfolgt nicht auf den Audio-Debugger.

+analyzer: Die Ausgabe erfolgt auf den Spectrum-Analyzer.

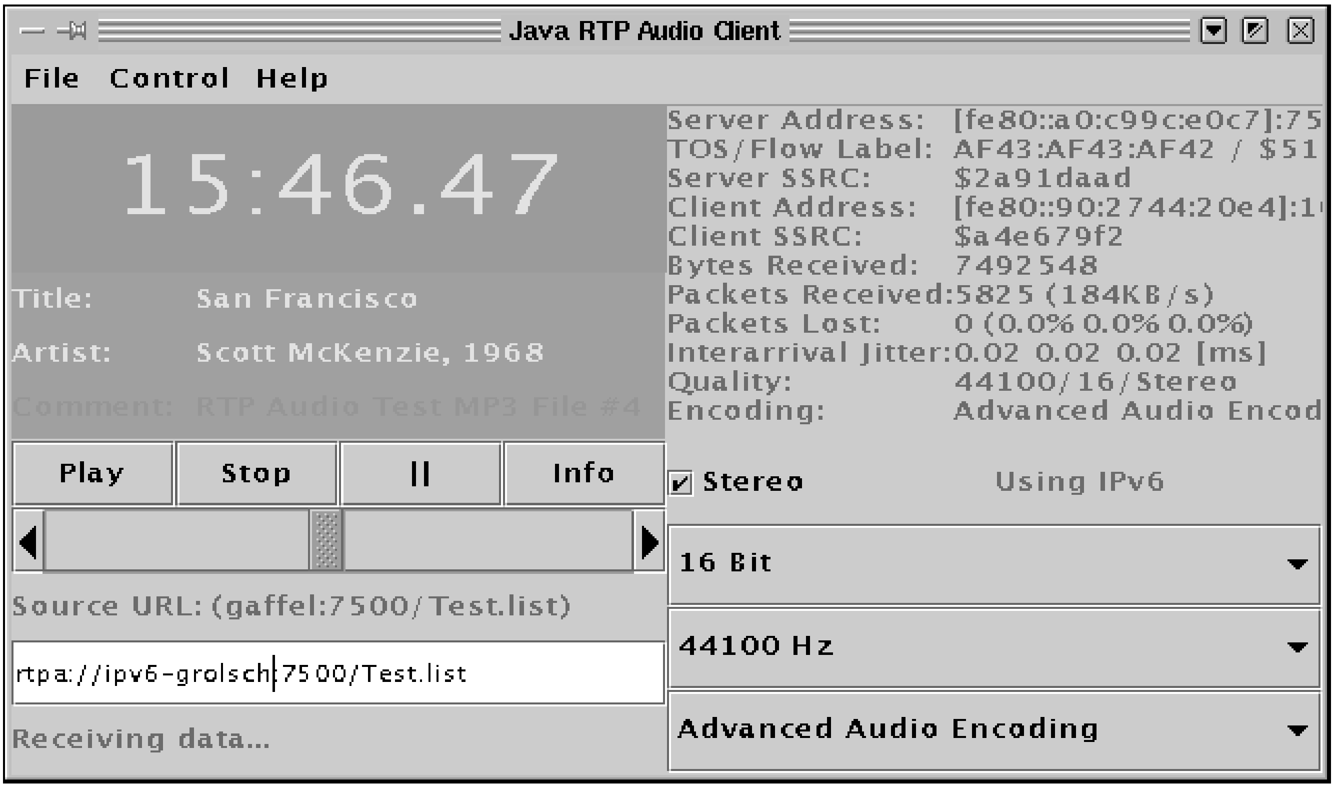


Abbildung A.6: Der Java-Client

-analyzer: Die Ausgabe erfolgt nicht auf den Spectrum-Analyzer. Wird diese Option angegeben, so wird kein SpectrumAnalyzer-Objekt erzeugt. Daher kann der Spectrum Analyzer dann auch nicht mehr im Menü Tools geöffnet werden! Ist dies jedoch gewünscht, sollte man -analyzer weglassen und +device oder +null oder +debug angeben.

+null: Empfangene Daten werden ignoriert (Null-Device).

-null: Empfangene Daten werden nicht ignoriert (Null-Device).

-force-ipv4: Hierdurch wird die Benutzung von IPv6 ausgeschaltet.

Hinweise

- Wird kein Ausgabe-Device angegeben (also +null oder +debug oder +device oder +analyzer), so wird standardmäßig +device +analyzer +mixer angenommen.
- Wird nur +analyzer angegeben (ohne +null oder +debug), so wird +device +mixer angenommen.

A.2.3 AudioClientMain.class - Der RTP Audio Client (Java-Version)

Beschreibung

Dies ist der Java Client (Swing-Oberfläche) für RTP Audio. Er startet eine Verbindung zu einem gegebenen RTP Audio Server und gibt die empfangenen Daten auf das Audio-Device aus.

Als Java-System wurde das [Blackdown] JDK 1.2.2-RC4 verwendet.

Bildschirmfoto

Siehe Abbildung A.6.

Aufruf

java -Djava.library.path=. AudioClientMain {URL}

Parameter

URL: Hiermit können Adresse und Portnummer des RTP Audio Servers sowie der Dateiname der abzuspielenden Audioliste angegeben werden. Beispiel: rtpa://ipv6-odin:7500/Test1.list. Wird eine URL angegeben, so erscheint diese im URL-Textfield und die Ausgabe wird automatisch gestartet.

Hinweise

Der Java Client verwendet JNI für die Anbindung an die C++-Klassen. Daher muß sich im gleichen Verzeichnis die Datei libAudioClient.so (Shared-Library der C++-Klassen) befinden bzw. das Verzeichnis mit -Djava.library.path=pfad dem Java-Interpreter mitgeteilt werden.

A.2.4 vclient - Der RTP Audio Verifikations-Client

Beschreibung

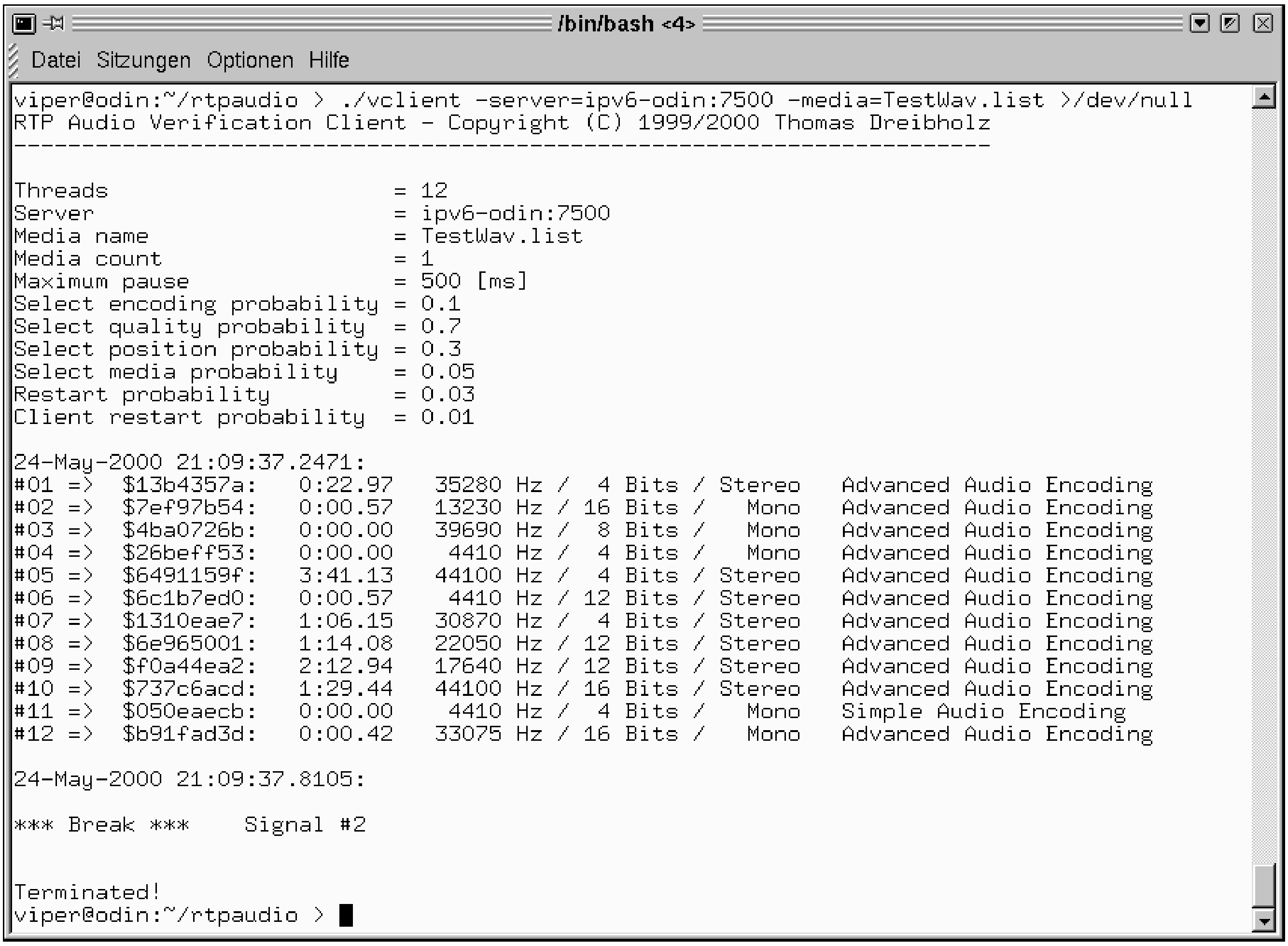
Dieses Programm startet eine gegebene Anzahl von AudioClient-Threads, die mit gegebenen, gleichverteilten Wahrscheinlichkeiten Anfragen an einen Server stellen. Dies kann z.B. dazu verwendet werden, Thread-Synchronisations-Probleme und andere Fehler im Server zu lokalisieren.

Bildschirmfoto

Siehe Abbildung A.7.

Aufruf

./vclient
    {-threads=count}
    {-pause=milliseconds}
    {-server=host:port}



```
/bin/bash <4>
Datei  Sitzungen  Optionen  Hilfe
viper@odin:~/rtpaudio > ./vclient -server=ipv6-odin:7500 -media=TestWav.list >/dev/null
RTP Audio Verification Client - Copyright (C) 1999/2000 Thomas Dreibholz
------------------------------------------------------------------------

Threads                      = 12
Server                       = ipv6-odin:7500
Media name                   = TestWav.list
Media count                  = 1
Maximum pause                = 500 [ms]
Select encoding probability = 0.1
Select quality probability  = 0.7
Select position probability = 0.3
Select media probability    = 0.05
Restart probability          = 0.03
Client restart probability  = 0.01

24-May-2000 21:09:37.2471:
#01 =>  $13b4357a:   0:22.97   35280 Hz /  4 Bits / Stereo   Advanced Audio Encoding
#02 =>  $7ef97b54:   0:00.57   13230 Hz / 16 Bits /   Mono   Advanced Audio Encoding
#03 =>  $4ba0726b:   0:00.00   39690 Hz /  8 Bits /   Mono   Advanced Audio Encoding
#04 =>  $26beff53:   0:00.00    4410 Hz /  4 Bits /   Mono   Advanced Audio Encoding
#05 =>  $6491159f:   3:41.13   44100 Hz /  4 Bits / Stereo   Advanced Audio Encoding
#06 =>  $6c1b7ed0:   0:00.57    4410 Hz / 12 Bits / Stereo   Advanced Audio Encoding
#07 =>  $1310eae7:   1:06.15   30870 Hz /  4 Bits / Stereo   Advanced Audio Encoding
#08 =>  $6e965001:   1:14.08   22050 Hz / 12 Bits / Stereo   Advanced Audio Encoding
#09 =>  $f0a44ea2:   2:12.94   17640 Hz / 12 Bits / Stereo   Advanced Audio Encoding
#10 =>  $737c6acd:   1:29.44   44100 Hz / 16 Bits / Stereo   Advanced Audio Encoding
#11 =>  $050eaecb:   0:00.00    4410 Hz /  4 Bits /   Mono   Simple Audio Encoding
#12 =>  $b91fad3d:   0:00.42   33075 Hz / 16 Bits /   Mono   Advanced Audio Encoding

24-May-2000 21:09:37.8105:

*** Break ***    Signal #2


Terminated!
viper@odin:~/rtpaudio >
```

Abbildung A.7: Der Verifikations-Client

{-media=name with %d}
{-count=media count}
{-se=probability}
{-sq=probability}
{-sp=probability}
{-sm=probability}
{-st=probability}
{-re=probability}

Parameter

-threads=count: Hier wird die Anzahl der Threads angegeben.

-pause=milliseconds: Dies ist die obere Grenze für das Pauseintervall zwischen zwei Aktionen eines Threads. Die Pausezeit wird zufällig aus diesem Intervall gewählt.

-server=host:port: Mit diesem Parameter wird der RTP AUDIO-Server angegeben.

-media=name. Dies ist der Name der abzuspielenden Audiodatei. Der Dateiname kann den Platzhalter "%d" enthalten. Dafür wird dann eine zufällige Zahl zwischen 1 und dem mit "-count=count" angegebenen Wert eingesetzt.

-se=probability: Die Wahrscheinlichkeit (aus [0,1]) für die Wahl einer anderen Kodierung.

-sq=probability: Die Wahrscheinlichkeit (aus [0,1]) für die Wahl einer neuen Qualität.

-sp=probability: Die Wahrscheinlichkeit (aus [0,1]) für die Wahl einer neuen Position.

-sm=probability: Die Wahrscheinlichkeit (aus [0,1]) für die Wahl einer anderen Audiodatei. Damit diese Option Sinn macht, muß der Name in "-media=Name" den Platzhalter "%d" enthalten und Anzahl in "-count=Anzahl" größer 1 sein.

-st=probability: Die Wahrscheinlichkeit (aus [0,1]) für Stop und Neustart.

-re=probability: Die Wahrscheinlichkeit (aus [0,1]) für Stop, Entfernen des *AudioClient*-Objektes und Neustart mit neuem AudioClient-Objekt.

Hinweise

- Um den Server auszutesten, sollten i.d.R. als Eingabedateien WAV-Files statt MP3-Files benutzt werden. Der Rechenzeitaufwand für deren Dekodierung ist wesentlich geringer, insbesondere bei häufigen Positionswechseln!

- Die Ausgabe der zufällig ausgewählten Aktionen erfolgt zur Standard-Ausgabe, in regelmäßigen Abständen wird die aktuelle Position jedes Client-Threads zur Standard-Fehlerausgabe ausgegeben. Um bei großer Thread-Anzahl die Übersicht auf dem Bildschirm zu behalten, ist es i.d.R. sinnvoll, die Standard-Ausgabe in eine Datei (oder auch nach /dev/null) umzulenken. Dann sind nur noch die aktuellen Positionen zu sehen. Beispiel: ./vclient -server=odin:7500 -media=TestWav%d.list -count=6 -threads=20 >/tmp/ausgabe.

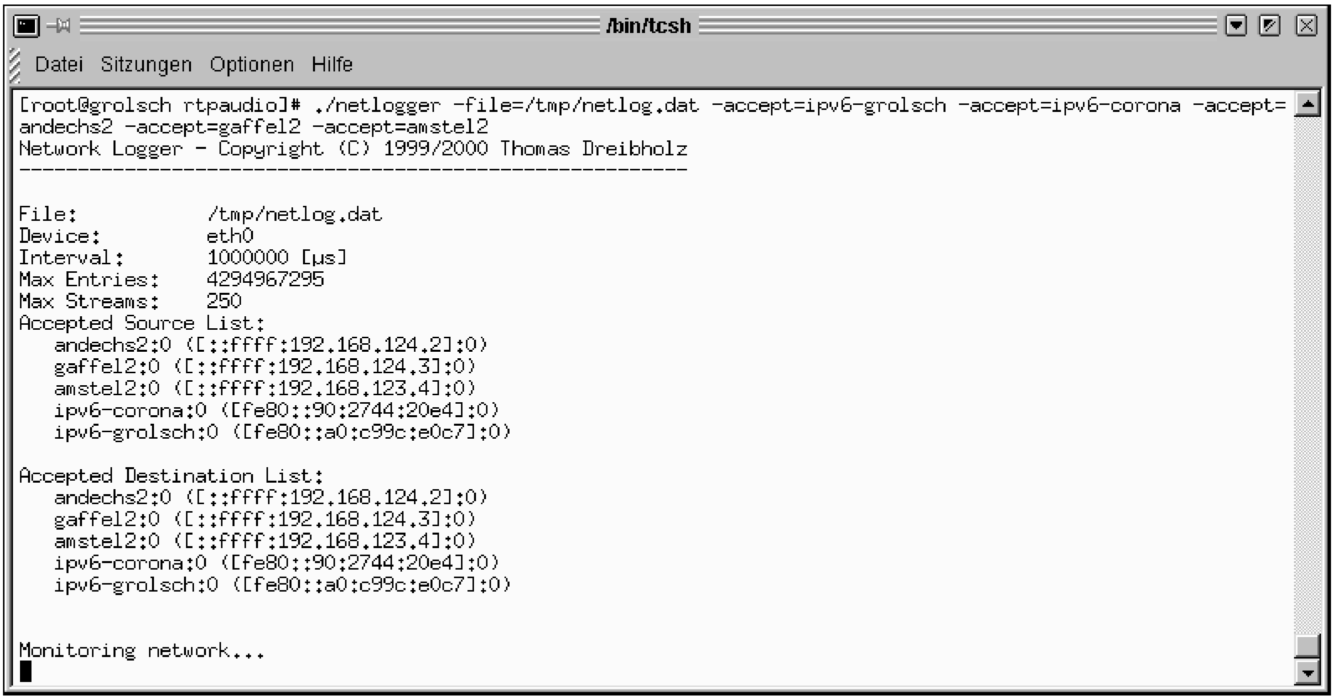


Abbildung A.8: Der NetLogger

A.3 Werkzeuge für die Netzwerk-Leistungsbewertung

A.3.1 NetLogger - Aufzeichnung des Netzwerkverkehrs

Beschreibung

NetLogger zeichnet mittels *libpcap* den Netzwerkverkehr an einem Netzwerk-Device (z.B. *eth0* oder *lo*) auf. Die Aufzeichnung umfaßt Anzahl von Bytes und Paketen pro Intervall. Diese wird, gruppiert nach Protokollen und Traffic Classes, als Summe über alle beobachteten Ströme sowie optional für einzelne ausgewählte oder alle Ströme gespeichert.

Wichtig: Das beobachten des Netzwerkverkehrs erfordert aus Sicherheitsgründen *root*-Rechte, da das Mitschneiden des Netzwerkverkehres sicherheitskritisch ist (z.B. Übertragung unverschlüsselter Passwörter u.ä..).

Bildschirmfoto

Siehe Abbildung A.8.

Aufruf

./netlogger
{device=name}
{-interval=milliseconds}
{-file=name}
{-maxstreams=streams}

{-max=secs}
{-accept=host{:port}}
{-accepts=sourcehost{:port}}
{-acceptd=desthost{:port}}
{-accept-all-sources}
{-accept-all-destinations}
{-print-accepted}
{-print-rejected}

Parameter

-device=name: Hier wird der Name des Netzwerk-Devices angegeben, z.B. *eth0* für die 1. Ethernet-Karte oder *lo* für das Loopback-Device. Beim Loopback-Device ist zu beachten, daß jede Übertragung doppelt gezählt wird: Einmal beim Eingang und einmal beim Ausgang!

-file=name: Der Name der Ausgabedatei wird hier angegeben, z.B. -file=netlog.dat. Zu beachten ist, daß wenn man NetLogger im eigenen NFS-Home-Verzeichnis nach "su"-Aufruf startet, dieses nicht mehr beschreibbar ist (man ist jetzt *root*, also User #0; dieser User hat jedoch über NFS keine Schreibrechte auf das Home-Verzeichnis von User #5xxx). Lösung: z.B. -file=/tmp/netlog.dat.

-interval=milliseconds: Das Aufzeichnungsintervall gibt die Zeit in Millisekunden an, nach der die während des Intervalls gewonnenen Daten in die Logdatei geschrieben werden.

-maxstreams=streams: Hier wird die maximale Anzahl von Streams angegeben, die während eines Intervalls zusätzlich noch einzeln beobachtet werden. Die Angabe von 0 bewirkt, daß keine Einzelbeobachtung stattfindet. In diesem Fall erhält man nur die Summen über alle Ströme.

-max=secs: Hier kann eine Maximalzeit in Sekunden angegeben werden, in der die Beobachtung läuft. Nach dieser Zeit beendet sich NetLogger automatisch.

-accept=host{:port}: Eine Einzelbeobachtung von Strömen findet nur statt, wenn sich die Quell- und Zieladresse in der Accepted-Sources bzw. Accepted-Destinations-Liste befindet. Dieser Parameter fügt eine Adresse in beide Listen ein. Die optionale Angabe einer Portnummer bewirkt, daß nur Ströme mit dieser Portnummer akzeptiert werden (falls das Protokoll UDP bzw. TCP ist); das Weglassen bzw. die Angabe 0 implizieren das Akzeptieren aller Ströme mit angegebener Adresse und beliebiger Portnummer. Beispiele: -accept=ipv6-odin:1234 -accept=1.2.3.4:9999 -accept=[3ffe:4711:0815::111]:9876.

-accepts=host{:port}: Analog zu -accept mit Accepted-Sources-Liste.

-acceptd=host{:port}: Analog zu -accept mit Accepted-Destinations-Liste.

-accept-all-sources: Alle Quell-Adressen werden akzeptiert.

-accept-all-destinations: Alle Ziel-Adressen werden akzeptiert.

-print-accepted: Alle akzeptierten Ströme werden während des Aufzeichnens ausgegeben (Quell- und Zieladresse sowie Flowlabel und Traffic Class). Jeder akzeptiere Strom wird nur einmal pro Intervall ausgegeben!
Wichtiger Hinweis: -print-accepted und -print-rejected (nächster Parameter) sind nur für Testzwecke gedacht, um die Accepted-Listen einzustellen. Bei vielen akzeptierten Strömen und kleinem Intervall ist es empfehlenswert, diese Optionen für die eigentliche Messung wieder auszuschalten (wegen CPU-Verbrauch). Dies gilt insbesondere dann, wenn die Adressen zur Auflösung in Hostnamen einen Nameserver-Zugriff erfordern!

-print-rejected: Alle nicht akzeptierten Ströme werden während des Aufzeichnens ausgegeben (Quell- und Zieladresse sowie Flowlabel und Traffic Class).
Achtung: NetLogger versucht, die Adresse bei der Ausgabe in einen Hostnamen aufzulösen. Dies kann einen Nameserver-Zugriff erfordern. Dieser - je nach Netzwerk-Device - ebenfalls aufgezeichnet wird. Wird die Adresse des Nameservers ebenfalls nicht akzeptiert, so wird versucht, diese evtl. mittels Nameserver-Zugriff aufzulösen usw.. Daraus folgt dann eine endlose Lawine von Nameserver-Anfragen! Lösungen:

- NetLogger abbrechen und entweder Adressen für Nameserver-Anfragen akzeptieren oder
- die Hosts der Nameserver-Anfragen in /etc/hosts eintragen oder
- nur -print-accepted verwenden.

A.3.2 NetAnalyzer - Auswertung des Netzwerkverkehrs

Beschreibung

Der NetAnalyzer wertet die mit NetLogger gewonnenen Daten aus und erzeugt daraus GNU PLOT-Ausgaben.

Bildschirmfoto

Siehe Abbildung A.9.

Aufruf

./netanalyzer
  [Input file]
  [Output prefix]
  {-interval=milliseconds}
  {-info=infostring}
  {[+/-]{ |pkt|len}tcp}
  {[+/-]tcpdata}
  {[+/-]{ |pkt|len}udp}
  {[+/-]udpdata}
  {[+/-]{ |pkt|len}icmp4}
  {[+/-]{ |pkt|len}icmp6}

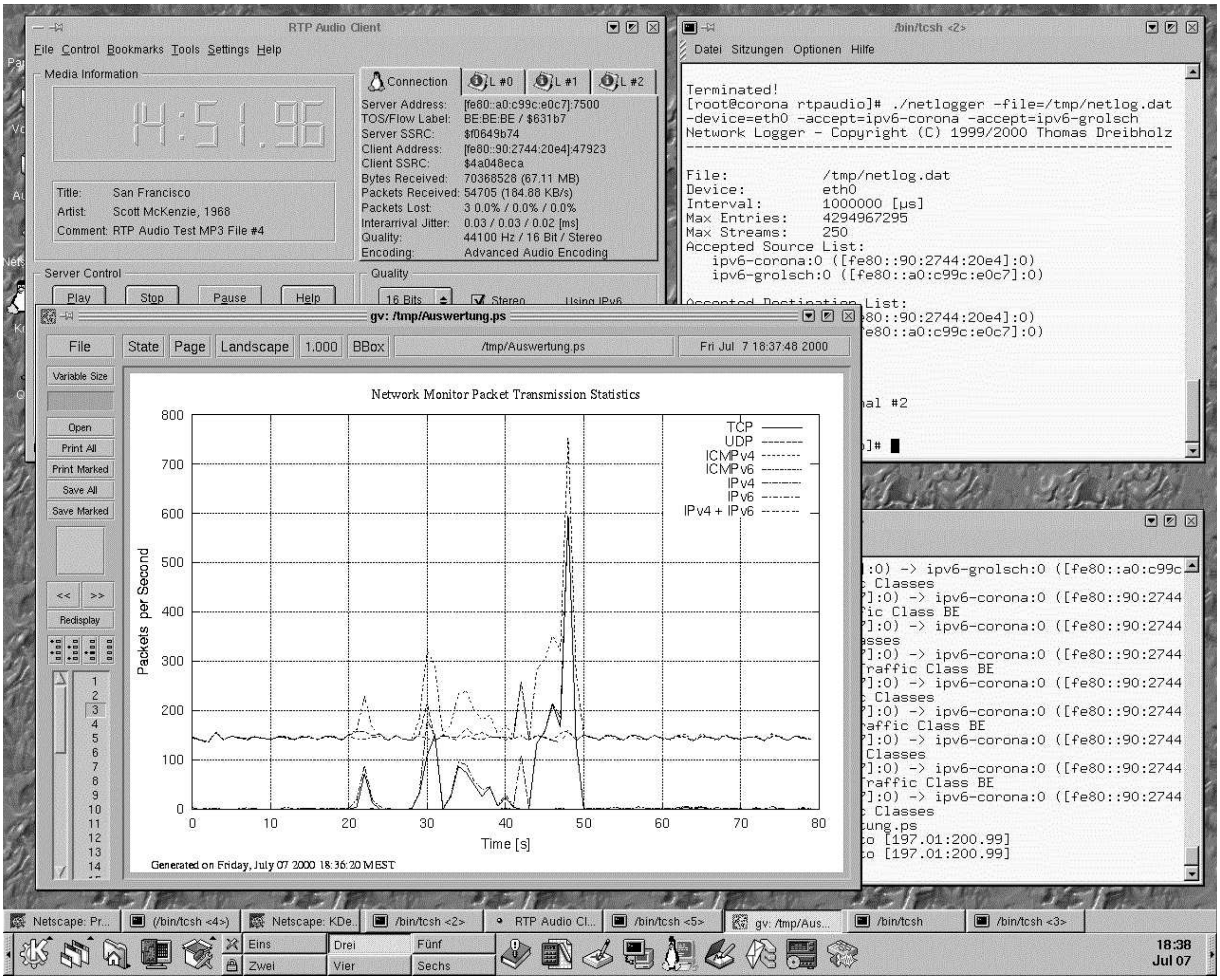


Abbildung A.9: Eine Messung und Auswertung mit NetLogger und NetAnalyzer

{[+/-]{ |pkt|len}ip}
{[+/-]{ |pkt|len}ipv4}
{[+/-]{ |pkt|len}ipv6}
{[+/-]{ |pkt|len}classes}
{[+/-]{ |pkt|len}streams}
{[+/-]streamsraw}
{[+/-]streamspayload}
{[+/-]address}
{[+/-]hostname}
{[+/-]ownpage}
{[+/-]ownpagestreams}
{[+/-]autoscale}
{+all}
{-force-ipv4}

Parameter

Input file: Dies ist der Name der Ausgabedatei von NetLogger, z.B. `/tmp/netlog.dat`.

Output prefix: Das Dateinamen-Prefix für die Ausgabedateien, z.B. `/tmp/auswertung`. NetAnalyzer erzeugt dann `/tmp/auswertung.gp` (GNU PLOT-Skript) sowie eine GNU PLOT Datendatei für die Daten aller Ströme (Summen) und für jeden beobachteten Einzelstrom eine eigene Datendatei mit Dateinamen `/tmp/auswertung-*.data`.

-interval=milliseconds: Das Intervall für die Anzeige von Bytes/Intervall und Paketen/Intervall kann hier in Millisekunden gesetzt werden. Default ist 1000 (= eine Sekunde, also Bytes/Sekunde und Pakete/Sekunde).

-info=infostring: Hier kann ein optionaler Informationstext für die GNU PLOT-Ausgabe angegeben werden. Dieser erscheint dann hinter dem Datum.

[+/-]{ |pkt|len}tcp: Anzeige der Zusammenfassung aller TCP-Übertragungen (Header + Payload) in Bytes pro Intervall (+tcp), Paketen pro Intervall (+pkttcp) oder durchschnittlicher TCP-Paketlänge (+lentcp) bzw. Ausschalten dieser Anzeige durch “-“ statt “+”.

[+/-]tcpdata: Anzeige der Zusammenfassung aller TCP-Übertragungen (nur Payload) in Bytes pro Intervall (+tcpdata) bzw. Ausschalten dieser Anzeige durch “-“ statt “+”.

[+/-]{ |pkt|len}udp: Anzeige der Zusammenfassung aller UDP-Übertragungen (Header + Payload) in Bytes pro Intervall (+udp), Paketen pro Intervall (+pktudp) oder durchschnittlicher UDP-Paketlänge (+lenudp) bzw. Ausschalten dieser Anzeige durch “-“ statt “+”.

[+/-]udpdata: Anzeige der Zusammenfassung aller UDP-Übertragungen (nur Payload) in Bytes pro Intervall (+udpdata) bzw. Ausschalten dieser Anzeige durch “-“ statt “+”.

[+/-]{ |pkt|len}icmp4: Anzeige der Zusammenfassung aller ICMP4-Übertragungen (Header + Payload) in Bytes pro Intervall (+icmp4), Paketen pro Intervall (+pkticmp4) oder durchschnittlicher ICMPv4-Paketlänge (+lenicmp4) bzw. Ausschalten dieser Anzeige durch "-" statt "+".

[+/-]{ |pkt|len}icmp6: Anzeige der Zusammenfassung aller ICMP6-Übertragungen (Header + Payload) in Bytes pro Intervall (+icmp6), Paketen pro Intervall (+pkticmp6) oder durchschnittlicher ICMPv6-Paketlänge (+lenicmp6) bzw. Ausschalten dieser Anzeige durch "-" statt "+".

[+/-]{ |pkt|len}ip: Anzeige der Zusammenfassung aller IP-Übertragungen (IPv4 oder IPv6 mit Header + Payload) in Bytes pro Intervall (+ip), Paketen pro Intervall (+pktip) oder durchschnittlicher IPv4- oder IPv6-Paketlänge (+lenip) bzw. Ausschalten dieser Anzeige durch "-" statt "+".

[+/-]{ |pkt|len}ipv4: Anzeige der Zusammenfassung aller IPv4 Übertragungen (Header + Payload) in Bytes pro Intervall (+ipv4), Paketen pro Intervall (+pktipv4) oder durchschnittlicher IPv4-Paketlänge (+lenipv4) bzw. Ausschalten dieser Anzeige durch "-" statt "+".

[+/-]{ |pkt|len}ipv6: Anzeige der Zusammenfassung aller IPv6 Übertragungen (Header + Payload) in Bytes pro Intervall (+ipv6), Paketen pro Intervall (+pktipv6) oder durchschnittlicher IPv6-Paketlänge (+lenipv6) bzw. Ausschalten dieser Anzeige durch "-" statt "+".

[+/-]{ |pkt|len}classes: Anzeige aller Übertragungen (Header + Payload) gruppiert nach Traffic Class in Bytes pro Intervall (+classes), Paketen pro Intervall (+pktclasses) oder durchschnittlicher Paketlänge (+lenclasses) bzw. Ausschalten dieser Anzeige durch "-" statt "+".

[+/-]streamsraw: Anzeige der Übertragungen (Header incl. IP-Header + Payload) je einzeln beobachtetem Stream in Bytes pro Intervall (+streamsraw). Ausschalten dieser Anzeige durch "-" statt "+".

[+/-]streamspayload: Anzeige der Übertragungen (nur Payload) je einzeln beobachtetem Strom in Bytes pro Intervall bzw. Ausschalten dieser Anzeige durch "-" statt "+".

[+/-]{ |pkt|len}streams: Anzeige der Übertragungen (Header incl. IP-Header + Payload sowie nur Payload) in Bytes pro Intervall (+streams, entspricht +streamsraw +streamspayload), Paketen pro Intervall (+pktstreams) oder durchschnittlicher Gesamt-Paketlänge (+lenstreams) bzw. Ausschalten dieser Anzeige durch "-" statt "+".

+all: Alle diese Ausgabe-Optionen werden eingeschaltet.

[+/-]address: Anzeige von IP-Adresse ein- bzw. ausschalten. Standardmäßig werden sowohl Hostname (falls auflösbar) als auch IP-Adresse angezeigt. Ist weder Hostname noch Adresse eingeschaltet, wird nur die Adresse ausgegeben.

[+/-]hostname: Anzeige von Hostname ein- bzw. ausschalten. Standardmäßig werden sowohl Hostname (falls auflösbar) als auch IP-Adresse angezeigt. Ist weder Hostname noch Adresse eingeschaltet, so wird nur die Adresse ausgegeben. **Wichtig:** Die Hostname-Ausgabe sollte unbedingt abgeschaltet werden, wenn die Analyse nicht im gleichen Netzwerk wie die NetLogger-Aufnahme durchgeführt wird (lokale Adressen in verschiedenen Netzen haben i.d.R. verschiedene Hostnamen).

[+/-]ownpage: Eigene Seite für jede Zusammenfassungsausgabe ein- bzw. ausschalten. Default: aus.

[+/-]ownpagestreams: Eigene Seite für jede Stream-Ausgabe ein- bzw. ausschalten. Default: aus.

[+/-]autoscale: Ist AutoScale ausgeschaltet (default), wählt GNU PLOT bei der Ausgabe für jede Seite die bestmögliche Skalierung. Um Ausgaben miteinander vergleichen zu können, ist es u.U. sinnvoll, z.B. für alle "Bytes pro Intervall"-Werte die gleiche Skalierung zu verwenden. Dies geschieht durch Ausschalten von AutoScale (-autoscale). In diesem Fall ist eine Ausgabe geringer Bandbreite (z.B. 500 Bytes/s) bei einer Skalierung für eine große Bandbreite (z.B. 1 MByte/s) allerdings kaum noch zu erkennen.

-force-ipv4: Hierdurch wird die Benutzung von IPv6 ausgeschaltet. Reine IPv6-Adressen werden in der Ausgabe jedoch weiterhin angezeigt!

A.3.3 rttp - Round Trip Time Pinger

Der Round TripTime Pinger mißt die Roundtripzeit zu gegebenen Hosts mit gegebener Traffic Class mittels ICMPv4 bzw. ICMPv6 Echo-Requests und Echo-Replys und gibt diese auf dem Bildschirm und optional als GNU PLOT-Ausgabe aus. **Wichtig**: Der Zugriff auf ICMPv4 bzw. ICMPv6-Sockets erfordert *root*-Rechte.

Bildschirmfoto

Siehe Abbildung A.10 (Ergebnis einer Round Trip Time-Messung).

Aufruf

./rttp
{-alpha=value}
{-file=prefix}
{-delay=milliseconds}
{-info=string}
[Host{,Traffic Class{,TrafficClas,...}}]
{Host{,Traffic Class{,TrafficClas,...}}}
...

Parameter

-alpha=value: Hier kann die Konstante α für die Alterungsfunktion der Round Trip Time-Berechnung gesetzt werden:

$$RTT_{Neu} := \alpha * RTT_{Alt} + (1 - \alpha) * \vartheta$$

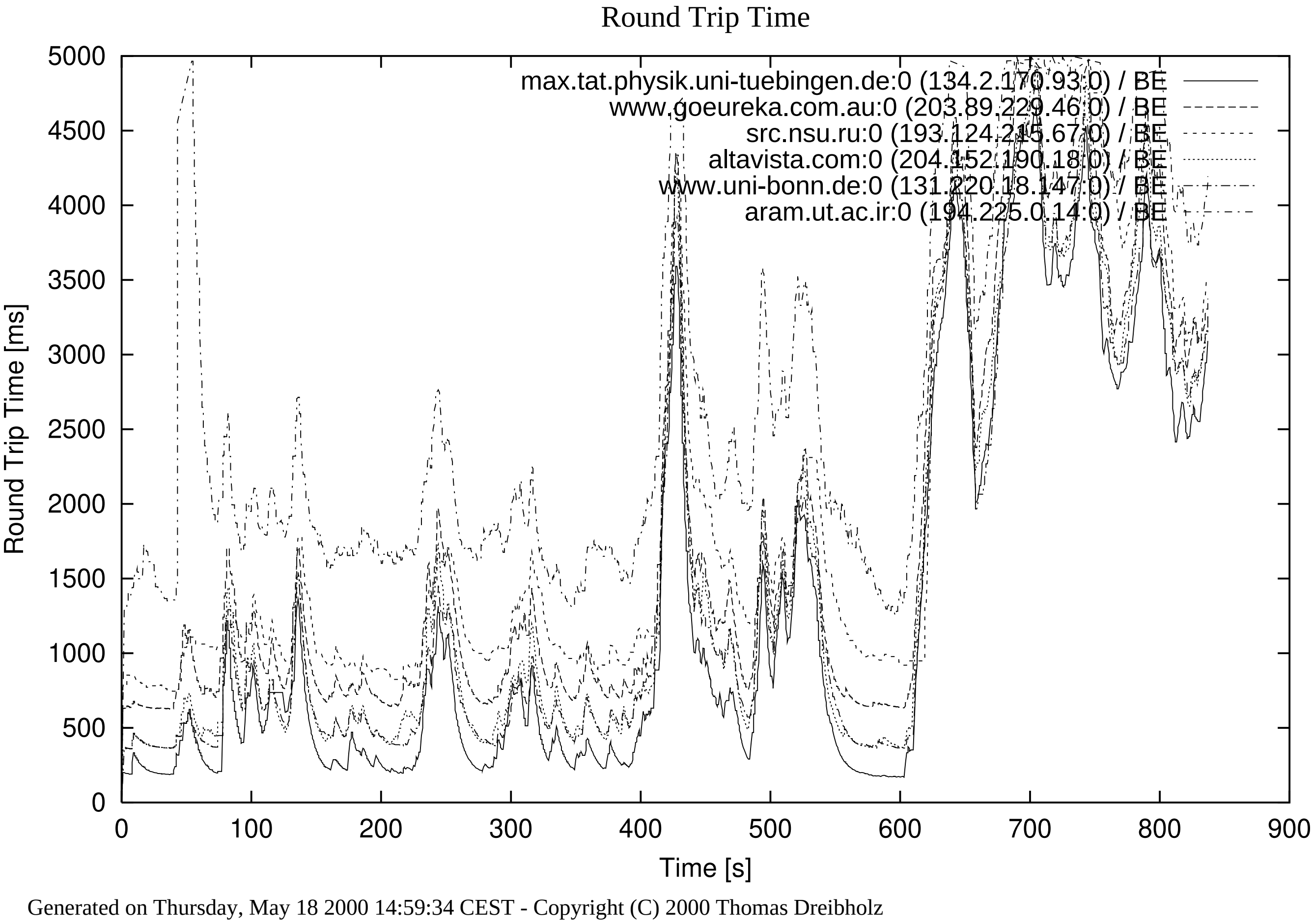


Abbildung A.10: Eine Round Trip Time-Messung zu verschiedenen Rechnern im Internet

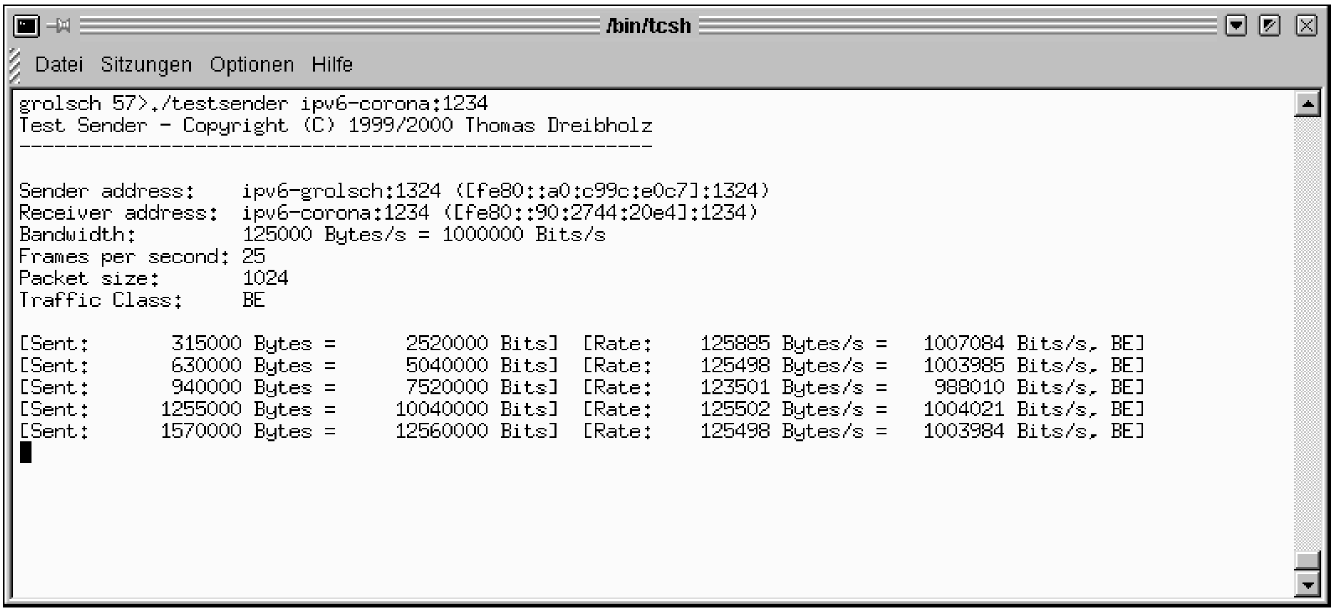


Abbildung A.11: Der UDP-Testsender

Der Default-Wert für α ist $\frac{7}{8}$ (wie bei TCP).

-file=prefix: Dateinamen-Prefix für die GNU Plot-Ausgabe. `prefix.gp` ist das GNU Plot-Skript, `prefix.dat` ist die Datei mit den GNU Plot-Daten.

-delay=milliseconds: Hiermit kann die maximale Zeit zwischen zwei Echo Requests in Millisekunden gesetzt. Das eigentliche Intervall wird immer zufällig gewählt aus [1, Delay] gewählt.

-info=string: Hier kann ein zusätzlicher Informationstext für die Datumszeile im GNU Plot-Skript gesetzt werden.

Host: Hostname und Liste von Traffic Classes, für die Roundtripzeiten berechnet werden sollen. Eine Traffic Class kann hier durch Hexadezimal-Wert bzw. Namen angegeben werden. Mögliche Namen sind BE, EF, AF11, AF12, AF13, AF21, AF22, AF23, AF31, AF32, AF33, AF41, AF42, AF43.
Beispiel: "odin,EF,AF11,AF21,BE".

A.3.4 TestSender - UDP-Sender für Testdaten

Beschreibung

TestSender sendet UDP-Daten mit konstanter Datenrate bei konstanter Anzahl von Frames pro Sekunde und Bytes pro Paket zu einer angegebenen Adresse über eine festgelegte Traffic Class.

Bildschirmfoto

Siehe Abbildung A.11.

Aufruf

./testsender
[Host:Port]
{-bits=bits/s}|{-bytes=bytes/s}
{-tc=traffic class}
{-frames=fps}
{-pktsize=bytes}
{-force-ipv4}

Parameter

[Host:Port]: Hier werden Zielrechner und Zielport angegeben, z.B. gaffel:1234.

-bits=bits/s: Hiermit wird eine Payload-Datenrate in Bits pro Sekunde angegeben.

-bytes=bytes/s: Hiermit wird eine Payload-Datenrate in Bytes pro Sekunde angegeben.

-tc=traffic class: Eine Traffic Class kann hier durch Hexadezimal-Wert bzw. Namen angegeben werden. Mögliche Namen sind BE, EF, AF11, AF12, AF13, AF21, AF22, AF23, AF31, AF32, AF33, AF41, AF42, AF43.

-frames=fps: Die Anzahl der Frames pro Sekunde kann durch diesen Parameter gesetzt werden.

-pktsize=bytes: Dies ist die Paketgröße in Bytes.

-force-ipv4: Hierdurch wird die Benutzung von IPv6 ausgeschaltet.

A.3.5 TestReceiver - UDP-Empfänger für Testdaten

Beschreibung

TestReceiver empfängt UDP-Daten auf einem angegebenen Port und gibt die empfangene Payload-Datenrate aus.

Bildschirmfoto

Siehe Abbildung A.12.

Aufruf

./testreceiver
[Port]
{-force-ipv4}

Parameter

Port: Dies ist der Port, auf dem die Daten empfangen werden sollen.

-force-ipv4: Hierdurch wird die Benutzung von IPv6 ausgeschaltet.

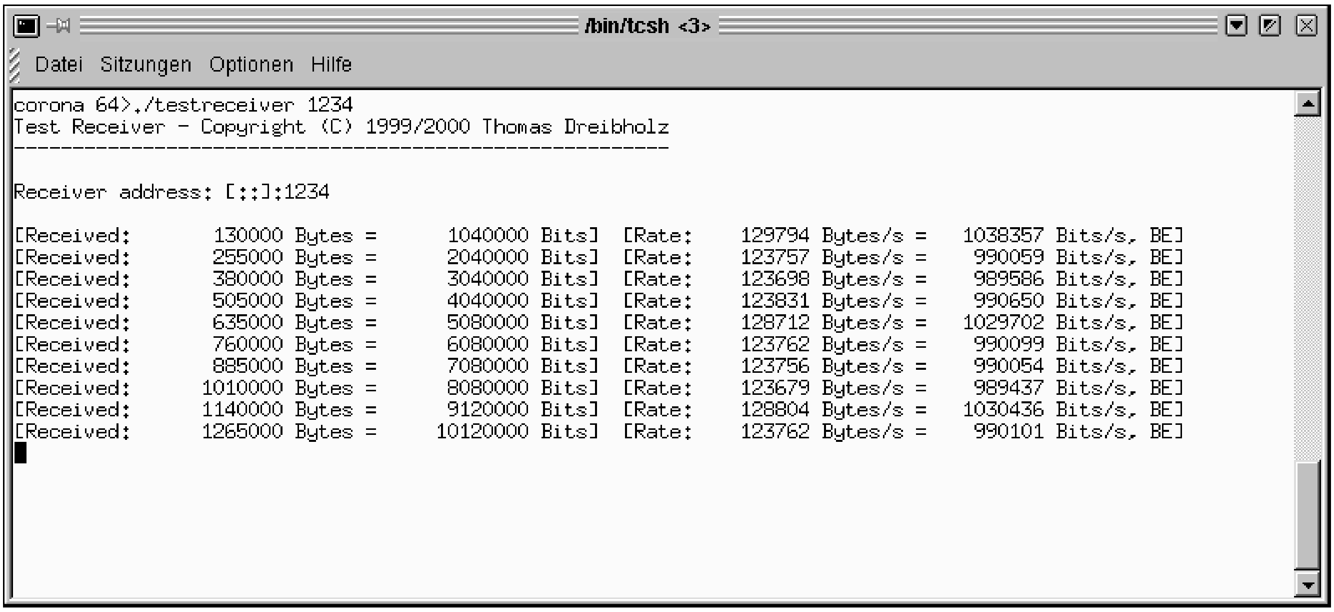


Abbildung A.12: Der UDP-Testreceiver

A.3.6 TCPTestSender - TCP-Sender für Testdaten

Beschreibung

TCPTestSender baut eine TCP-Verbindung auf und sendet Daten mit maximaler Datenrate an einen TCP-Empfänger über eine festgelegte Traffic Class.

Bildschirmfoto

Siehe Abbildung A.13.

Aufruf

./tcptestsender
    [Host:Port]
    {-tc=traffic class}

Parameter

- [Host:Port]: Hier werden Zielrechner und Zielport angegeben, z.B. gaffel:1234.
- -tc=traffic class: Eine Traffic Class kann hier durch Hexadezimal-Wert bzw. Namen angegeben werden. Mögliche Namen sind BE, EF, AF11, AF12, AF13, AF21, AF22, AF23, AF31, AF32, AF33, AF41, AF42, AF43.
- -force-ipv4: Hierdurch wird die Benutzung von IPv6 ausgeschaltet.

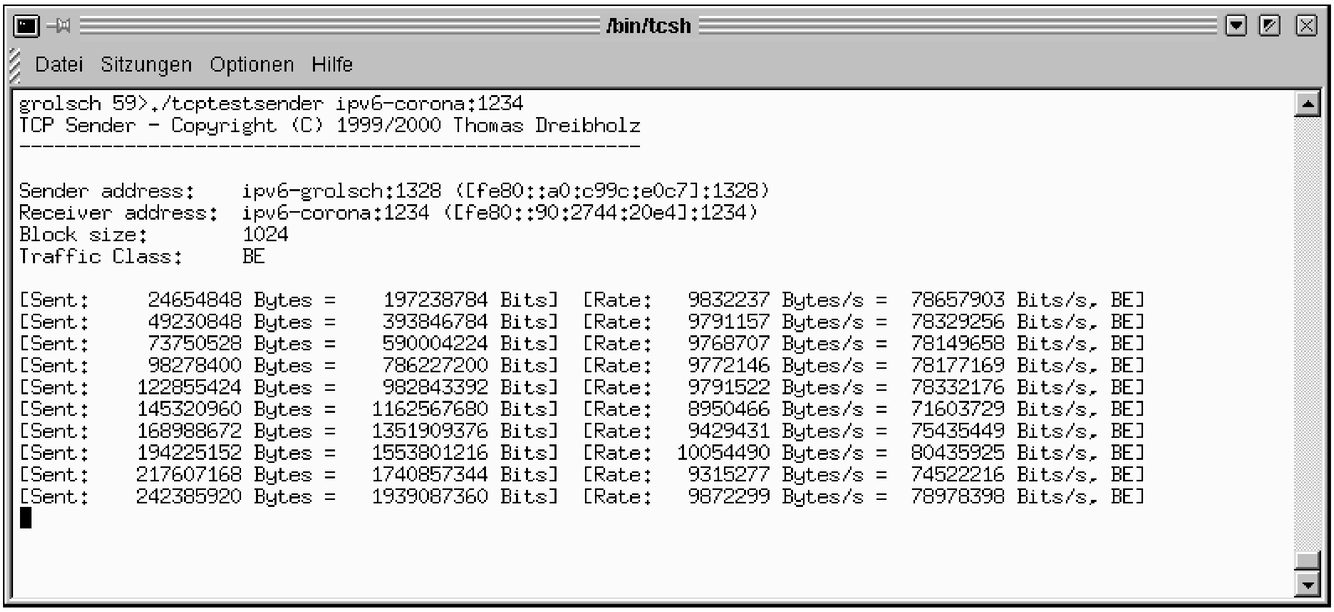


Abbildung A.13: Der TCP-Testsender

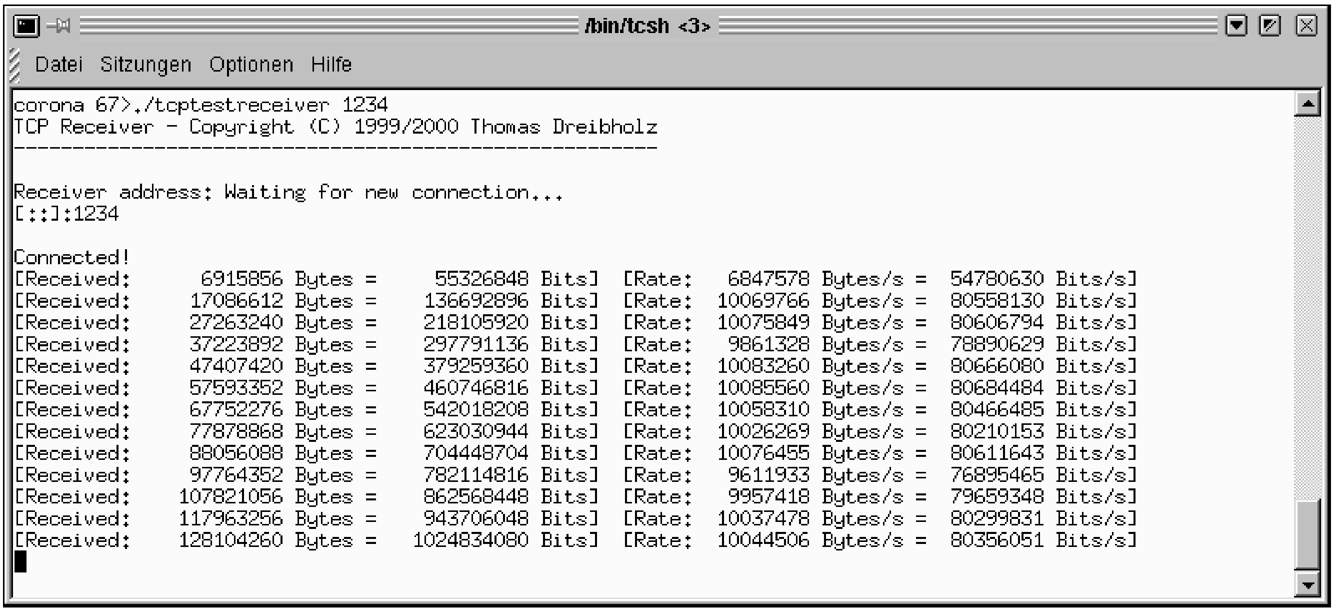


Abbildung A.14: Der TCP-Testreceiver

A.3.7 TCPTestReceiver - TCP-Empfänger für Testdaten

Beschreibung

TCPTestReceiver empfängt TCP-Daten auf einem angegebenen Port und gibt die empfangene Datenrate aus.

Bildschirmfoto

Siehe Abbildung A.14.

Aufruf

./tcptestreceiver
  [Port]
  {-force-ipv4}

Parameter

- Port: Dies ist der Port, an welchem auf TCP-Verbindungen gewartet werden soll.
- -force-ipv4: Hierdurch wird die Benutzung von IPv6 ausgeschaltet.

A.4 Das libpcap Congestion Management

A.4.1 cmgr - Der Congestion Manager

Beschreibung

cmgr ist der Congestion Manager für das *libpcap*-Congestion-Management. Ein Congestion Monitor beobachet mittels libpcap das Netzwerk und berechnet die Congestion. Diese Daten werden zum Congestion Manager gesendet, welcher die vorhandene Bandbreite unter den bei ihm angemeldeten Strömen aufteilt.

Um den Congestion Manager mit RTP Audio zu verwenden, muß am RTP Audio Server mit dem Parameter -manager=host:port die Adresse des Congestion Managers gesetzt werden.

Bildschirmfoto

Siehe Abbildung A.15.

Aufruf

./cmgr
  [-monitor=host:port]
  {-port=port}
  {-local=host{:port}}

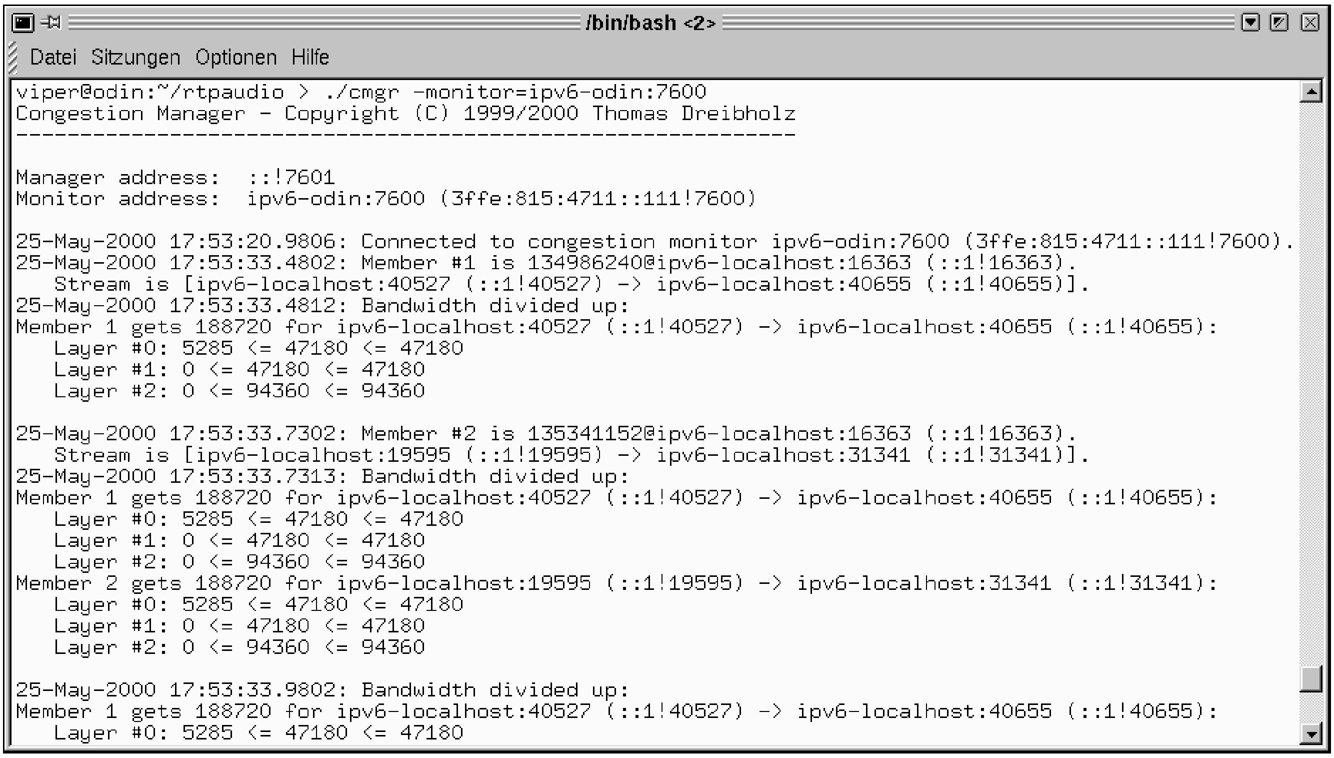


Abbildung A.15: Der Congestion Manager

Parameter

-monitor=host:port: Hier wird die Adresse des Congestion Monitors angegeben.

-port=port: Hier kann eine Portnummer für den Congestion Manager angegeben werden, an welcher er Anfragen von Clients erwartet. Der Default-Wert ist 7601.

-local=host:{port}: Soll das Empfänger-Socket an eine bestimmte Adresse und optional an einen bestimmten Port dieser Adresse gebunden werden, so kann dies mit diesem Parameter erreicht werden. Beispiel: -local=phoenix oder -local=gaffel:7601.

A.4.2 cmon - Der Congestion Monitor

Beschreibung

cmon ist der Congestion Monitor für das *libpcap*-Congestion-Management. Er beobachet mittels *libpcap* das Netzwerk und berechnet die Congestion. Diese Daten werden zum Congestion Manager gesendet, welcher die vorhandene Bandbreite unter den bei ihm angemeldeten Strömen aufteilt.

Bildschirmfoto

Siehe Abbildung A.16.

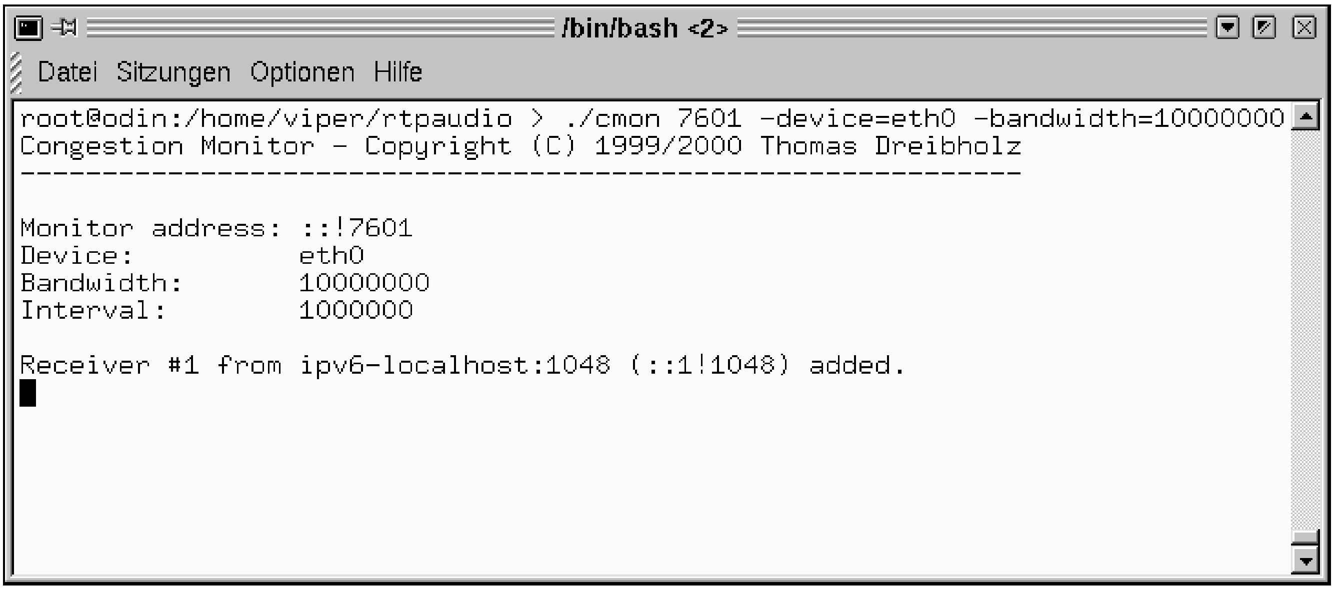


Abbildung A.16: Der Congestion Monitor

Aufruf

./cmon
  [Port]
  {-device=name}
  {-bandwidth=bits/s}
  {-interval=ms}
  {-local=hostname}

Parameter

Port: Dies ist die TCP-Portnummer des Congestion Monitors.

-device=name: Hier wird das mittels *libpcap* zu beobachtende Device angegeben, z. B. -device=eth0.

-bandwidth=bits/s: Hiermit wird die maximale Bandbreite des Devices gesetzt, z. B. -bandwidth=10000 für 10 MBit/s.

-interval=ms: Dies setzt das Update-Intervall für die Congestion Reports. Default ist 1000 ms.

-local=host:{port}: Soll das Empfänger-Socket an eine bestimmte Adresse und optional an einen bestimmten Port dieser Adresse gebunden werden, so kann dies mit diesem Parameter erreicht werden. Beispiel: -local=phoenix oder -local=gaffel:7600.

A.4.3 csim - Der Congestion Monitor Simulator

Beschreibung

csim ist eine Simulation des Congestion Monitors. Die Congestion ist dabei mittels eines Sliders (Qt-GUI) einstellbar.

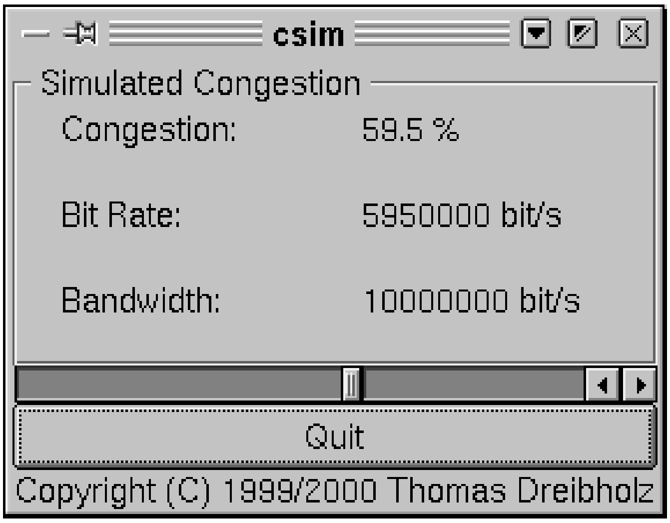


Abbildung A.17: Der Congestion Monitor Simualtor

Bildschirmfoto

Siehe Abbildung A.17.

Aufruf

./csim
  [Port]
  {-bandwidth=bits/s}
  {-interval=ms}
  {-local=hostname}

Parameter

Port: Dies ist die TCP-Portnummer des Congestion Monitor Simulators.

-bandwidth=bits/s: Hiermit wird die maximale Bandbreite des simulierten Devices gesetzt, z. B. -bandwidth=10000000 für 10 MBit/s.

-interval=ms: Dies setzt das Update-Intervall für die Congestion Reports. Default ist 1000 ms.

-local=host:{port}: Soll das Empfänger-Socket an eine bestimmte Adresse und optional an einen bestimmten Port dieser Adresse gebunden werden, so kann dies mit diesem Parameter erreicht werden. Beispiel: -local=phoenix oder -local=gaffel:7600.

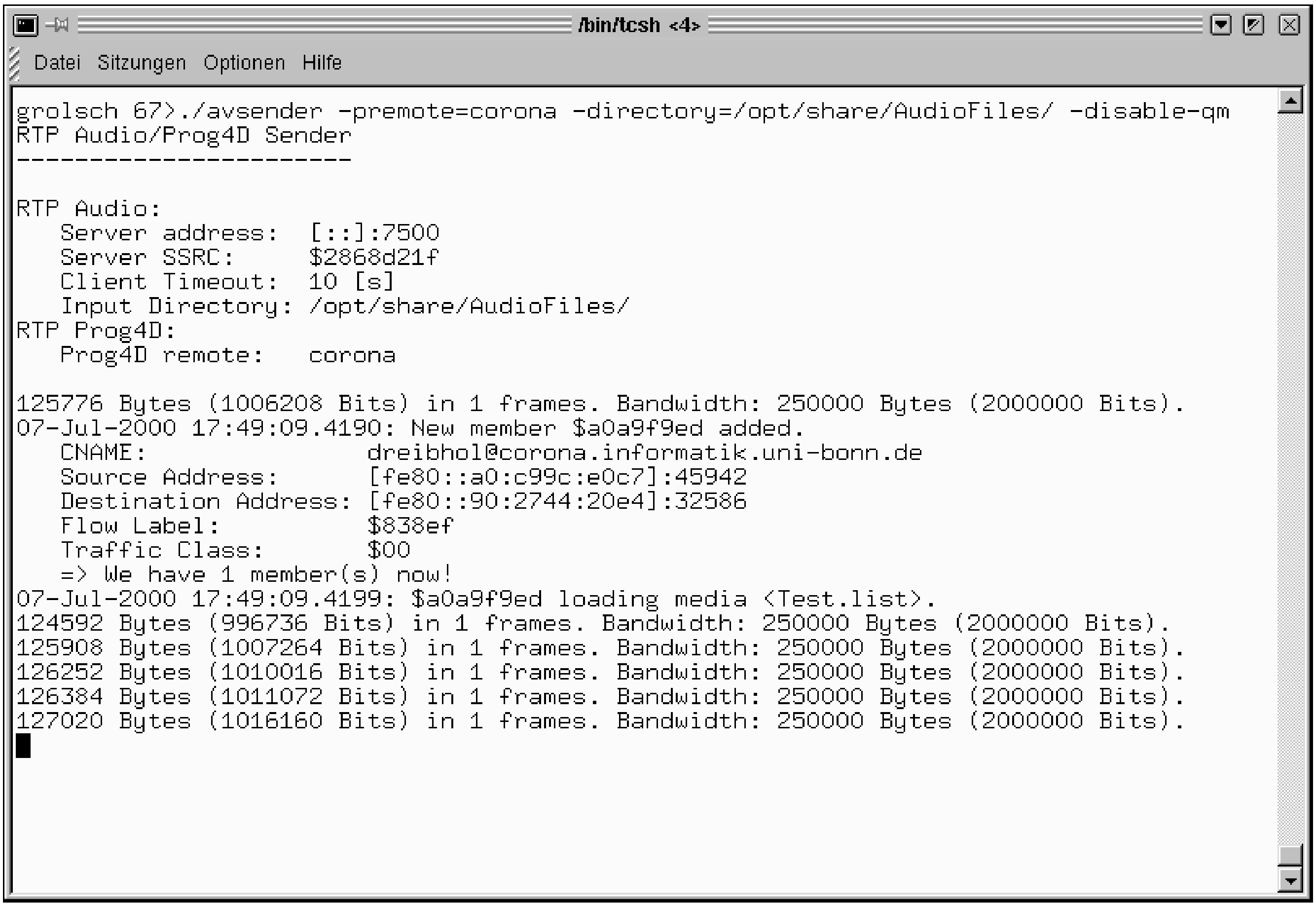


Abbildung A.18: Der Prog4D/RTP Audio-Sender

A.5 Die Verbindung von RTP Audio mit Prog4D

A.5.1 AVSender - Die RTP Audio/Prog4D-Server

Beschreibung

AVSender ist eine Kombination vom RTP Audio Server und dem Prog4D Server. Die folgende Beschreibung geht nur auf die Unterschiede zum RTP Audio Server ein, der bereits in Abschnitt A.1.1 beschrieben wird.

Der Prog4D-Server sendet zu genau einem vorher mit dem Parameter -premote festgelegten Client, wobei die lokale Portnummer 5000 und die Remote-Portnummer 6000 fest eingestellt sind.

Bildschirmfoto

Siehe Abbildung A.18.

Aufruf

./avsender
RTP Audio Parameter:
{-port=port}
{-directory=path}
{-manager=host:port}
{-local=host}
{-timeout=secs}
{-disable-qm}
{-force-ipv4}
Prog4D Parameter:
{-premote=host}

Parameter

-premote=host: Hostname des Remote-Hosts für den Prog4D-Server. Der Default-Wert ist localhost.

Alle anderen Parameter gehören zum RTP Audio Server und haben die gleiche Bedeutung wie im Abschnitt A.1.1 beschrieben.

A.5.2 MainControl.class - Bandbreite-Steuerung für Prog4D Client und RTP Audio Clients

Beschreibung

MainControl erlaubt das Setzen einer gemeinsamen Gesamt-Bandbreite für einen Prog4D-Client und bis zu 4 RTP Audio Clients. Die Clients können mittels der Start/Stop-Buttons ein- und ausgeschaltet werden.

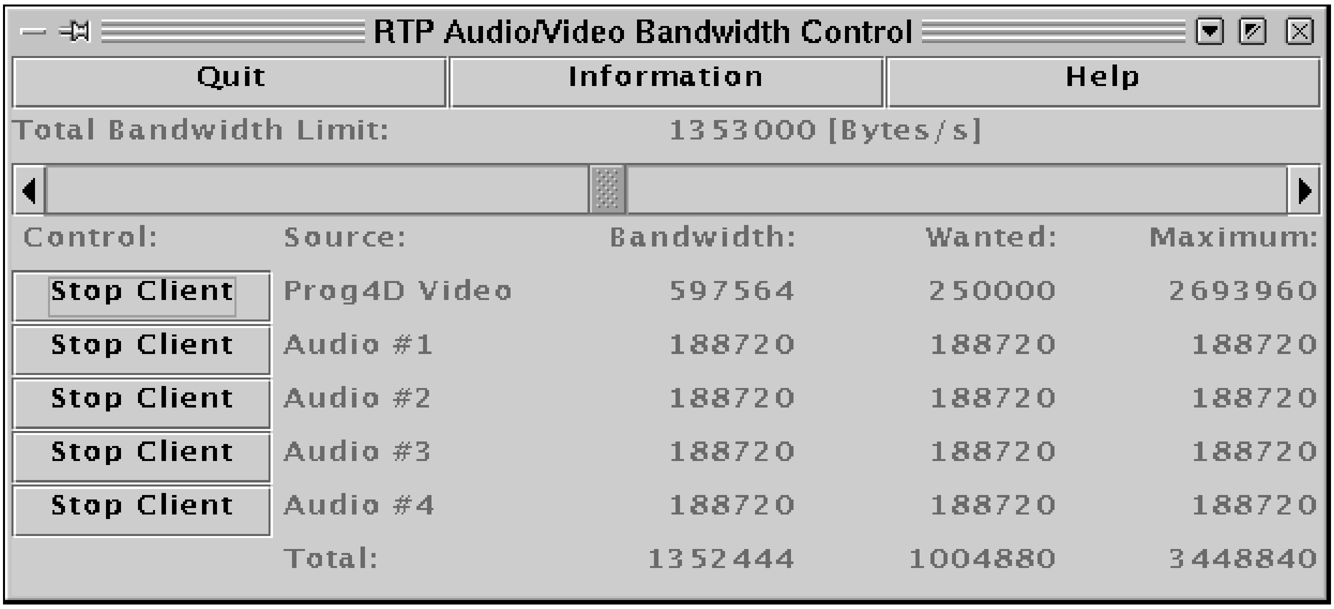


Abbildung A.19: MainControl - Steuerung von einem PROG4D- und bis zu vier RTP AUDIO Clients

Als Java-System wurde das [Blackdown] JDK 1.2.2-RC4 verwendet.

Bildschirmfotos

Siehe Abbildungen A.19 (MainControl) und A.20 (Arbeit mit MainControl).

Aufruf

java -classpath .:../../prog4d-2.0 -Djava.library.path=. MainControl {[Audio URL] [Video Location]}

Parameter

Audio URL: Hier wird die Default-URL für RTP AUDIO angegeben. Beispiel: z.B. rtpa://odin:7500/CD

Video Location: Hiermit wird der Hostname für den PROG4D-Server angegeben, der Standardwert ist localhost.

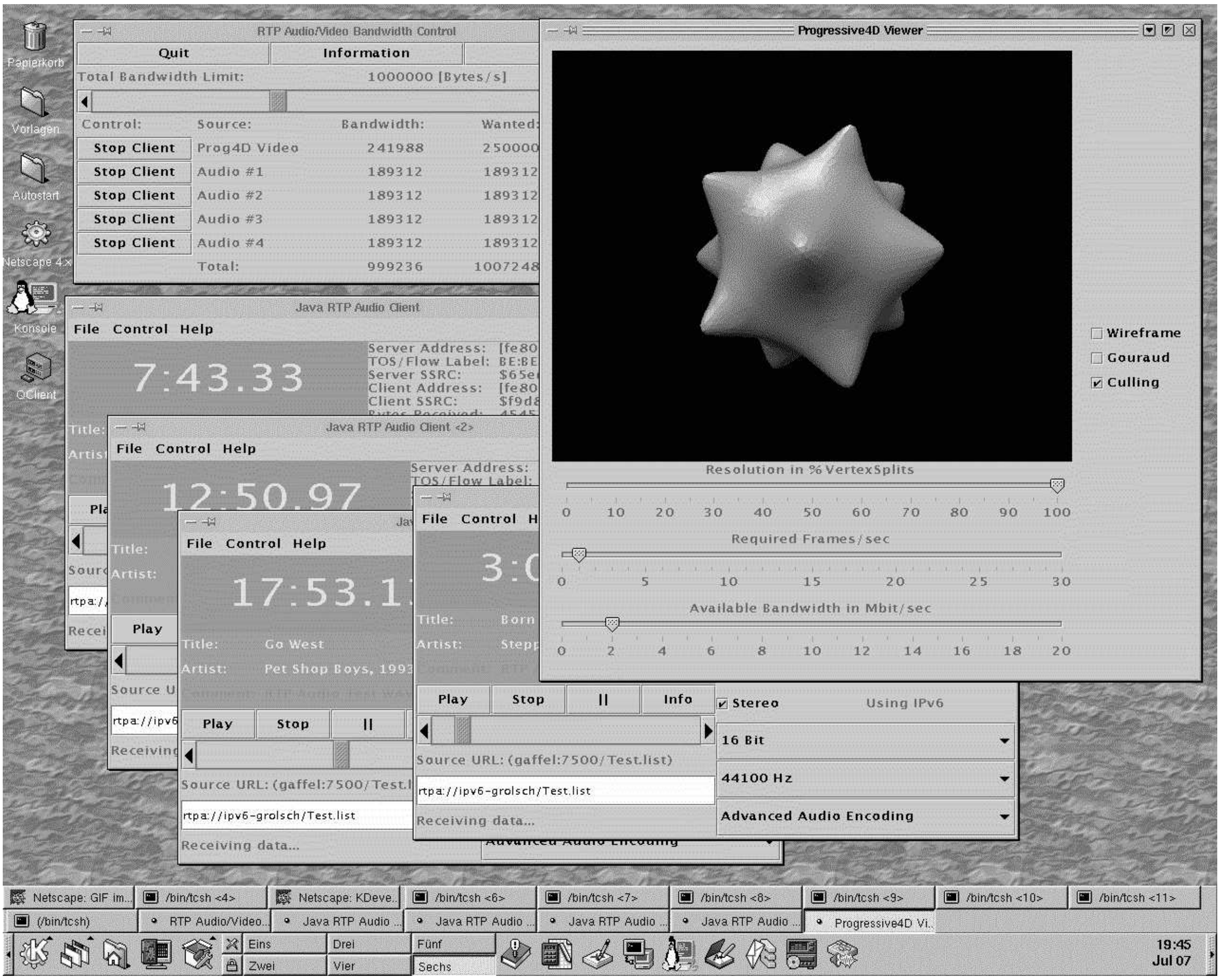


Abbildung A.20: MainControl, PROG4D Client und 4 RTP AUDIO Clients bei der Arbeit

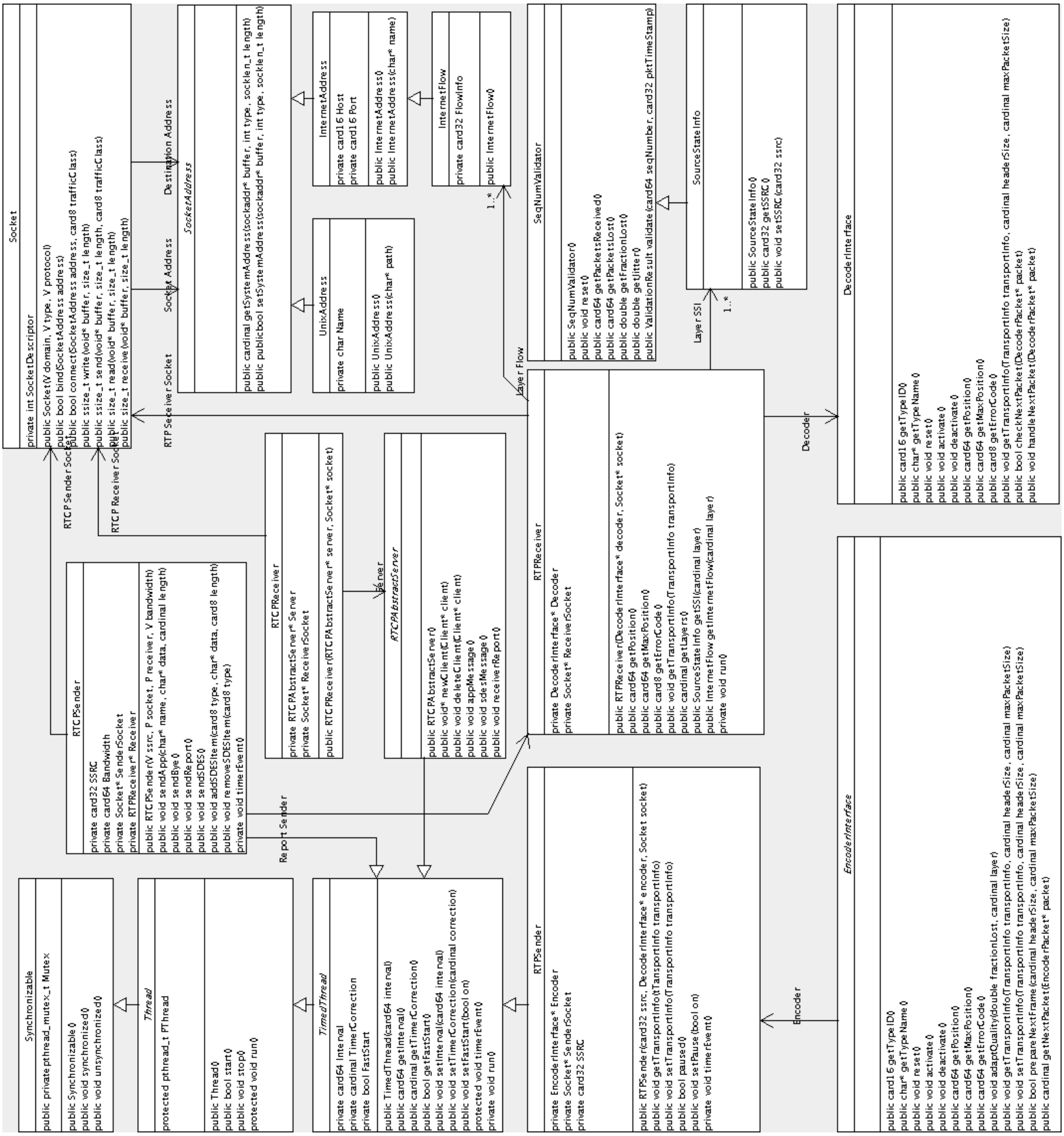
Anhang B

Die UML-Diagramme zu RTP Audio

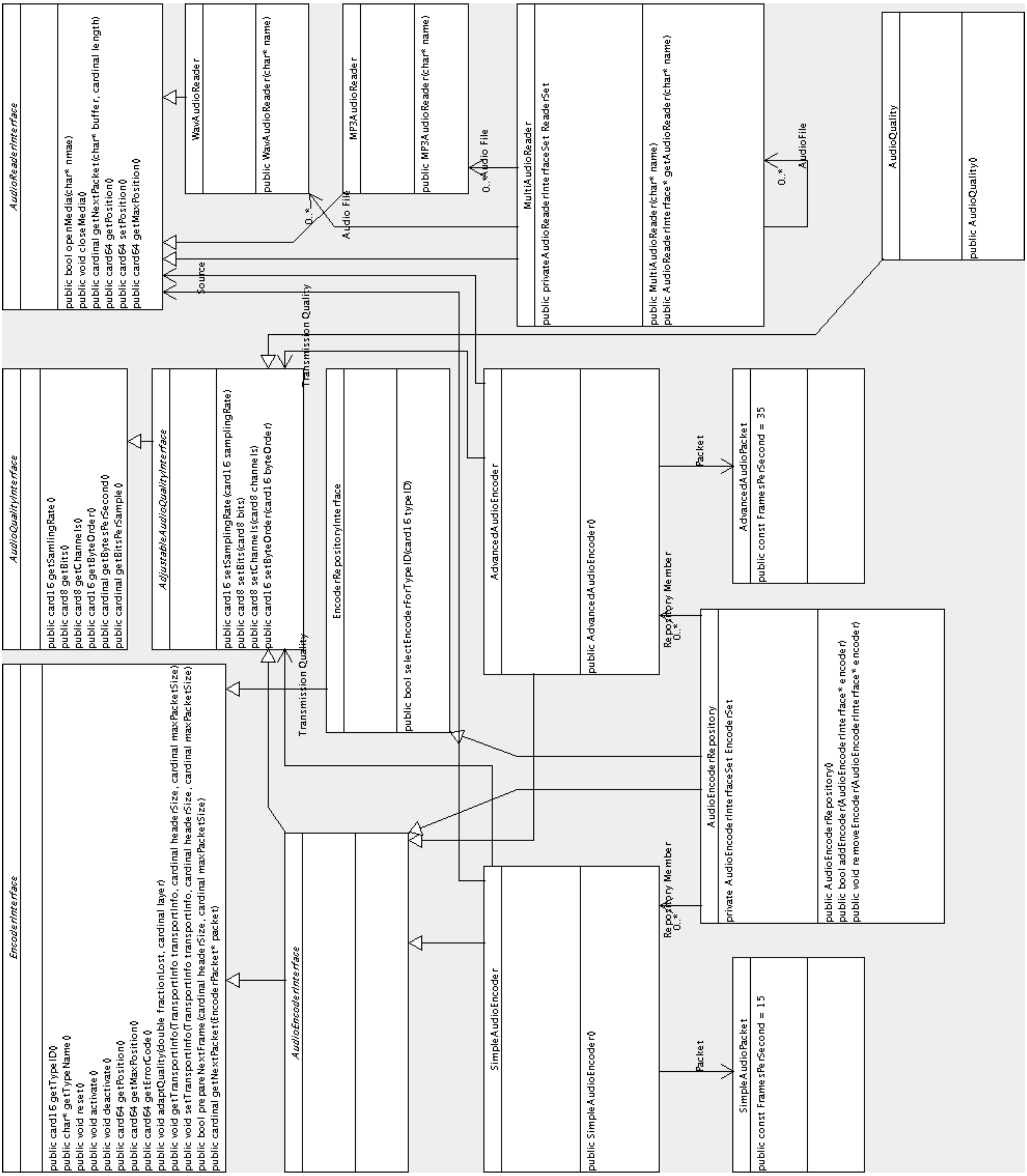
von Thomas Dreibholz

In diesem Anhangskapitel befinden sich UML-Klassendiagramme zu RTP Audio. Der Download der gesamten Implementation ist auf der [RTP Audio] Homepage möglich. Dort ist auch die Dokumentation der einzelnen Klassen zu finden.

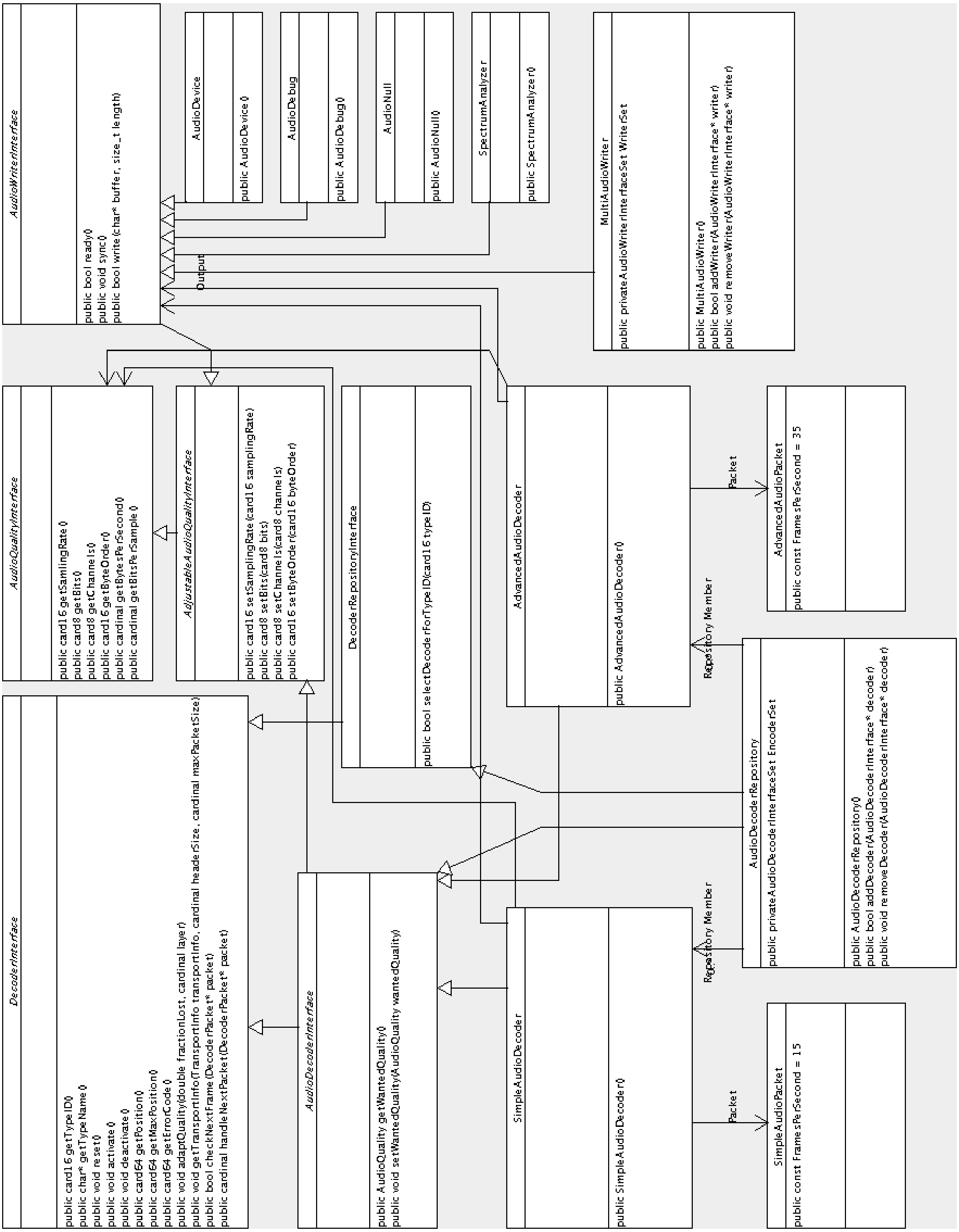
B.1 UML-Diagramm zum RTP Transport



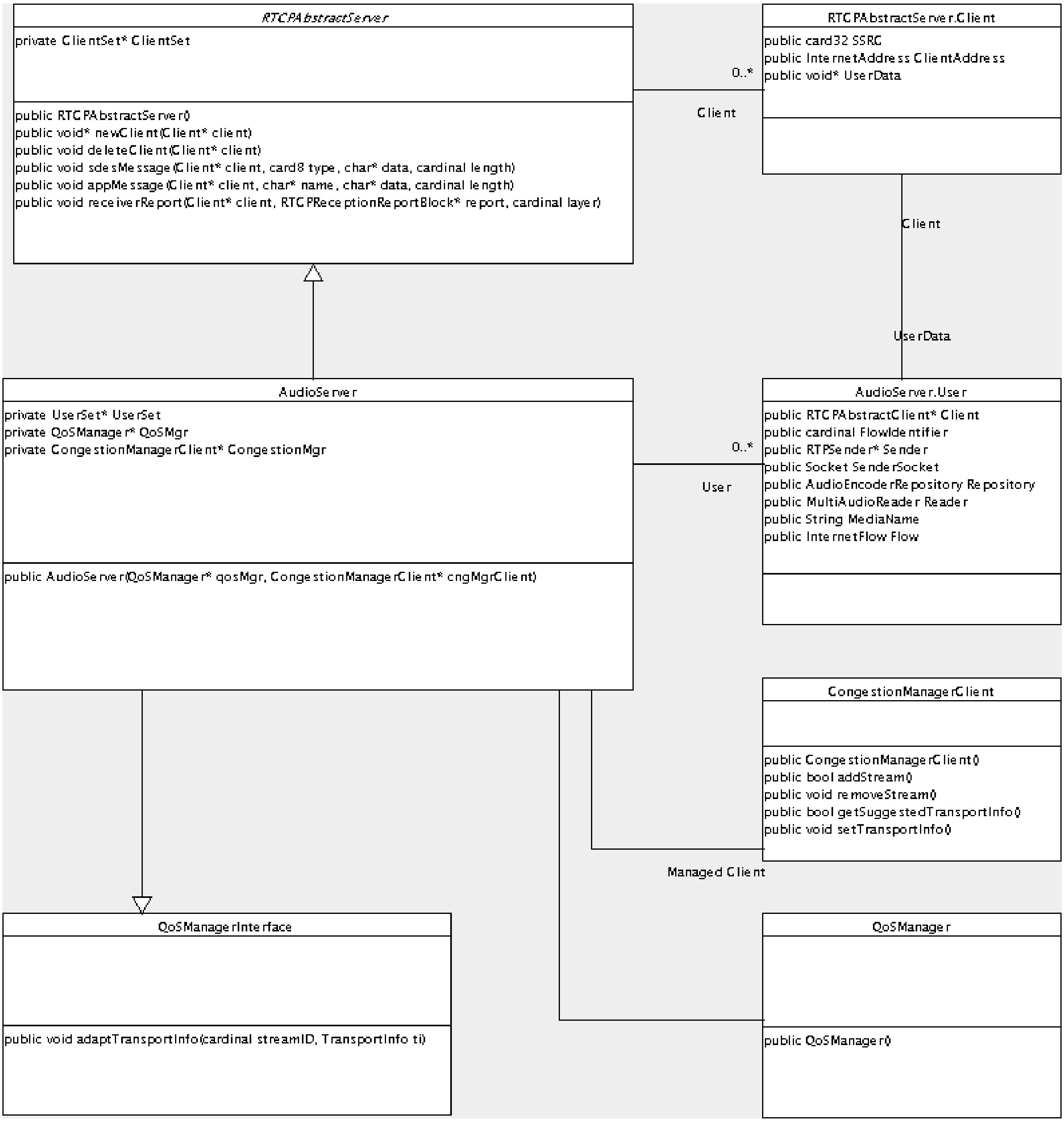
B.2 UML Diagramm zum Audio Encoder



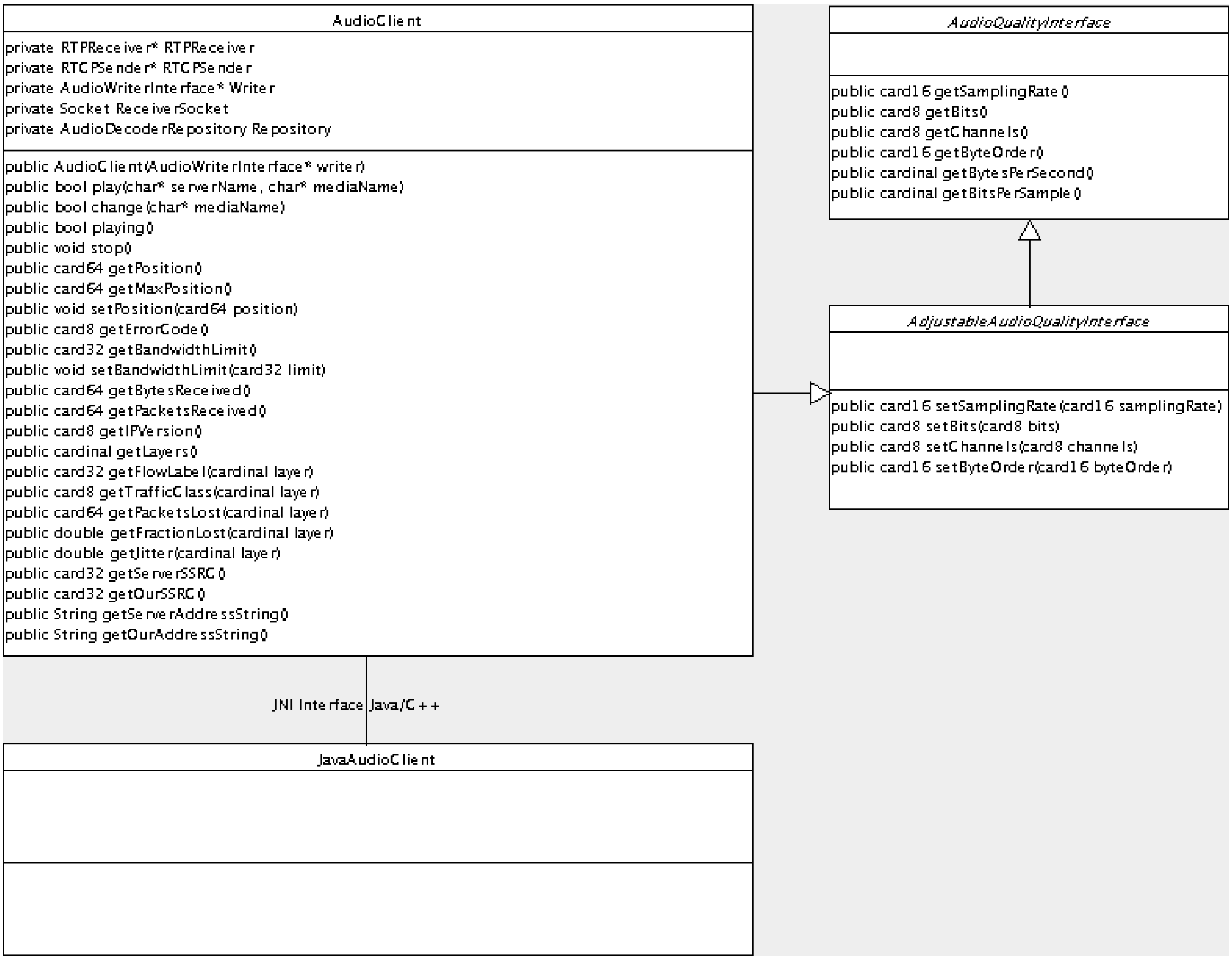
B.3 UML-Diagramm zum Audio Decoder



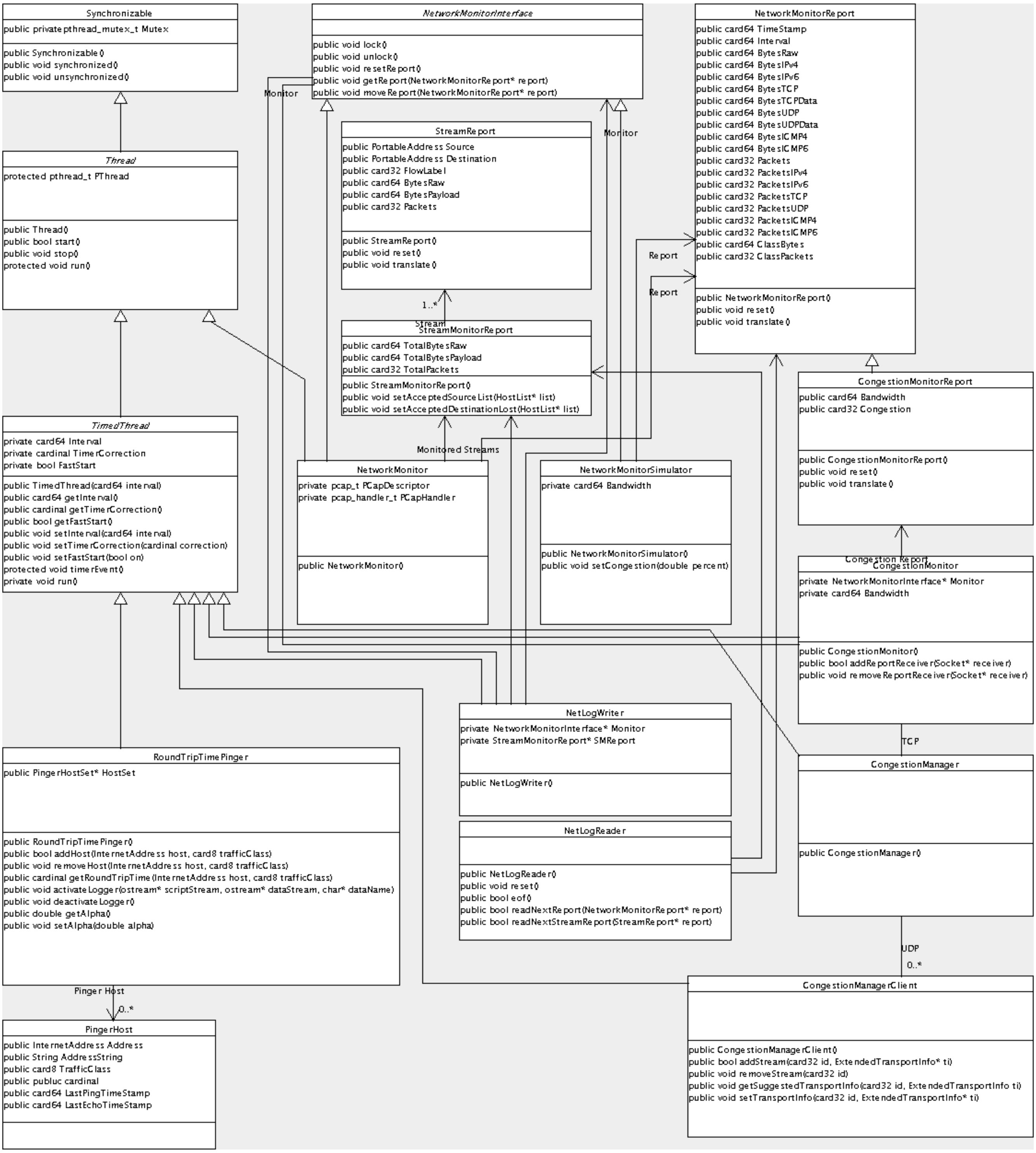
B.4 UML-Diagramm zum Audio Server



B.5 UML-Diagramm zum Audio Client



B.6 UML-Diagramm zum Network Monitor



Anhang C

Die QoS-Beschreibungen der Audiokodierungen

von Thomas Dreibholz

Eine Auflistung der kompletten QoS-Beschreibungen für die 23 Level von Simple Audio Encoding und Advanced Audio Encoding befindet sich in diesem Anhangskapitel. Alle Ausgaben wurden mit dem Programm EncoderInfo erzeugt (siehe Abschnitt A.1.2). Mit diesem Programm können bei Bedarf auch die QoS-Beschreibungen für sämtliche 152 Qualitätseinstellungen angezeigt werden.

C.1 Simple Audio Encoding

```
Wanted values:
Layer #0:
   WantedBytesPerSecond   =
      4117 <= 188872 <= 188872
   WantedPacketsPerSecond =
      17 <= 137 <= 137
   WantedMaxLossRate      = 0
WantedFramesPerSecond      =
      15 <= 15 <= 15
WantedMaxTransferDelay     = 1500 [ms]
Flags                      = $0000
StartFramesPerSecond       = 15
QualityLevels              = 23
QualityLayers              = 1
TotalBytesPerSecondLimit   =
   18446744073709551615
TotalPacketsPerSecondLimit = 4294967295
TotalFramesPerSecondLimit  = 4294967295

Level #0:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 3
   LevelUp               = 1
   LevelDown             = 255
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 4117
      PacketsPerSecond = 17
      FrameSize        = 275
      MaxLossRate      = 0
      Flags            = $0000

Level #1:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
```

```
      Quality              = 4
      LevelUp              = 2
      LevelDown            = 0
      QualityLayers        = 1
      Layer #0:
         BytesPerSecond   = 5227
         PacketsPerSecond = 17
         FrameSize        = 349
         MaxLossRate      = 0
         Flags            = $0000

   Level #2:
      BytesPerSecondScale   = 1
      PacketsPerSecondScale = 1
      FramesPerSecond       = 15
      FramesPerSecondScale  = 1
      MaxTransferDelay      = 1500 [ms]
      Quality               = 6
      LevelUp               = 3
      LevelDown             = 1
      QualityLayers         = 1
      Layer #0:
         BytesPerSecond   = 6322
         PacketsPerSecond = 17
         FrameSize        = 422
         MaxLossRate      = 0
         Flags            = $0000

   Level #3:
      BytesPerSecondScale   = 1
      PacketsPerSecondScale = 1
      FramesPerSecond       = 15
      FramesPerSecondScale  = 1
      MaxTransferDelay      = 1500 [ms]
      Quality               = 12
      LevelUp               = 4
      LevelDown             = 2
      QualityLayers         = 1
      Layer #0:
         BytesPerSecond   = 10732
         PacketsPerSecond = 17
         FrameSize        = 716
         MaxLossRate      = 0
         Flags            = $0000

   Level #4:
      BytesPerSecondScale   = 1
      PacketsPerSecondScale = 1
      FramesPerSecond       = 15
      FramesPerSecondScale  = 1
      MaxTransferDelay      = 1500 [ms]
      Quality               = 15
      LevelUp               = 5
      LevelDown             = 3
      QualityLayers         = 1
      Layer #0:
         BytesPerSecond   = 12937
         PacketsPerSecond = 17
         FrameSize        = 863
         MaxLossRate      = 0
         Flags            = $0000

   Level #5:
      BytesPerSecondScale   = 1
      PacketsPerSecondScale = 1
      FramesPerSecond       = 15
      FramesPerSecondScale  = 1
      MaxTransferDelay      = 1500 [ms]
      Quality               = 31
      LevelUp               = 6
      LevelDown             = 4
      QualityLayers         = 1
      Layer #0:
         BytesPerSecond   = 25282
         PacketsPerSecond = 32
         FrameSize        = 1686
         MaxLossRate      = 0
         Flags            = $0000

   Level #6:
      BytesPerSecondScale   = 1
      PacketsPerSecondScale = 1
      FramesPerSecond       = 15
      FramesPerSecondScale  = 1
      MaxTransferDelay      = 1500 [ms]
      Quality               = 38
      LevelUp               = 7
      LevelDown             = 5
      QualityLayers         = 1
      Layer #0:
         BytesPerSecond   = 29692
         PacketsPerSecond = 32
         FrameSize        = 1980
         MaxLossRate      = 0
         Flags            = $0000

   Level #7:
      BytesPerSecondScale   = 1
      PacketsPerSecondScale = 1
      FramesPerSecond       = 15
      FramesPerSecondScale  = 1
      MaxTransferDelay      = 1500 [ms]
      Quality               = 44
      LevelUp               = 8
      LevelDown             = 6
      QualityLayers         = 1
```

```
    Layer #0:
      BytesPerSecond   = 34102
      PacketsPerSecond = 32
      FrameSize        = 2274
      MaxLossRate      = 0
      Flags            = $0000

  Level #8:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 15
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 1500 [ms]
    Quality               = 51
    LevelUp               = 9
    LevelDown             = 7
    QualityLayers         = 1
    Layer #0:
      BytesPerSecond   = 38512
      PacketsPerSecond = 32
      FrameSize        = 2568
      MaxLossRate      = 0
      Flags            = $0000

  Level #9:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 15
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 1500 [ms]
    Quality               = 57
    LevelUp               = 10
    LevelDown             = 8
    QualityLayers         = 1
    Layer #0:
      BytesPerSecond   = 42922
      PacketsPerSecond = 32
      FrameSize        = 2862
      MaxLossRate      = 0
      Flags            = $0000

  Level #10:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 15
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 1500 [ms]
    Quality               = 63
    LevelUp               = 11
    LevelDown             = 9
    QualityLayers         = 1
    Layer #0:
      BytesPerSecond   = 48652
      PacketsPerSecond = 47
      FrameSize        = 3244
      MaxLossRate      = 0
      Flags            = $0000

  Level #11:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 15
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 1500 [ms]
    Quality               = 95
    LevelUp               = 12
    LevelDown             = 10
    QualityLayers         = 1
    Layer #0:
      BytesPerSecond   = 72022
      PacketsPerSecond = 62
      FrameSize        = 4802
      MaxLossRate      = 0
      Flags            = $0000

  Level #12:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 15
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 1500 [ms]
    Quality               = 105
    LevelUp               = 13
    LevelDown             = 11
    QualityLayers         = 1
    Layer #0:
      BytesPerSecond   = 78637
      PacketsPerSecond = 62
      FrameSize        = 5243
      MaxLossRate      = 0
      Flags            = $0000

  Level #13:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 15
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 1500 [ms]
    Quality               = 114
    LevelUp               = 14
    LevelDown             = 12
    QualityLayers         = 1
    Layer #0:
      BytesPerSecond   = 85252
      PacketsPerSecond = 62
      FrameSize        = 5684
```

```
      MaxLossRate      = 0
      Flags            = $0000

Level #14:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 124
   LevelUp               = 15
   LevelDown             = 13
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 93187
      PacketsPerSecond = 77
      FrameSize        = 6213
      MaxLossRate      = 0
      Flags            = $0000

Level #15:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 133
   LevelUp               = 16
   LevelDown             = 14
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 99802
      PacketsPerSecond = 77
      FrameSize        = 6654
      MaxLossRate      = 0
      Flags            = $0000

Level #16:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 143
   LevelUp               = 17
   LevelDown             = 15
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 106417
      PacketsPerSecond = 77
      FrameSize        = 7095
      MaxLossRate      = 0
      Flags            = $0000

Level #17:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 153
   LevelUp               = 18
   LevelDown             = 16
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 114352
      PacketsPerSecond = 92
      FrameSize        = 7624
      MaxLossRate      = 0
      Flags            = $0000

Level #18:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 204
   LevelUp               = 19
   LevelDown             = 17
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 150952
      PacketsPerSecond = 107
      FrameSize        = 10064
      MaxLossRate      = 0
      Flags            = $0000

Level #19:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 216
   LevelUp               = 20
   LevelDown             = 18
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 161092
      PacketsPerSecond = 122
      FrameSize        = 10740
      MaxLossRate      = 0
      Flags            = $0000
```

```
Level #20:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 229
   LevelUp               = 21
   LevelDown             = 19
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 169912
      PacketsPerSecond = 122
      FrameSize        = 11328
      MaxLossRate      = 0
      Flags            = $0000

Level #21:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 242
   LevelUp               = 22
   LevelDown             = 20
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 180052
      PacketsPerSecond = 137
      FrameSize        = 12004
      MaxLossRate      = 0
      Flags            = $0000

Level #22:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 255
   LevelUp               = 255
   LevelDown             = 21
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 188872
      PacketsPerSecond = 137
      FrameSize        = 12592
      MaxLossRate      = 0
      Flags            = $0000

Current Setting:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 15
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 1500 [ms]
   Quality               = 255
   LevelUp               = 22
   LevelDown             = 0
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 188872
      PacketsPerSecond = 137
      FrameSize        = 12592
      MaxLossRate      = 0
      Flags            = $0000
```

C.2 Advanced Audio Encoding

```
Wanted values:
Layer #0:
   WantedBytesPerSecond =
      5877 <= 47772 <= 47772
   WantedPacketsPerSecond =
      37 <= 37 <= 39
   WantedMaxLossRate      = 0
Layer #1:
   WantedBytesPerSecond   =
      0 <= 47180 <= 47180
   WantedPacketsPerSecond =
      0 <= 35 <= 37
   WantedMaxLossRate      = 15
Layer #2:
   WantedBytesPerSecond   =
      0 <= 94360 <= 94360
   WantedPacketsPerSecond =
      0 <= 70 <= 74
   WantedMaxLossRate      = 30
WantedFramesPerSecond      =
      35 <= 35 <= 35
WantedMaxTransferDelay     = 100 [ms]
Flags                      = $0000
StartFramesPerSecond       = 35
QualityLevels              = 23
QualityLayers              = 3
TotalBytesPerSecondLimit   =
   18446744073709551615
TotalPacketsPerSecondLimit = 4294967295
```

```
TotalFramesPerSecondLimit  = 4294967295

Level #0:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 3
   LevelUp               = 1
   LevelDown             = 255
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 5877
      PacketsPerSecond = 37
      FrameSize        = 168
      MaxLossRate      = 0
      Flags            = $0000

Level #1:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 4
   LevelUp               = 2
   LevelDown             = 0
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 6962
      PacketsPerSecond = 37
      FrameSize        = 199
      MaxLossRate      = 0
      Flags            = $0000

Level #2:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 6
   LevelUp               = 3
   LevelDown             = 1
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 8082
      PacketsPerSecond = 37
      FrameSize        = 231
      MaxLossRate      = 0
      Flags            = $0000

Level #3:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 12
   LevelUp               = 4
   LevelDown             = 2
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 12492
      PacketsPerSecond = 37
      FrameSize        = 357
      MaxLossRate      = 0
      Flags            = $0000

Level #4:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 15
   LevelUp               = 5
   LevelDown             = 3
   QualityLayers         = 1
   Layer #0:
      BytesPerSecond   = 14697
      PacketsPerSecond = 37
      FrameSize        = 420
      MaxLossRate      = 0
      Flags            = $0000

Level #5:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 31
   LevelUp               = 6
   LevelDown             = 4
   QualityLayers         = 2
   Layer #0:
      BytesPerSecond   = 14697
      PacketsPerSecond = 37
      FrameSize        = 420
      MaxLossRate      = 0
      Flags            = $0000
   Layer #1:
      BytesPerSecond   = 14105
      PacketsPerSecond = 35
```

```
      FrameSize        = 403
      MaxLossRate      = 15
      Flags            = $0000

Level #6:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 38
   LevelUp               = 7
   LevelDown             = 5
   QualityLayers         = 2
   Layer #0:
      BytesPerSecond   = 16902
      PacketsPerSecond = 37
      FrameSize        = 483
      MaxLossRate      = 0
      Flags            = $0000
   Layer #1:
      BytesPerSecond   = 16310
      PacketsPerSecond = 35
      FrameSize        = 466
      MaxLossRate      = 15
      Flags            = $0000

Level #7:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 44
   LevelUp               = 8
   LevelDown             = 6
   QualityLayers         = 2
   Layer #0:
      BytesPerSecond   = 19107
      PacketsPerSecond = 37
      FrameSize        = 546
      MaxLossRate      = 0
      Flags            = $0000
   Layer #1:
      BytesPerSecond   = 18515
      PacketsPerSecond = 35
      FrameSize        = 529
      MaxLossRate      = 15
      Flags            = $0000

Level #8:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 51
   LevelUp               = 9
   LevelDown             = 7
   QualityLayers         = 2
   Layer #0:
      BytesPerSecond   = 21312
      PacketsPerSecond = 37
      FrameSize        = 609
      MaxLossRate      = 0
      Flags            = $0000
   Layer #1:
      BytesPerSecond   = 20720
      PacketsPerSecond = 35
      FrameSize        = 592
      MaxLossRate      = 15
      Flags            = $0000

Level #9:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 57
   LevelUp               = 10
   LevelDown             = 8
   QualityLayers         = 2
   Layer #0:
      BytesPerSecond   = 23517
      PacketsPerSecond = 37
      FrameSize        = 672
      MaxLossRate      = 0
      Flags            = $0000
   Layer #1:
      BytesPerSecond   = 22925
      PacketsPerSecond = 35
      FrameSize        = 655
      MaxLossRate      = 15
      Flags            = $0000

Level #10:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 63
   LevelUp               = 11
   LevelDown             = 9
   QualityLayers         = 2
```

```
    Layer #0:
      BytesPerSecond   = 25722
      PacketsPerSecond = 37
      FrameSize        = 735
      MaxLossRate      = 0
      Flags            = $0000
    Layer #1:
      BytesPerSecond   = 25130
      PacketsPerSecond = 35
      FrameSize        = 718
      MaxLossRate      = 15
      Flags            = $0000

  Level #11:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 35
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 100 [ms]
    Quality               = 95
    LevelUp               = 12
    LevelDown             = 10
    QualityLayers         = 3
    Layer #0:
      BytesPerSecond   = 25722
      PacketsPerSecond = 37
      FrameSize        = 735
      MaxLossRate      = 0
      Flags            = $0000
    Layer #1:
      BytesPerSecond   = 25130
      PacketsPerSecond = 35
      FrameSize        = 718
      MaxLossRate      = 15
      Flags            = $0000
    Layer #2:
      BytesPerSecond   = 25130
      PacketsPerSecond = 35
      FrameSize        = 718
      MaxLossRate      = 30
      Flags            = $0000

  Level #12:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 35
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 100 [ms]
    Quality               = 105
    LevelUp               = 13
    LevelDown             = 11
    QualityLayers         = 3
    Layer #0:
      BytesPerSecond   = 27927
      PacketsPerSecond = 37
      FrameSize        = 798
      MaxLossRate      = 0
      Flags            = $0000
    Layer #1:
      BytesPerSecond   = 27335
      PacketsPerSecond = 35
      FrameSize        = 781
      MaxLossRate      = 15
      Flags            = $0000
    Layer #2:
      BytesPerSecond   = 27335
      PacketsPerSecond = 35
      FrameSize        = 781
      MaxLossRate      = 30
      Flags            = $0000

  Level #13:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 35
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 100 [ms]
    Quality               = 114
    LevelUp               = 14
    LevelDown             = 12
    QualityLayers         = 3
    Layer #0:
      BytesPerSecond   = 30132
      PacketsPerSecond = 37
      FrameSize        = 861
      MaxLossRate      = 0
      Flags            = $0000
    Layer #1:
      BytesPerSecond   = 29540
      PacketsPerSecond = 35
      FrameSize        = 844
      MaxLossRate      = 15
      Flags            = $0000
    Layer #2:
      BytesPerSecond   = 29540
      PacketsPerSecond = 35
      FrameSize        = 844
      MaxLossRate      = 30
      Flags            = $0000

  Level #14:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 35
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 100 [ms]
```

```
    Quality              = 124
    LevelUp              = 15
    LevelDown            = 13
    QualityLayers        = 3
    Layer #0:
       BytesPerSecond   = 32337
       PacketsPerSecond = 37
       FrameSize        = 924
       MaxLossRate      = 0
       Flags            = $0000
    Layer #1:
       BytesPerSecond   = 31745
       PacketsPerSecond = 35
       FrameSize        = 907
       MaxLossRate      = 15
       Flags            = $0000
    Layer #2:
       BytesPerSecond   = 31745
       PacketsPerSecond = 35
       FrameSize        = 907
       MaxLossRate      = 30
       Flags            = $0000

 Level #15:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 35
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 100 [ms]
    Quality               = 133
    LevelUp               = 16
    LevelDown             = 14
    QualityLayers         = 3
    Layer #0:
       BytesPerSecond   = 34542
       PacketsPerSecond = 37
       FrameSize        = 987
       MaxLossRate      = 0
       Flags            = $0000
    Layer #1:
       BytesPerSecond   = 33950
       PacketsPerSecond = 35
       FrameSize        = 970
       MaxLossRate      = 15
       Flags            = $0000
    Layer #2:
       BytesPerSecond   = 33950
       PacketsPerSecond = 35
       FrameSize        = 970
       MaxLossRate      = 30
       Flags            = $0000

 Level #16:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 35
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 100 [ms]
    Quality               = 143
    LevelUp               = 17
    LevelDown             = 15
    QualityLayers         = 3
    Layer #0:
       BytesPerSecond   = 36747
       PacketsPerSecond = 37
       FrameSize        = 1050
       MaxLossRate      = 0
       Flags            = $0000
    Layer #1:
       BytesPerSecond   = 36155
       PacketsPerSecond = 35
       FrameSize        = 1033
       MaxLossRate      = 15
       Flags            = $0000
    Layer #2:
       BytesPerSecond   = 36155
       PacketsPerSecond = 35
       FrameSize        = 1033
       MaxLossRate      = 30
       Flags            = $0000

 Level #17:
    BytesPerSecondScale   = 1
    PacketsPerSecondScale = 1
    FramesPerSecond       = 35
    FramesPerSecondScale  = 1
    MaxTransferDelay      = 100 [ms]
    Quality               = 153
    LevelUp               = 18
    LevelDown             = 16
    QualityLayers         = 3
    Layer #0:
       BytesPerSecond   = 38952
       PacketsPerSecond = 37
       FrameSize        = 1113
       MaxLossRate      = 0
       Flags            = $0000
    Layer #1:
       BytesPerSecond   = 38360
       PacketsPerSecond = 35
       FrameSize        = 1096
       MaxLossRate      = 15
       Flags            = $0000
    Layer #2:
       BytesPerSecond   = 38360
       PacketsPerSecond = 35
```

```
      FrameSize        = 1096
      MaxLossRate      = 30
      Flags            = $0000

Level #18:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 204
   LevelUp               = 19
   LevelDown             = 17
   QualityLayers         = 3
   Layer #0:
      BytesPerSecond   = 38952
      PacketsPerSecond = 37
      FrameSize        = 1113
      MaxLossRate      = 0
      Flags            = $0000
   Layer #1:
      BytesPerSecond   = 38360
      PacketsPerSecond = 35
      FrameSize        = 1096
      MaxLossRate      = 15
      Flags            = $0000
   Layer #2:
      BytesPerSecond   = 76720
      PacketsPerSecond = 70
      FrameSize        = 2192
      MaxLossRate      = 30
      Flags            = $0000

Level #19:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 216
   LevelUp               = 20
   LevelDown             = 18
   QualityLayers         = 3
   Layer #0:
      BytesPerSecond   = 41157
      PacketsPerSecond = 37
      FrameSize        = 1176
      MaxLossRate      = 0
      Flags            = $0000
   Layer #1:
      BytesPerSecond   = 40565
      PacketsPerSecond = 35
      FrameSize        = 1159
      MaxLossRate      = 15
      Flags            = $0000
   Layer #2:
      BytesPerSecond   = 81130
      PacketsPerSecond = 70
      FrameSize        = 2318
      MaxLossRate      = 30
      Flags            = $0000

Level #20:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 229
   LevelUp               = 21
   LevelDown             = 19
   QualityLayers         = 3
   Layer #0:
      BytesPerSecond   = 43362
      PacketsPerSecond = 37
      FrameSize        = 1239
      MaxLossRate      = 0
      Flags            = $0000
   Layer #1:
      BytesPerSecond   = 42770
      PacketsPerSecond = 35
      FrameSize        = 1222
      MaxLossRate      = 15
      Flags            = $0000
   Layer #2:
      BytesPerSecond   = 85540
      PacketsPerSecond = 70
      FrameSize        = 2444
      MaxLossRate      = 30
      Flags            = $0000

Level #21:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 242
   LevelUp               = 22
   LevelDown             = 20
   QualityLayers         = 3
   Layer #0:
      BytesPerSecond   = 45567
      PacketsPerSecond = 37
      FrameSize        = 1302
      MaxLossRate      = 0
```

```
      Flags             = $0000
   Layer #1:
      BytesPerSecond    = 44975
      PacketsPerSecond = 35
      FrameSize         = 1285
      MaxLossRate       = 15
      Flags             = $0000
   Layer #2:
      BytesPerSecond    = 89950
      PacketsPerSecond = 70
      FrameSize         = 2570
      MaxLossRate       = 30
      Flags             = $0000

Level #22:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 255
   LevelUp               = 255
   LevelDown             = 21
   QualityLayers         = 3
   Layer #0:
      BytesPerSecond    = 47772
      PacketsPerSecond = 37
      FrameSize         = 1365
      MaxLossRate       = 0
      Flags             = $0000
   Layer #1:
      BytesPerSecond    = 47180
      PacketsPerSecond = 35
      FrameSize         = 1348
      MaxLossRate       = 15
      Flags             = $0000
   Layer #2:
      BytesPerSecond    = 94360
      PacketsPerSecond = 70
      FrameSize         = 2696
      MaxLossRate       = 30
      Flags             = $0000

Current Setting:
   BytesPerSecondScale   = 1
   PacketsPerSecondScale = 1
   FramesPerSecond       = 35
   FramesPerSecondScale  = 1
   MaxTransferDelay      = 100 [ms]
   Quality               = 255
   LevelUp               = 22
   LevelDown             = 0
   QualityLayers         = 3
   Layer #0:
      BytesPerSecond    = 47772
      PacketsPerSecond = 37
      FrameSize         = 1365
      MaxLossRate       = 0
      Flags             = $0000
   Layer #1:
      BytesPerSecond    = 47180
      PacketsPerSecond = 35
      FrameSize         = 1348
      MaxLossRate       = 15
      Flags             = $0000
   Layer #2:
      BytesPerSecond    = 94360
      PacketsPerSecond = 70
      FrameSize         = 2696
      MaxLossRate       = 30
      Flags             = $0000
```

Anhang D

Die Skripte der DiffServ-Router

von Jan Selzer

In diesem Anhangskapitel befinden sich je ein DiffServ-Skript für den Core-Router (*holsten*) und den Border-Router (*grolsch*). Diese Skripte sind mit dem Programmpaket *iproute2* (siehe [IPRoute]) zu verwenden und konfigurieren die DiffServ-Klassen EF, AF11, AF21 und BE mit einem SLA. Für die einzelnen Messungen wurden diese Skripte entsprechend den Bandbreiteangaben angepaßt.

D.1 Core Router

```
#!  /bin/bash

TC=/home/IV/Labor/Stud/selzer/iproute2/tc/tc
# hier ist die Ethernet-Karte einzugeben, auf der das Skript installiert
# werden soll
dev="dev eth2"

# um das Szenario zu löschen, ersetze in den nächsten
# 3 Zeilen "add" auf "del".
# First a DSMARK qdisc is introduced in order to retrieve TOS from IP header
$TC qdisc add $dev handle 1:0 root dsmark indices 64 set_tc_index
$TC filter add $dev parent 1:0 protocol ip prio 1 tcindex mask 0xffff \
shift 0 pass_on

# Second a CBQ qdisc is used in order to support EF, AF and BE classes
# um die gemeinsame Bandbreite zu ändern,
# ersetze die Zahl nach dem "bandwidth".
# Eingabe ist in "Mbit" oder "KBit" oder "Mbps"(Megabyte per second) gültig
$TC qdisc add $dev parent 1:0 handle 2:0 cbq bandwidth 100Mbit \
cell 8 avpkt 1200 mpu 70
$TC filter add $dev parent 2:0 protocol ip prio 1 tcindex mask \
0xf0 shift 4 pass_on
```

```
##### EF class specific setup
## um die Banbreite für EF-Klasse zu ändern, ersetze die Zahl nach dem
## Wort "rate" in der nächsten Zeile.
$TC class add $dev parent 2:0 classid 2:5 cbq bandwidth 100Mbit rate \
3000Kbit avpkt 1200 prio 1 bounded isolated allot 1514 weight 300Kbit \
maxburst 10 #defmap 0
$TC qdisc add $dev parent 2:5 pfifo limit 5
$TC filter add $dev parent 1:0 protocol ip prio 1 handle 0x2e tcindex \
classid 1:151
$TC filter add $dev parent 2:0 protocol ip prio 1 handle 5 \
tcindex classid 2:5

##### BE class specific setup
## um die Banbreite für BE-Klasse zu ändern, ersetze die Zahl nach dem
## Wort "rate" in der nächsten Zeile.
$TC class add $dev parent 2:0 classid 2:6 cbq bandwidth 100Mbit rate \
5000Kbit avpkt 1200 prio 7 bounded isolated allot 1514 weight 500Kbit \
maxburst 40
#borrow
$TC qdisc add $dev parent 2:6 red limit 60KB min 15KB max 45KB burst 20 \
avpkt 1200 bandwidth 100Mbit probability 0.4
$TC filter add $dev parent 1:0 protocol ip prio 1 handle 0x0 tcindex \
classid 1:161
$TC filter add $dev parent 2:0 protocol ip prio 1 handle \
6 tcindex classid 2:6

##### AF Class 1 specific setup
## um die Banbreite für AF1-Klasse zu ändern, ersetze die Zahl nach dem
## Wort "rate" in der nächsten Zeile
$TC class add $dev parent 2:0 classid 2:2 cbq bandwidth 100Mbit \
rate 4000Kbit avpkt 1200 prio 4 bounded isolated allot 1514 weight  \
400Kbit maxburst 10 #defmap 0 #borrow
$TC filter add $dev parent 2:0 protocol ip prio 1 handle 1 tcindex \
classid 2:2 #police rate 25Kbit burst 10 drop
$TC qdisc add $dev parent 2:2 gred setup DPs 1 default 1 grio
 # --- AF Class 2 DP 1---
   $TC filter add $dev parent 1:0 protocol ip prio 1 handle 18 tcindex \
classid 1:121
   $TC qdisc add $dev parent 2:2 gred limit 60KB min 15KB \
max 45KB burst 20 avpkt 1200 bandwidth 100Mbit DP 1 probability 0.02 prio 2

##### AF Class 2 specific setup
## um die Banbreite für AF2-Klasse zu ändern, ersetze die Zahl nach dem
## Wort "rate" in der nächsten Zeile
$TC class add $dev parent 2:0 classid 2:1 cbq bandwidth 100Mbit \
rate 3000Kbit avpkt 1200 prio 5 bounded isolated allot 1514 weight \
300Kbit maxburst 20 #defmap 0 #borrow
```

```
$TC filter add $dev parent 2:0 protocol ip prio 1 handle 2 tcindex \
classid 2:1 #police rate 25Kbit burst 10 drop
$TC qdisc add $dev parent 2:1 gred setup DPs 1 default 1 grio
# --- AF Class 1 DP 1---
$TC filter add $dev parent 1:0 protocol ip prio 1 handle 10 tcindex \
classid 1:111
$TC qdisc add $dev parent 2:1 gred limit 60KB min 15KB \
max 45KB burst 20 avpkt 1200 bandwidth 100Mbit DP 1 \
probability 0.02 prio 2
```

D.2 Border Router

```
#! /bin/sh

TC=/home/IV/Labor/Stud/selzer/iproute2/tc/tc
# hier sind die Ethernet-Karten einzugeben, auf denen
# das Skript installiert werden soll.
# INDEV ist Device für eingehenden Strom.
# EGDEV ist Device für ausgehenden Strom.
EGDEV=eth1
INDEV=eth0

# um das Szenario zu löschen, ersetze in den nächsten
# 2 Zeilen "add" auf "del".
$TC qdisc add dev $INDEV handle ffff: ingress
$TC filter add dev $INDEV parent ffff: protocol ip prio 4 \
handle 1: u32 divisor 1

$TC filter add dev $INDEV parent ffff: protocol ip prio 4 u32 \
match ip tos 0x2e 0xff \
police index 1 rate 2Mbit burst 100K \
drop flowid :1

$TC filter add dev $INDEV parent ffff: protocol ip prio 4 u32 \
match ip tos 0x0a 0xff \
police index 2 rate 2Mbit burst 100K \
drop flowid :2

$TC filter add dev $INDEV parent ffff: protocol ip prio 4 u32 \
match ip tos 0x12 0xff \
police index 3 rate 2Mbit burst 100K \
drop flowid :3

$TC filter add dev $INDEV parent ffff: protocol ip prio 4 u32 \
match ip tos 0x0 0xff \
police index 4 rate 4Mbit burst 100K \
```

```
drop flowid :4

####################### Egress side #######################
# um das Szenario zu löschen, ersetze in der nächsten
# Zeile "add" auf "del".
$TC qdisc add dev $EGDEV handle 1:0 root dsmark indices 64
$TC class change dev $EGDEV classid 1:1 dsmark mask 0x0 \
       value 0x2e
$TC class change dev $EGDEV classid 1:2 dsmark mask 0x0 \
       value 0x0a
$TC class change dev $EGDEV classid 1:3 dsmark mask 0x0 \
       value 0x12
$TC class change dev $EGDEV classid 1:4 dsmark mask 0x0 \
       value 0x0



$TC filter add dev $EGDEV parent 1:0 protocol ip prio 1 \
          handle 1 tcindex classid 1:1
$TC filter add dev $EGDEV parent 1:0 protocol ip prio 1 \
          handle 2 tcindex  classid 1:2
$TC filter add dev $EGDEV parent 1:0 protocol ip prio 1 \
          handle 3 tcindex classid 1:3
$TC filter add dev $EGDEV parent 1:0 protocol ip prio 1 \
          handle 4 tcindex  classid 1:4
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[RTP Audio] Homepage von RTP AUDIO
http://www-student.cs.uni-bonn.de/~dreibhol/rn
http://www.bigfoot.com/~dreibholz/rn (Redirector 1)
http://dreibholz.da.ru/rn (Redirector 2)

[Qt] Homepage der Qt-Bibliothek
http://www.troll.no

[Blackdown] Blackdown Java-Linux Porting Project
http://www.blackdown.org

[ThreadFAQ] LinuxThreads Frequently Asked Questions
http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html

[IETF] Internet Engineering Task Force (IETF)
http://www.ietf.org

[RFC Editor] RFC Editor
http://www.rfc-editor.org

[RFC 791] Defense Advanced Research Projects Agency (DARPA), September 1981
RFC 791: Internet Protocol, Version 4 (IPv4)
ftp://ftp.isi.edu/in-notes/rfc791.txt

[RFC 2460] Deering/Hinden, Dezember 1998
RFC 2460: Internet Protocol, Version 6 (IPv6)
ftp://ftp.isi.edu/in-notes/rfc2460.txt

[RFC 2675] Borman/Deering/Hinden, August 1999
RFC 2675: IPv6 Jumbograms
ftp://ftp.isi.edu/in-notes/rfc2675.txt

[RFC 2732] Hinden/Carpenter/Masinter, Dezember 1999
RFC 2732: Format for Literal IPv6 Addresses in URLs
ftp://ftp.isi.edu/in-notes/rfc2732.txt

[RFC 792] Postel, September 1981
RFC 792: Internet Control Message Protocol (ICMPv4)
ftp://ftp.isi.edu/in-notes/rfc792.txt

[RFC 2464] Conta/Deering, Dezember 1998
RFC 2463: Internet Control Message Protocol (ICMPv6)
ftp://ftp.isi.edu/in-notes/rfc2664.txt

[RFC 2462] Thomson/Narten, Dezember 1998
RFC 2462: IPv6 Stateless Address Autoconfiguration
ftp://ftp.isi.edu/in-notes/rfc2462.txt

[RFC 768] Postel, August 1980
RFC 768: User Datagram Protocol (UDP)
ftp://ftp.isi.edu/in-notes/rfc768.txt

[RFC 761] Defense Advanced Research Projects Agency (DARPA), Januar 1980
RFC 761: Transmission Control Protocol (TCP)
ftp://ftp.isi.edu/in-notes/rfc761.txt

[RFC 1122] Braden, Oktober 1989
RFC 1122: Requirements for Internet Hosts – Communication Layers
ftp://ftp.isi.edu/in-notes/rfc1122.txt

[RFC 1323] Jacobson/Braden/Borman, Mai 1992
RFC 1323: TCP Extensions for High Performance
ftp://ftp.isi.edu/in-notes/rfc1323.txt

[RFC 1071] Braden/Borman/Partridge, September 1988
RFC 1071: Computing the Internet Checksum
ftp://ftp.isi.edu/in-notes/rfc1071.txt

[RFC 1349] Almquist, Juli 1992
RFC 1349: Type of Service in the Internet Protocol Suite
ftp://ftp.isi.edu/in-notes/rfc1349.txt

[RFC 2474] Nichols/Blake/Baker/Black, Dezember 1998
RFC 2474: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers
ftp://ftp.isi.edu/in-notes/rfc2474.txt

[RFC 1149] Waitzman, April 1990
RFC 1149: A Standard for the Transmission of IP Datagrams on Avian Carriers
ftp://ftp.isi.edu/in-notes/rfc1149.txt

[RFC 2549] Waitzman, April 1999
RFC 2549: IP over Avian Carriers with Quality of Service
ftp://ftp.isi.edu/in-notes/rfc2549.txt

[RFC 1305] Mills, März 1992
RFC 1305: Network Time Protocol
ftp://ftp.isi.edu/in-notes/rfc1305.txt

[RFC 2208] Mankin/Baker/Braden/Bradner/O'Dell/Romanow/Weinrib/Zhang, September 1997
RFC 2208: Resource ReSerVation Protocol (RSVP) - Some Guidelines on Deployment
`ftp://ftp.isi.edu/in-notes/rfc2208.txt`

[RFC 1889] Schulzrinne/Casner/Frederick/Jacobson, Januar 1996
RFC 1889: A Transport Protocol for Real-Time Applications (RTP)
`ftp://ftp.isi.edu/in-notes/rfc1889.txt`

[RFC 1890] Schulzrinne, Januar 1996
RFC 1890: RTP Profile for Audio and Video Conferences with Minimal Control
`ftp://ftp.isi.edu/in-notes/rfc1890.txt`

[3GPP] Third Generation Partnership Project (3GPP),
`http://www.3gpp.org`

[EtherTap] EtherTap: Linux Kernel-Sources, Documentation/networking/ethertap.txt

[InternetQoS] Xipeng Xiao/Lionel M. Ni, Michigan State University
Internet QoS: A Big Picture
`http://www.cse.msu.edu/~xiaoxipe/papers/inet.qos.bigpicture.pdf`

[AdvNet] Saravanan Radhakrishnan, 1999
Linux - Advanced Networking Overview
`http://icawww1.epfl.ch/linux-diffserv`

[DSFolien] Albrecht/Koester, 1999
`file:/home/IV/Labor/labor/v3d2/doc/diffserv-folien13121999.pdf`

[LinuxDS] Werner Almesberger, April 1999
Differentiated Services on Linux
`ftp://lrcftp.epfl.ch/pub/linux/diffserv/misc/dsid-01.ps.gz`
`http://icawww1.epfl.ch/linux-diffserv`

[LinuxTC] Werner Almesberger, Juni 1999
Linux Network Traffic Control
`ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz`
`http://icawww1.epfl.ch/linux-diffserv`

[IPRoute] IPRoute 2
`ftp://lrcftp.epfl.ch/pub/linux/diffserv/mirror/iproute2-2.2.4-now-ss991023.tar.gz`

Index