IP-Telefonie über Differentiated Services

Diplomarbeit
der Philosophisch-naturwissenschaftlichen
Fakultät der Universität Bern

vorgelegt von

Matthias Scheidegger

2001

Leiter der Arbeit:

Professor Dr. Torsten Braun

Zusammenfassung

Diese Arbeit erörtert den Einsatz von Differentiated Services für die IP-Telefonie. Zu diesem Zweck gibt sie erst einen Überblick über die grundlegenden Begriffe und Konzepte, unter anderem Signalisierungsprotokolle, Audiokodierung und RTP, um dann die verschiedenen Probleme beim Einsatz der IP-Telefonie und übliche Lösungsansätze näher zu beleuchten. Der Hauptteil beginnt damit, dass einige Vorschläge für Einsatz- und Optimierungsmöglichkeiten von Differentiated Services in der IP-Telefonie vorgestellt werden. Es folgt Beschrieb und Design der im Rahmen der Arbeit geschriebenen Software und eine Reihe von Tests, die das Verhalten der Software und die Anwendbarkeit der vorgeschlagenen Ansätze überprüft.

Inhaltsverzeichnis

Kapitel 1

Grundlagen zu VoIP

Lange bevor Informationstechnologie und das Internet die Art, wie Menschen kommunizieren, revolutionierte, tat dies bereits die Telefonie. Heute, am Übergang zum 21. Jahrhundert, ist sie ein fester Bestandteil der Kultur der Industrienationen geworden und hat durch das Aufkommen von Mobiltelefonen noch einmal massiv an Bedeutung hinzugewonnen.
Bedingt durch die schnelle Entwicklung digitaler Technologien ist die Menge möglicher Kommunikationsformen aber heute bedeutend grösser als sie es noch vor wenigen Jahren war. Als Beispiele seien Email, das World Wide Web und Multimedia-Kommunikation genannt. Es liegt nun nahe, die Telefonie, die inzwischen auch zum grössten Teil auf Digitaltechnik beruht, mit diesen neuen Kommunikationsformen zu vereinen, einerseits um die Neuentwicklungen ebenso allgemein zugänglich zu machen, wie dies Telefone heute sind; andererseits können durch die Vereinheitlichung der Kommunikationstechnologie Kosten gespart werden, vor allem im Hinblick darauf, dass die Telefonie in Zukunft voraussichtlich nur einen geringen Teil des gesamten Datenvolumens ausmachen wird.

1.1 Public Switched Telephone Network

Das heutige Telefonnetzwerk ist das Public Switched Telephone Network (PSTN), ein digitales, auf Zeitmultiplex (Time Division Multiplex TDM) basiertes Netz, das nur auf der so genannten “letzten Meile” zum Teil noch mit analogen Signalen arbeitet. Im Begriff des PSTN sind drahtlose Netze wie das GSM-Netz nicht enthalten, sondern nur die Zugangspunkte zu diesen. Allerdings wird für die Übertragung zwischen den Funkstationen solcher Netze meist das PSTN verwendet, weshalb auch sie nicht unabhängig betrachtet werden können.

Nebst der Grundfunktion der Sprachübertragung bietet es auch die Möglichkeit, Datenverkehr darüber abzuwickeln; mit herkömmlichen Analogmodems,

Faxgeräten oder direkt über ISDN. Diese Möglichkeit wird aber, mit Ausnahme von Fax, meist nur dazu verwendet, ein Computersystem oder -netz mit dem Internet zu verbinden.

1.2 Der Übergang zu VoIP

1.2.1 Gründe für den Übergang

Das PSTN bietet qualitativ hochstehende Sprachübertragung mit grosser Zuverlässigkeit. Warum also ein Wechsel?
Jede Verbindung auf dem PSTN benötigt eine fixe Bandbreite, was sich vor allem in dicht besiedelten Gegenden stark aufaddiert und die Sicherstellung der benötigten Kapazität zu einer teuren Angelegenheit macht. Durch die stetig steigende Nachfrage und die sich ändernden Bedürfnisse der Konsumenten verschärft sich dieses Problem noch. Die Telefonanbieter haben versucht, mit ISDN, dem Integrated Services Digital Network, auf die veränderten Bedürfnisse ihrer Kunden einzugehen. Weil aber die Kosten für die Erweiterung und Anpassung der Infrastruktur beträchtlich und daher die Preise für den Endkunden entsprechend hoch waren, hat sich dieses System nie wirklich durchgesetzt.

Es ist eine weit verbreitete Ansicht, dass das Problem einerseits bei der fehlenden Integration von Stimm- und Datendiensten und andererseits bei der tiefen Bandbreite der analogen Leitungen der “Letzten Meile” zu suchen ist.
Diese Integration zu erreichen war unter anderem das Ziel von ATM (Asynchronous Transfer Mode), einem auf optischen Leitungen basierenden, paket-orientierten Netz. Trotz seiner komplexen Architektur hat sich dieses System im Backbone-Bereich vorübergehend etabliert, beim Endbenutzer aber konnte es wegen des explosiven Wachstums des Internet nie Fuss fassen. Das Internet Protocol (IP) und der dazugehörige Protokollstack wurden dadurch zum dominanten Standard.

1.2.2 Grundlegende Unterschiede zu IP

Zwischen IP-Netzen und den geswitchten PSTN-Netzen bestehen einige grundlegende Unterschiede. Der wichtigste ist die Datagramm-Orientierung von IP: Pakete werden mit einer Zieladresse versehen und zum nächsten Router geschickt. Dieser bestimmt mittels einer Routing-Tabelle einen nachfolgenden Router, dem er das Paket schicken kann. Dies geht solange weiter, bis ein Router erreicht wurde, der direkt mit dem Ziel verbunden ist und diesem das Paket übermitteln kann. Die Entscheidung, über welchen Pfad ein Paket von einem Router weitergesendet wird, muss dabei nicht notwendigerweise immer derselbe sein, was bedeutet, dass

Pakete nicht immer in derselben Reihenfolge ankommen, in der sie gesendet wurden. Unter anderem können Auslastung, Übertragungskosten und Übertragungsgeschwindigkeit einer Verbindung bei der Entscheidung berücksichtigt werden. Im Falle eines Übertragungsfehlers oder bei Überlastung eines Routers können IP-Pakete auch verloren gehen, ohne dass der Sender oder der Empfänger explizit benachrichtigt werden. Im Gegensatz dazu wird auf dem PSTN die Route der Pakete im voraus bestimmt und auf allen dazwischen liegenden Stationen registriert, bevor überhaupt Daten gesendet werden. Nach dieser Reservierung werden die Daten dann reihenfolgetreu und verlustfrei übertragen. Nach Beendigung der Übertragung müssen diese Reservationen auch wieder rückgängig gemacht werden.
Ein anderer gewichtiger Unterschied ist, dass die Grösse von IP-Paketen innerhalb bestimmter Grenzen variieren kann. Sie können sogar von einem Router in mehrere Pakete aufgeteilt werden, die im Endsystem wieder zusammengesetzt werden müssen. Die Grösse der Übertragungseinheiten auf dem PSTN ist im Gegensatz dazu fest vorgegeben. Die verwendeten TDM[1]-Rahmen beispielsweise sind 125 ms, ATM-Zellen immer 53 Bytes lang.

Diese Eigenheiten von IP haben mehrere Implikationen auf die Router und Endsysteme. Da ein Router keine Kenntnis über den Datenfluss zu haben braucht, dem die einzelnen Pakete angehören, kann die Entscheidungsstruktur sehr einfach gehalten werden. Diese Einfachheit bleibt auch bei einer erhöhten Anzahl von Datenströmen erhalten. Hingegen kann nie genau vorausgesagt werden, wie viele Pakete zu einem bestimmten Zeitpunkt von einem Router bewältigt werden müssen, weshalb dieser überschüssige Pakete mitunter auch zwischenlagert. Dies geschieht in so genannten Queues (Warteschlangen). Wenn nun die Überlast zu gross wird und die Queues sich füllen, müssen einzelne Pakete fallen gelassen (gedroppt) werden.
In den Endsystemen führen diese Vereinfachungen zu erhöhter Komplexität. Sender und Empfänger von Daten müssen eventuelle Paketverluste oder Veränderungen in der Reihenfolge erkennen und korrigieren. Aufgeteilte Pakete müssen nach ihrer Ankunft wieder zusammengefügt werden. Dies übernimmt typischerweise das Transportprotokoll TCP. Im Falle von VoIP kommt es allerdings nicht zum Zuge, weil es sich nicht zur Übertragung von Echtzeitdaten eignet.

Ein Vorteil von IP wurde auch schon in ATM realisiert, nämlich die Asynchronität der Datenübertragung. Im klassischen PSTN werden in konstanten Zeitabständen Daten geschickt, was für Telefongespräche durchaus angebracht sein mag, aber viel Bandbreite verschwendet, wenn nicht konstant Daten gesendet

[1]Time Division Multiplex

werden müssen, was vor allem für Computeranwendungen typisch ist. Bei IP und ATM werden nur Pakete gesendet, wenn überhaupt Daten zu übermitteln sind, wodurch viel Bandbreite eingespart werden kann.

1.2.3 Begriffe

Einige grundlegende Begriffe werden im Folgenden erläutert (nach [DousB], p.3):

Endpunkt Ein- und Austrittspunkt von Medienflüssen innerhalb eines MGs. Es wird unterschieden zwischen physikalischen und virtuellen Endpunkten.

Verbindung Assoziation zwischen verschiedenen Endpunkten zum Zweck der Datenübertragung zwischen ihnen. Diese können Teil eines einzigen MGs oder mehrerer, über ein Netz verbundenener MGs sein.

Media Gateway (MG) Gerät, an das man Telefone, Daten-Anschlüsse und Anderes anschliessen kann. Er ist charakterisiert durch eine Menge von Endpunkten und Verbindungen.

Media Gateway Controller (MGC) Verwaltet die Benutzerinformationen und kommuniziert per Signalisierungsprotokoll mit MGs und Endpunkten. Dies ist der zentrale Teil eines VoIP-Netzes, der in der Zentrale des Telefonanbieters steht.

Anruf (Call) Logische Assoziation von Verbindungen zwischen zwei oder mehr Endpunkten. Ein Punkt-zu-Punkt-Anruf ist beispielsweise eine einzige Verbindung zwischen zwei Endpunkten. Verbindungen innerhalb eines Anrufs können aktiv oder inaktiv sein. Letzteres bedeutet, dass die Verbindung zwar besteht, aber nichts auf dem Endpunkt ausgegeben wird.

Signalisierungsprotokoll Protokoll zur Erstellung und Verwaltung von Verbindungen und Anrufen, das zwischen den Komponenten eines VoIP-Systems abläuft.

1.2.4 Überblick über die Signalling-Ansätze

Unter den Signalisierungsprotokollen für IP-Netze gibt es drei, die das Potential haben, zum Standard zu avancieren: Das von einer IETF-Gruppe entworfene SIP, das den ITU-T-Gremien entstammende H.323-Protokoll und Megaco, das Resultat der gemeinsamen Bemühungen dieser Institutionen. H.323 hat sich durch seine frühe Verbreitung in der Internet-Telefonie einen Vorsprung verschafft, der aber durch einige nicht zu unterschätzende Vorteile der anderen Protokolle, vor allem

von SIP, aufgeholt werden könnte. Der Grund dafür ist vor allem in den unterschiedlichen Designansätzen zu suchen. H.323 und SIP werden in den nächsten beiden Abschnitten vorgestellt.

1.2.4.1 H.323

Die Empfehlung H.323[2] von der ITU war der Vorreiter unter den Vorschlägen für IP-basierte Multimediakommunikation und wurde bereits früh in Produkten eingesetzt. Vor allem durch seine Verwendung in Internet-Telefonen konnte viel Erfahrung gesammelt und umgesetzt werden. In Firmennetzen ist seine Verbreitung beträchtlich.
H.323 verweist auf zwei andere ITU-Vorschläge, H.255.0[3] für die Anrufsignalisierung zwischen den Endpunkten oder zwischen Endpunkt und Gatekeeper, H.245[4] zum Aushandeln von Übertragungsparametern und Aufbau logischer Kanäle. Im Gegensatz zu den anderen besprochenen Signalisierungsprotokollen werden die Nachrichten in H.323 nicht mit ASCII-Zeichen kodiert, stattdessen wird ASN.1 für die Notation der Daten verwendet.
Die Elemente von H.323 sind das Terminal, der Gatekeeper (GK), die Multicast-Einheit (multicast unit, MCU) und der Gateway (GW). Es müssen jedoch nicht alle diese Elemente auf verschiedenen Maschinen implementiert werden; zum Beispiel kann eine MCU durchaus in einem Gatekeeper integriert sein.
Die logische Einheit solcher, möglicherweise geographisch entfernter Terminals, MCUs, Gateways und einem Gatekeeper nennt man eine H.323-*Zone*. Sie bildet die Management-Einheit von H.323. Der “intelligente” Teil einer Zone ist der *Gatekeeper* (es gibt nur einen GK pro Zone).
Ein *Terminal* ist ein Gerät, das einen Signalisierungs-Endpunkt enthält, der es einem oder mehreren Benutzern erlaubt, mit einer oder mehreren Parteien in Echtzeit zu kommunizieren [5]. Auf der Signalisierungs-Ebene entspricht es also ungefähr dem allgemeinen Konzept eines Endpunktes.

Ablauf der Signalisierung

Es gibt drei Möglichkeiten, wie die Signalisierung zwischen zwei Endpunkten abgewickelt werden kann. Die erste und einfachste ist die direkte Kommunikation zwischen den beiden. Alternativ dazu kann die Signalisierung über den Gatekeeper geleitet werden, was bei Verbindungen über einen Gateway sinnvoll ist. Als

[2] [H.323]
[3] [H.255.0]
[4] [H.245]
[5] [DousB], p.37

dritte Möglichkeit kann nur die erste Verbindung mit H.225.0 über den GK geleitet werden, der Austausch der Terminal-Fähigkeiten mit H.245 aber direkt geschehen. Der Einfachheit halber wird im Folgenden nur die direkte Variante mit bereits beim GK registrierten Terminals betrachtet.
Der Aufbau einer Verbindung mit H.323 besteht aus 5 Phasen:

Phase A: erster Verbindungsaufbau und Zulassungsprüfung.

Phase B: Austausch der Terminal-Fähigkeiten.

Phase C: Aufbau des Medienkanals zwischen den Endpunkten.

Phase D: optionales Aushandeln zusätzlicher Parameter mit dem GK.

Phase E: Beendigung.

In **Phase A** (siehe Abb. 1.1) öffnet der Anrufer eine TCP-Verbindung zum gegenüberliegenden Terminal und sendet, sobald diese steht, darüber eine H.225.0-SETUP-Nachricht, die u.a. besagt, ob eine Punkt-zu-Punkt- oder eine Mehrpunkt-Verbindung angestrebt wird, welche Transportadresse der Anrufer benutzt und ob eine FastStart-Prozedur(s.u.) erwünscht ist. Der angerufene Endpunkt handelt darauf mit dem ihm zugeordneten GK aus, ob die Verbindung akzeptiert werden darf, welches Signalisierungsmodell (direkt oder per GK) benutzt werden soll und welche Bandbreite zur Verfügung steht. Dazu wird das RAS-Protokoll (Reservation, Authentication and Status) verwendet, das Teil von H.225.0 ist. Falls dies erfolgreich verläuft, sendet der Angerufene eine ALERTING-Nachricht an den Anrufer zurück. Dies entspricht dem Klingelton von herkömmlichen Telefonen. Wenn der Anruf schliesslich akzeptiert wird, folgt eine CONNECT-Nachricht mit der für Phase B vorgesehenen TCP-Adresse, womit diese Phase abgeschlossen ist.

Phase B beginnt damit, dass der anrufende Endpunkt eine TCP-Verbindung zum vorher angegebenen TCP-Port öffnet, um darüber die folgende H.245-Kommunikation abzuwickeln. Daraufhin erhält er eine Beschreibung der Fähigkeiten des angerufenen Endpunktes und sendet wiederum eine Beschreibung der eigenen Fähigkeiten. Der Endpunkt mit den höheren Fähigkeiten übernimmt die Rolle des *Masters* (und damit die Aufgabe, eventuelle Konflikte zu beheben), der andere wird der *Slave*. Nachdem dies bestimmt ist, senden beide eine ACK-Nachricht, womit die Verbindungsparameter ausgehandelt und Phase B beendet ist.

Nachdem nun die Fähigkeiten beider Parteien bekannt sind, können in **Phase C** die Medienkanäle geöffnet werden (siehe Abb. 1.2. In H.323 sind diese immer *unidirektional*, weshalb für eine Konversation immer zwei Kanäle geöffnet werden müssen. Über den vorher geöffneten H.245 Kanal senden beide Seiten eine

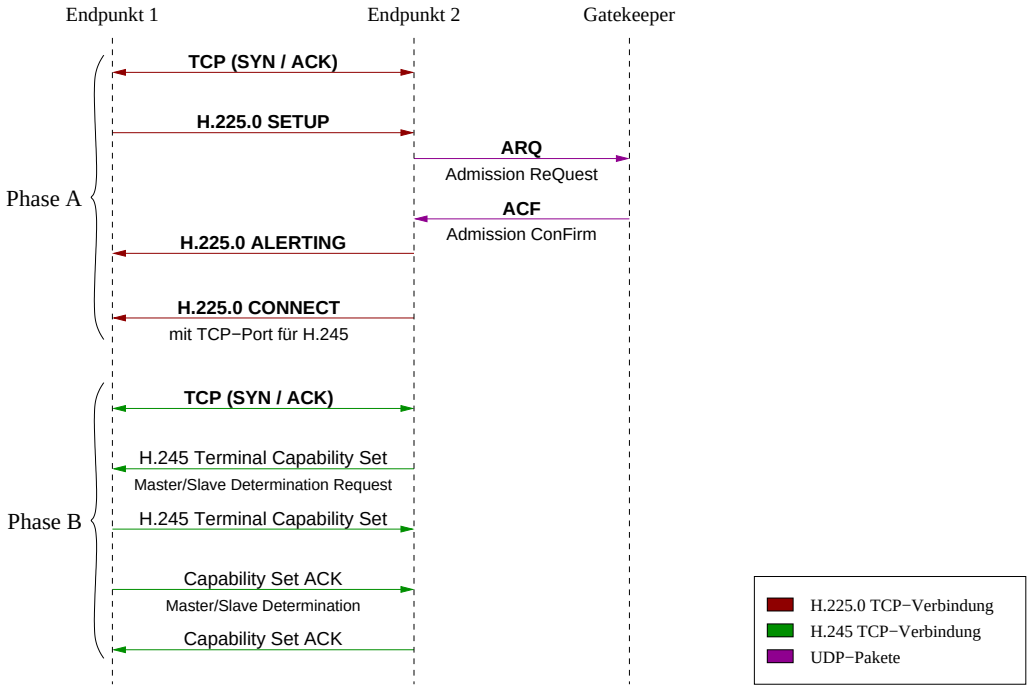


Abbildung 1.1: H.323-Aufbau, Phasen A und B

OpenLogicalChannel-Nachricht, die genaue Informationen über den zu öffnenden Kanal, wie beispielsweise Audiokodierung oder UDP-Port, enthält. Wenn für eine Partei die früher vom Gatekeeper zugewiesene Bandbreite nicht ausreicht, um den geforderten Kanal zu öffnen, wird in der optionalen **Phase D** weitere Bandbreite angefordert. Dazu wird wieder das RAS-Protokoll verwendet. In einem solchen Fall muss die betroffene Partei den Kanal mit einer CloseLogicalChannel-Nachricht schliessen und wieder öffnen. Durch senden von ACK-Nachrichten auf den jeweils gegenüberliegenden UDP-Port wird die Verbindung bestätigt und die Aufbauphase ist beendet.

Phase E, die Beendigung des Anrufs, ist bedeutend einfacher. Wenn eine Partei die Verbindung abbrechen will, schickt sie eine CloseLogicalChannel- gefolgt von einer EndSessionCommand-Nachricht, worauf die Gegenseite ihrerseits mit einer CloseLogicalChannel-Nachricht reagiert. Diese Nachrichten werden mit ACKs bestätigt, womit die H.245-Verbindung beendet ist. Schlussendlich sendet die abbrechende Partei noch eine ReleaseComplete-Nachricht über die H.225.0-Verbindung; damit ist der Anruf beendet.

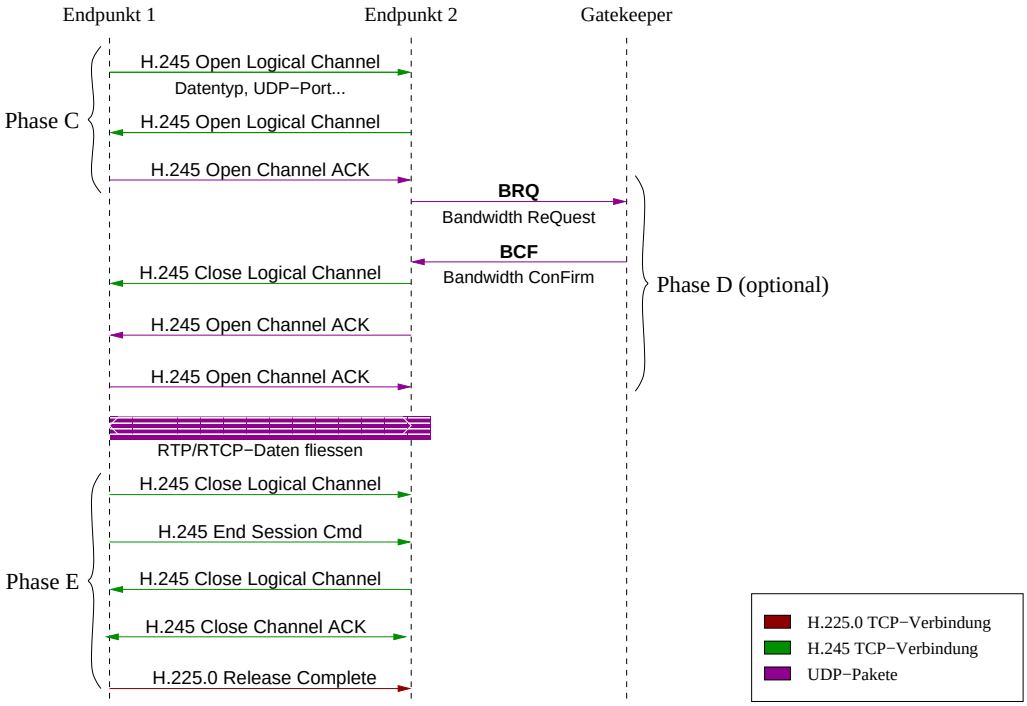


Abbildung 1.2: H.323-Aufbau, Phasen C,D und E

Konferenzschaltungen

Nach dem Punkt-zu-Punkt-Anruf ist die nächsthöhere Stufe eine Konferenzschaltung. Hierfür wird offensichtlich eine Multicast-Unit benötigt, die der Einfachheit halber im Gatekeeper implementiert sei. Weiter geschieht die Signalisierung hier nicht direkt, sondern über den GK (routed signalling).

Der anfängliche Verbindungsaufbau gestaltet sich analog zum oben beschriebenen Ablauf, mit einigen Unterschieden: Anstatt einer direkten Verbindung zwischen den Endpunkten werden zwei geöffnet, nämlich zwischen Anrufer und MCU und zwischen MCU und angerufenem Endpunkt. Ausserdem beinhalten die gesendeten SETUP-Nachrichten zusätzlich eine *ConferenceID* und die Aufforderung zum Erstellen einer Konferenzschaltung (*Create*). Nach dem öffnen der logischen Kanäle fliessen diese über die MCU, wo sie anfangs einfach weitergeleitet werden. Später, wenn weitere Teilnehmer hinzugestossen sind, wird sie die verschiedenen Ströme vor dem Weiterleiten mischen. Der angerufene Endpunkt nimmt im weiteren Verlauf eine wichtige Rolle ein, deswegen sei er mit E bezeichnet.
Nachdem die Verbindung auf diese Art geöffnet wurde, kann eine weitere Partei dazustossen. Dazu sendet sie eine SETUP-Nachricht an E, die dieselbe Con-

ferenceID wie oben, jedoch zusammen mit einer *Join*-Anfrage, enthält. E teilt dem Anrufer mittels einer FACILITY-Nachricht mit, welcher GK die Verbindung kontrolliert und daher kontaktiert werden muss, worauf der Anrufer die Verbindung schliesst und einen neuen Versuch über den angegebenen GK startet. Danach verläuft die Prozedur wie zu Beginn: Die Nachrichten des Anrufers werden an E weitergeleitet und eine Verbindung über die MCU geöffnet. Diese mischt jetzt die Signale der einzelnen Teilnehmer, bevor sie diese weiterleitet.

Eine noch komplexere Situation wäre den Einsatz eines Gateways gegeben, der die H.323-Zone mit dem PSTN oder einem anderen IP-basierten Signalisierungsprotokoll verbindet. Je nachdem, welches System “auf der anderen Seite” des Gateways liegt, ergeben sich hier eine ganze Reihe neuer Probleme. Darauf soll aber hier nicht im Detail eingegangen werden.

Erweiterungen

Wie aus obigen Prozeduren ersichtlich ist, gestaltet sich ein Verbindungsaufbau mit H.323 ausserordentlich komplex. Um diese Komplexität etwas zu verringern, wurde eine beschleunigte Methode namens **FastStart** eingeführt. Diese funktioniert so, dass in der SETUP-Nachricht eine Reihe von vordefinierten logischen Kanälen angegeben wird, aus welchen einer ausgewählt werden kann. Falls der Angerufene die FastStart-Prozedur unterstützt, sendet er in einer der Antwortnachrichten eine eigene Auswahl zurück. Nach Abschluss der H225.0-Phase wählen nun beide Seiten eine der Möglichkeiten und können sofort beginnen, auf diesem logischen Kanal zu senden, womit die (umständliche) H.245-Phase entfällt.
Obwohl der Gatekeeper die von H.323-Terminals beanspruchte Bandbreite unter Kontrolle hat, werden sehr häufig die Daten anderer Anwendungen über dasselbe Netzwerk übertragen. Um trotzdem die Quality of Service (QoS) aufrecht erhalten zu können, bietet H.323 Unterstützung für RSVP (siehe Seite 45). Wenn sich ein Terminal beim Gatekeeper registriert, kann es durch das *transportQoS*-Feld seine RSVP-Fähigkeit signalisieren. Da Reservationen in RSVP aber nur von den betroffenen Endpunkten getätigt werden können, resultiert daraus eine grössere Komplexität in den Endpunkten und mangelnde Kontrolle des Gatekeepers über die QoS.
H.323 hat verschiedene andere Fähigkeiten, wie die Umleitung von Anrufen oder spezielle Nachrichten, um die Verbindung zu den beteiligten Geräten zu prüfen. Diese haben hier aber nur geringe Relevanz, weshalb sie nicht weiter beschrieben werden.

Vor- und Nachteile

H.323 ist ein recht fortgeschrittenes Protokoll, in dessen Design durch seine grosse Verbreitung viele Erfahrungswerte eingeflossen sind. Gerade diese Verbreitung ist sein grösster Vorteil. Allerdings ist es auch das mit Abstand komplexeste der hier besprochenen Protokolle; die Verwandtschaft mit den Signalisierungsprotokollen der alten Telefonnetze ist klar erkennbar. Wie diese ist es aber sehr robust, wenn auch nur zwischen Produkten desselben Herstellers. Die Komplexität des Protokolls führt nämlich nicht nur zu einer Leistungsverminderung, sondern auch zu vielen Inkompatibilitäten zwischen den verschiedenen Implementierungen.
Wenn direkte Signalisierung verwendet wird, skaliert das Protokoll relativ gut. Es entsteht jedoch ein Flaschenhals, sobald routed signalling eingesetzt wird, was für Konferenzgespräche und dergleichen meist unumgänglich ist. Auch RSVP weist, wie weiter unten erläutert, Skalierungsprobleme auf.
Aus der Sicht des Netzwerkadministrators kann die verwendete ASN.1-Notation, die von "blossem Auge" nicht entzifferbar ist, als Nachteil erscheinen, da für eine Analyse des Netzwerkverkehrs immer spezielle Software vonnöten ist. Dadurch, und durch die Komplexität des Protokolls, dürfte der Einsatz von H.323 also einen nicht zu unterschätzenden Administrationsaufwand mit sich bringen.

1.2.4.2 SIP

Das Session Initiation Protocol ist ein weiteres Signalisierungsprotokoll zum Erstellen und Kontrollieren von Beenden von Sitzungen mit einem oder mehreren Teilnehmern ([RFC 2543]). Es stammt aus derselben Designschule wie HTTP, was sich durch eine klare, textbasierte Struktur ausdrückt. Das darunter liegende Protokoll kann UDP, aber auch TCP sein; In den meisten Fällen wird aber UDP wegen seiner Geschwindigkeitsvorteile zum Einsatz kommen. Die Fähigkeiten von SIP entsprechen denen der anderen besprochenen Protokolle. Zusätzlich bietet es Unterstützung von Teilnehmer-Mobilität (intelligente Netzwerk-Dienste), lässt aber bei Konferenzen mehr offen als beispielsweise H.323.

Auch SIP definiert eine Reihe von Begriffen, von denen im Folgenden die wichtigsten vorgestellt werden: Ein *Anruf* (Call) besteht aus allen Teilnehmern einer Konferenz, die von derselben Quelle eingeladen wurden (womit eine Konferenz aus mehreren Anrufen bestehen kann) und wird durch eine Anruf-ID identifiziert. Begonnen wird ein Anruf mit einer *Einladung* (Invitation). Die einladende Partei wird *Initiator* genannt.
Man unterscheidet zwei Arten von Antworten auf SIP-Anfragen: *provisorische* und *endgültige*. Provisorische Antworten werden dazu benutzt, einem Client Fortschritte beim Bearbeiten seiner Anfrage anzuzeigen, bevor eine endgültige Ant-

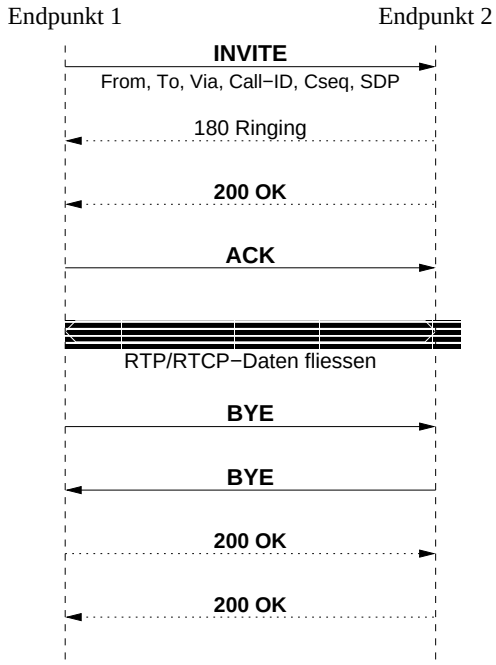


Abbildung 1.3: Einfacher Verbindungsablauf mit SIP

wort die Transaktion abschliesst. Antworten mit einem Code der Form 1xx sind provisorisch, alle anderen sind endgültig.
Das Konzept des *Proxy*-Servers ist analog zu dem aus HTTP bekannten, wird aber ergänzt durch den *Umleitungs*-Server (Redirect Server), der dem Client mögliche andere Adressen angibt, an die er seine Anfrage richten kann. Damit kann beispielsweise ein Anrufer an den aktuellen Anschluss eines Mobilteilnehmers verwiesen werden.
An den Endpunkten einer Verbindung gibt es zwei Arten von Applikationen: *User Agent Clients* (UAC), die SIP-Anfragen senden, und *User Agent Server* (UAS), die diese empfangen, wenn nötig beim Benutzer nachfragen und dann die Anfrage akzeptieren, ablehnen oder umleiten. Applikationen, die beide Rollen übernehmen können, nennt man schlicht *User Agents* (UA).

Signalisierungs-Ablauf

Anhand einiger Beispiele soll nun der Ablauf eines SIP-Anrufs skizziert werden, angefangen mit einem einfachen Punkt-zu-Punkt-Gespräch (vgl. Abb. 1.3).

Der Initiator beginnt den Anruf mit dem Senden einer INVITE-Anfrage an Endpunkt 2. In der Anfrage enthalten sind, nebst Informationen über die Anruf-

Strecke (From, To, Via), eine Anruf-ID, eine Sequenznummer und ein SDP-Block, mit dem eine Liste von akzeptierten Medienformaten mitsammt der dazugehörigen Transportadressen enthalten ist (SDP steht für “Session Description Protocol”, definiert in [RFC 2327]). Die Gegenseite antwortet darauf mit “180 Ringing” und, wenn die Verbindung angenommen wird, mit “200 OK”. Zusätzlich kann ein SDP-Block mit Gegenvorschlägen übertragen werden. Die Aufbauphase wird durch eine ACK-Anfrage vom Initiator abgeschlossen, in der er per SDP das definitive Medienformat angeben kann. Fehlt diese Angabe, gilt das mit der INVITE-Anfrage übertragene Format. Eine Antwort von der angerufenen Partei ist nicht mehr nötig. Wenn der Initiator den Anruf schon während der Aufbauphase wieder abbrechen will, sendet er eine BYE-Anfrage.
Um einen laufenden Anruf abzubrechen, kann eine der Parteien eine BYE-Anfrage senden. Nachdem diese bestätigt ist, gilt der Anruf als abgeschlossen.

Wenn ein Umleitungs-Server in das Modell einbezogen wird, ändert sich nur der Anfang des Ablaufs (siehe Abb. 1.4). Der Initiator schickt eine INVITE-Nachricht an den Ort, an dem er den gewünschten Gesprächspartner vermutet. Im Beispiel ist dieser jedoch permanent umgezogen, weshalb der Server mit “301 Moved” antwortet und den neuen Aufenthaltsort übermittelt. Die Verbindung wird darauf mit einem ACK beendet und neuer Versuch mit den erhaltenen Daten gestartet. Der Aufbau ist ab hier identisch zum oben beschriebenen, mit dem Unterschied, dass die Sequenznummer Cseq nicht zurückgesetzt wird.

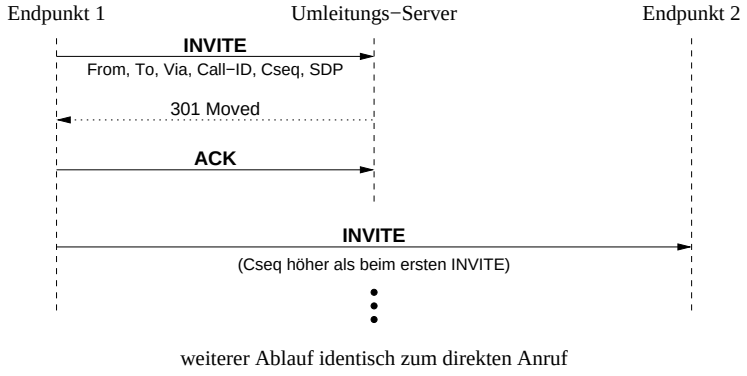


Abbildung 1.4: SIP-Anruf mit Umleitung

Konferenzen

Im Gegensatz zu H.323 verbinden sich die Clients bei Konferenzen in SIP typischerweise nicht mit der Multicast-Unit, sondern werden von ihr (oder einem

anderen User Agent) eingeladen. In der Sitzungsbeschreibung wird in diesem Fall die Multicast-Adresse der Konferenz, oder eine Liste von Teilnehmer-Adressen, mitgeschickt. Nach dem üblichen Verbindungsaufbau sendet der neue Teilnehmer seine Daten im ersten Fall an die Multicast-Adresse, im zweiten an jeden einzelnen Teilnehmer.
Alternativ dazu kann eine Konferenz auch dadurch begonnen werden, dass sich jeder durch Kontaktaufnahme mit der MCU “selber einlädt”, analog zu H.323. Die Details sind hierfür aber nicht so genau geregelt, wie das in H.323 der Fall ist.

Vor- und Nachteile

Der grösste Vorteil von SIP ist sein einfaches und klares Design, das trotz seiner Einfachheit die Anzahl der möglichen Anwendungen nicht einschränkt und leicht erweiterbar ist. Die Zahl der Erweiterungen ist denn auch jetzt schon beträchtlich, was glücklicherweise nicht zu Lasten der Stabilität des Protokolls zu gehen scheint. Vor allem bei Szenarien, die über die Komplexität einer firmeninternen Lösung hinausgehen, wird sich die gute Skalierbarkeit als weiterer positiver Punkt erscheinen. Negativ fällt auf, dass QoS-Unterstützung nur mit Erweiterungen machbar und daher wohl implementierungsabhängig ist. Der grösste Nachteil dürfte aber bei der im Vergleich zu H.323 mangelnden Verbreitung zu suchen sein.

1.2.5 RTP/RTCP

So viele Auffassungen über Signalisierung es auch gibt, soviel Einigkeit gibt es bei der Wahl des Medientransport-Protokolls. In diesem Gebiet hat sich das Realtime Transport Protocol (RTP) mit seiner Ergänzung, dem Realtime Transport Control Protocol (RTCP) etabliert.

Da es bei Realtime-Anwendungen wichtiger ist, dass die Daten möglichst schnell ausgetauscht werden können, als dass sie fehlerfrei und komplett übertragen werden, wurde RTP als verbindungsloses Protokoll konzipiert. RTCP übernimmt dabei die Übermittlung von Statistiken zum RTP-Verkehr zwischen den Endpunkten. Obwohl RTP theoretisch auf beliebigen Netzwerkprotokollen (und mit Einschränkungen auch auf Tranportprotokollen, siehe [RFC 1889], Section 10) aufgesetzt werden kann, ist doch meist UDP das darunterliegende Protokoll. Das RFC schlägt für diesen Fall vor, für den RTP-Port eine gerade Nummer zu wählen und dem dazugehörigen RTCP-Port die nächsthöhere (ungerade) Nummer zuzuweisen. Ein anderer Grund für die Verbindungslosigkeit von RTP ist die Unterstützung von Multicast-Anwendungen wie Videokonferenzen. Die Architektur

des Protokolls erlaubt es, sowohl Uni- als auch Multi- oder Broadcast-Übertragungen durchzuführen. Das Aufsetzen des Calls, vor allem bei Multicast nicht trivial, bleibt dabei allerdings einem Signalisierungsprotokoll überlassen.

Der Name RTP täuscht über einen wichtigen Aspekt hinweg, nämlich die Tatsache, dass RTP nicht ein fixes Protokoll ist, sondern viel eher ein Protokoll-Framework. Um in einer wirklichen Anwendung benutzt werden zu können, muss RTP mit einer so genannten *Profil-Spezifikation* ergänzt werden, welche mögliche Nutzlastformate und deren Identifikationscodes definiert. Ausserdem können in einem Profil auch Erweiterungen und änderungen von RTP definiert werden. Optional kann eine Profil-Spezifikation auch mit Nutzlastformat-Spezifikationen ergänzt werden, die die zu transportierenden Kodierungen genauer definieren.

Obwohl Quality of Service (QoS) bei Realtime-Anwendungen ein zentraler Punkt ist, enthält RTP keine Funktionalität zur aktiven Unterstützung entsprechender Protokolle, etwa von RSVP. Durch die oben erwähnten Profile kann dem aber in gewissen Masse abgeholfen werden, jedoch werden solche Lösungen an die Applikation oder Applikationsklasse gebunden bleiben.

1.2.5.1 Details von RTP

Endpunkte werden in RTP durch Netzwerkadresse und einen Port definiert, wobei durch erstere der MG und durch letzeren der Endpunkt innerhalb des MGs bezeichnet wird. Die Bedeutung beider Begriffe ist dabei analog zu der aus UDP/IP bekannten Bedeutung. Um die Struktur der Verbindungen flexibel zu halten, werden die Endpunkte innerhalb von RTP allerdings mittels eines Synchronization Source Identifiers (SSRC), einer 32 Bit breiten Nummer, gekennzeichnet. Dadurch wird u.a. das Zusammenfassen mehrerer Pakete in eines ermöglicht, wobei die einzelnen SSRCs in einer Liste im Paket aufgelistet werden. Die Elemente einer solchen Liste heissen Contributing Source Identifier (CSRC).
Zwei protokollspezifische Konzepte sind das der *RTP-Sitzung* und das des *Endsystems*. Eine RTP-Sitzung ist definiert als die “Assoziation zwischen einer Menge von Teilnehmern, die mit RTP kommunizieren”. Für jeden Teilnehmer definiert sich die Sitzung durch zwei Ziel-Transportadressen, je eine für die RTP- und RTCP-Pakete. Ein Endsystem ist “eine Applikation, die den Inhalt zu sendender Pakete generiert und/oder den Inhalt empfangener Pakete konsumiert.

Um einen Überblick über die in einem RTP-Paket vorhandenen Informationen zu erhalten, bietet es sich an, zuerst einen Blick auf das Paketformat zu werfen.

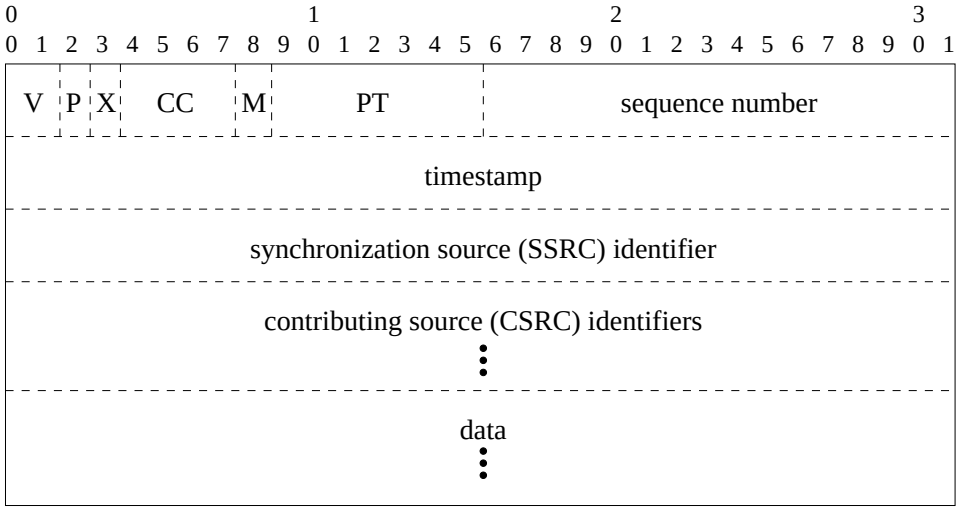


Abbildung 1.5: RTP-Paket-Format

Die ersten zwölf Oktetts sind in jedem RTP-Paket enthalten, die CSRC-Liste hingegen ist optional und nur vorhanden, wenn das Paket von einem Mixer gesendet wurde. Die einzelnen Felder sind (gemäss [RFC 1889]) wie folgt definiert:

Version (V): 2 Bits
: Die Nummer der verwendeten RTP-Version. In neueren RTP-Implementationen ist dieses Feld auf 2 gesetzt.

Padding (P): 1 Bit
: Falls dieses Bit gesetzt ist, wurden den Daten weitere Oktetts angehängt. Dies kann für Verschlüsselungsalgorithmen notwendig sein, die eine gewisse Blockgrösse verlangen. In diesem Fall steht im letzten Oktett des Pakets die Anzahl zu ignorierender Oktetts.

Extension (X): 1 Bit
: Zeigt an, dass der Paket-Header erweitert ist. Die Erweiterung wird zwischen CSRC-Liste und Daten eingefügt und hat profil-spezifisches Format.

CSRC Count (CC): 4 Bits
: Enthält die Anzahl Einträge in der CSRC-Liste.

Marker (M): 1 Bit
: Die Interpretation dieses Bits ist profilabhängig. In vielen Anwendungen bezeichnet es den Beginn eines Frames in Videodaten.

Payload Type (PT): 7 Bits

Bezeichnet das Format der im Paket enthaltenen Daten. Die Interpretation dieses Feldes wird im Profil definiert, kann aber auch dynamisch festgelegt werden, etwa durch das Signalisierungsprotokoll. Für einige übliche Kodierungsverfahren gibt es Vorschläge, die im IETF-Dokument draft-ietf-avt-profile spezifiziert sind. Dieses Feld sollte nicht dazu verwendet werden, verschiedene Medienströme (wie Audio und Video) zu multiplexen. Dazu sollten verschiedene Sitzungen verwendet werden.

Sequenznummer: 16 Bits

Dient dazu, die ursprüngliche Reihenfolge beim Empfang wieder herzustellen. Die Sequenznummer sollte mit einem möglichst unvorhersehbaren Wert initialisiert und für jedes Paket um eins erhöht werden.

Timestamp: 32 Bits

Der Zeitstempel des ersten im Paket enthaltenen DatenOktetts. Bei Audioübertragungen wird dazu üblicherweise die Sampleclock verwendet, d.h. wenn in jedem Paket 64 Samples übertragen werden, sollte auch der Timestamp jedesmal um 64 erhöht werden. Dieser Wert kann über mehrere Pakete derselbe bleiben, falls alle Pakete denselben Zeitpunkt repräsentieren (mehrere Teile eines Videoframes). Wenn die Daten nicht in derselben Reihenfolge gesendet werden, in der sie aufgenommen wurden, müssen die Timestamps nicht einmal monoton wachsen (dies kann beispielsweise bei MPEG der Fall sein).

SSRC: 32 Bits

Identifiziert den Sender des Pakets (die “Synchronisationsquelle”). Diese Zahl wird im Zufallsverfahren bestimmt, was eine kleine aber vorhandene Gefahr von Kollisionen birgt. RTP-Implementationen müssen daher solche Situationen erkennen und beheben können.

CSRC List: 0 bis 15 Einträge à 32 Bits

Identifiziert die ursprünglichen Quellen, wenn das Paket von einem Mixer gesendet wurde. Die Anzahl der Einträge wird durch das CC-Feld bestimmt.

1.2.5.2 Details von RTCP

RTP alleine ist zwar hinreichend für eine simple Übertragung von Echtzeitdaten, genügt aber nur den Anforderungen sehr einfacher Anwendungen. Meist ist es vonnöten, die Qualität einer laufenden Übertragung zu überwachen und gegebenenfalls korrigierend einzugreifen, beispielsweise durch wechseln des Kodierungsschemas der zu sendenden Daten. Um diese und einige andere Aufgaben

anzugehen gibt es RTCP. Dessen hauptsächliche Funktion ist die Rückmeldung über die Qualität der empfangenen Daten. Weitere Funktionen sind die Assoziation mehrerer RTP-Sitzungen (z.B. der Video- und Audio-Sitzungen einer Videokonferenz), und die Verbreitung allgemeiner Informationen über einen Teilnehmer bei “offenen” Sitzungen.
Da RTCP-Pakete über dasselbe Medium gesendet werden wie die RTP-Pakete, muss klar definiert werden, wie Daten- und Kontrollverkehr die Bandbreite unter sich aufteilen. [RFC 1889] schlägt vor, 5% der Sitzungsbandbreite für den RTCP-Verkehr zu reservieren, sowie das Intervall zwischen zwei RTCP-Paketen auf minimal 5 Sekunden zu halten.
Nachfolgend wird auf die einzelnen Pakete und ihre Inhalte genauer eingegangen.

Receiver Report (RR)

Receiver-Report-Pakete dienen der Rückmeldung von Empfangsstatistiken, wie die Anzahl verlorener Pakete und die Unregelmässigkeit ihrer Ankunft, an den Sender. Zusammen mit den Sender-Report-Paketen kann mittels der enthaltenen Informationen auch die Roundtrip-Zeit zwischen Sender und Empfänger geschätzt werden.

Das Paket zerfällt in drei Teile: Header, Report-Blöcke und profil-spezifische Erweiterungen. Die Felder der ersten beiden werden im folgenden näher erläutert.

Header

Version (V): 2 Bits
: Protokollversion, äquivalent zum Feld in RTP-Paketen. Dieses Feld hat den konstanten Wert 2.

Padding (P): 1 Bit
: Falls dieses Bit gesetzt ist, wurden den Daten weitere Oktetts angehängt. Siehe auch Seite 15.

Reception Report Count (RC): 5 Bits
: Die Anzahl im Paket enthaltener Reception-Report-Blocks, wobei 0 eine gültige Angabe ist.

Pakettyp (PT): 8 Bits
: Enthält die Konstante 201, um das RTCP-Paket als Receiver-Report-Paket zu kennzeichnen.

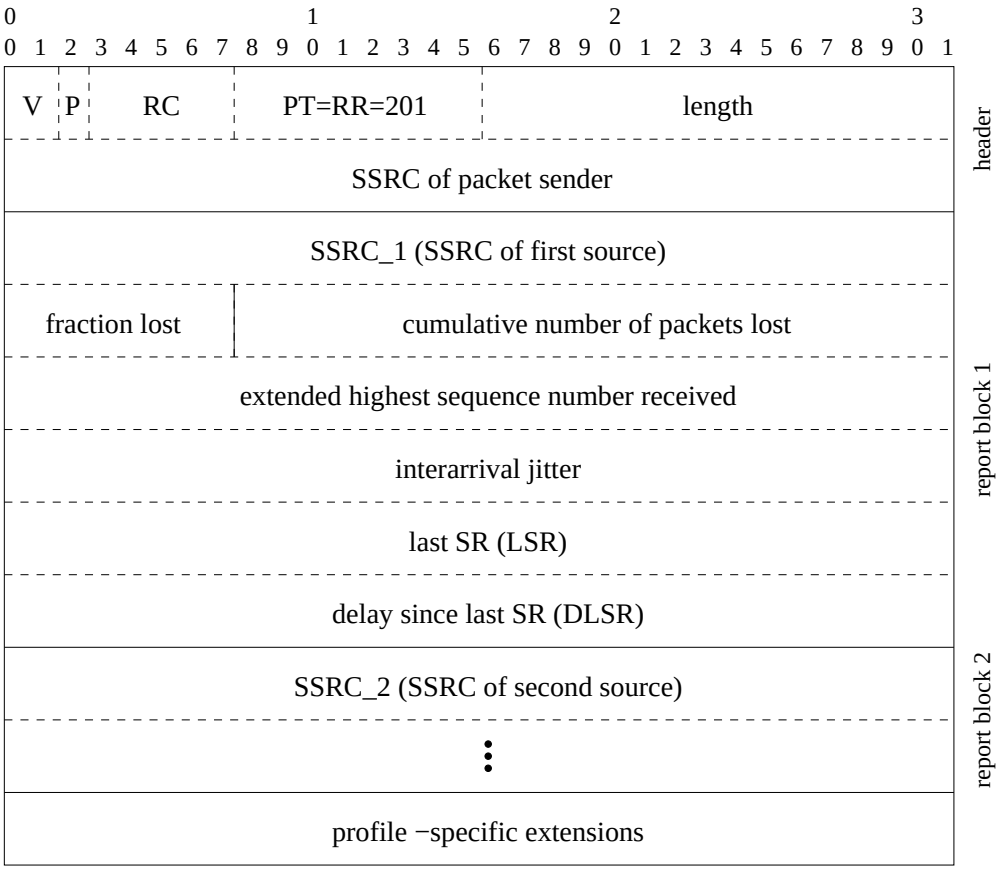


Abbildung 1.6: RR-Format

Length: 16 Bits

Dieses Feld enthält die Länge des Pakets in 32Bit-Wörtern minus 1, inklusive Header und Padding. 0 ist also ein zulässiger Wert für dieses Feld.

SSRC: 32 Bits

Identifiziert den Sender des Pakets, analog zum Feld in RTP-Paketen. Siehe auch Seite 16.

Dem Header folgen 0-31 *Reception-Report-Blöcke*, abhängig vom Wert des RC-Feldes im Header, deren Felder folgende Bedeutung haben:

SSRC_n (Quellenidentifikation): 32 Bits

Bezeichnet die Synchronisationsquelle, auf die sich die Empfangsstatistiken im Report-Block beziehen.

Fraction lost: 8 Bits

Das Verhältnis zwischen der Anzahl verlorener und der Anzahl erwarteter RTP-Pakete von Quelle SSRC_n, seit das letzte RR- oder SR-Paket gesendet wurde, dargestellt als Fixkommazahl mit 0 Bits vor und 8 Bits nach dem Komma. Das bedeutet, dass die möglichen Werte im Intervall $[0, \frac{255}{256}]$ im Abstand von $\frac{1}{256}$ liegen. Falls während des letzten Reportintervalles kein Paket von Quelle SSRC_n empfangen wurde, sollte auch kein Report geschickt werden.

Totale Anzahl verlorener Pakete: 24 Bits

Enthält die Anzahl verlorener RTP-Pakete von Quelle SSRC_n seit Beginn der Übertragung, definiert durch die Differenz zwischen erwarteten und tatsächlich erhaltenen Paketen. Die Anzahl erwarteter Pakete ergibt sich direkt aus der erweiterten höchsten erhaltenen Sequenznummer, die im nächsten Punkt erläutert wird. Verspätete Pakete und Duplikate müssen als erhalten gewertet werden, was auch zu einem negativen Wert führen kann.

Erweiterte höchste erhaltene Sequenznummer: 32 Bits

Eine um 16 Bit erweiterte Sequenznummer. Die unteren 16 Bits entsprechen der höchsten (genauer: der neuesten) empfangenen Sequenznummer, während die oberen 16 Bits die Anzahl Sequenznummerzyklen enthalten.

Jitter: 32 Bits

Dieses Feld enthält eine Schätzung der statistischen Varianz der Zwischenankunftszeit zwischen den RTP-Paketen, gemessen in RTP-Timestamp-Einheiten. Definiert ist dieser Wert (im Folgenden J) durch die Standardabweichung der Differenz D der Paketabstände bei Sender und Empfänger. Formell:

Sei S_i der RTP-Timestamp von Paket i und R_i die Ankunftszeit dieses Pakets in RTP-Timestamp-Einheiten. Dann ist für zwei Pakete i und j

$$D(i,j) = (R_j - R_i) - (S_j - S_i) = (R_j - S_j) - (R_i - S_i)$$

Weiter ist der Schätzwert J_i für jedes empfangene Paket i mit $i \geq 0$ definiert als

$$J_i = \begin{cases} 0, i = 0 \\ J_{i-1} + \frac{|D(i-1,i)| - J_{i-1}}{16}, sonst \end{cases}$$

Der gesendete Wert J ist immer der zuletzt berechnete.

Last SR-Timestamp (LSR): 32 Bits

Die mittleren 32 Bits aus dem 64 Bit breiten NTP-Timestamp des letzten von SSRC_n empfangenen Sender-Report-Pakets. Falls noch kein Paket von dieser Quelle empfangen wurde, wird dieses Feld auf Null gesetzt.

Delay since last SR (DLSR): 32 Bits

Die vergangene Zeit, seit das letzte Sender-Report-Paket von SSRC_n empfangen wurde, in $\frac{1}{65536}$ Sekunden. Auch dieses Feld wird auf Null gesetzt, falls noch kein SR-Paket von dieser Quelle empfangen wurde.

Sender Report (SR)

Teilnehmer an einer RTP-Sitzung, die aktiv Daten senden, verwenden zur Übermittlung von Statistiken Sender-Report-Pakete, welche im Unterschied zu Receiver-Report-Paketen zusätzlich Informationen über die gesendeten Daten enthalten. Ihr Format sieht wie folgt aus:

Offensichtlich ist die Struktur fast identisch zu derjenigen in Receiver-Report-Paketen, mit dem Unterschied, dass sich zwischen Header und Report-Blöcken ein Abschnitt mit zusätzlichen Informationen befindet. Die Bedeutung der einzelnen Felder ist wie folgt.

NTP Timestamp: 64 Bits

Zusätzlich zum RTP-Timestamp, dessen Werte und Auflösung je nach transportiertem Medium ändern, bietet der NTP-Timestamp die Möglichkeit, die absolute Sendezeit des Reports zu übermitteln. NTP steht für Network Time Protocol. Die in diesem Protokoll definierte Zeitdarstellung zählt die Sekunden seit 00:00 Uhr am 1. Januar 1900 (UTC) als 64bit-Fixkommazahl mit 32 Bits nach dem Komma.

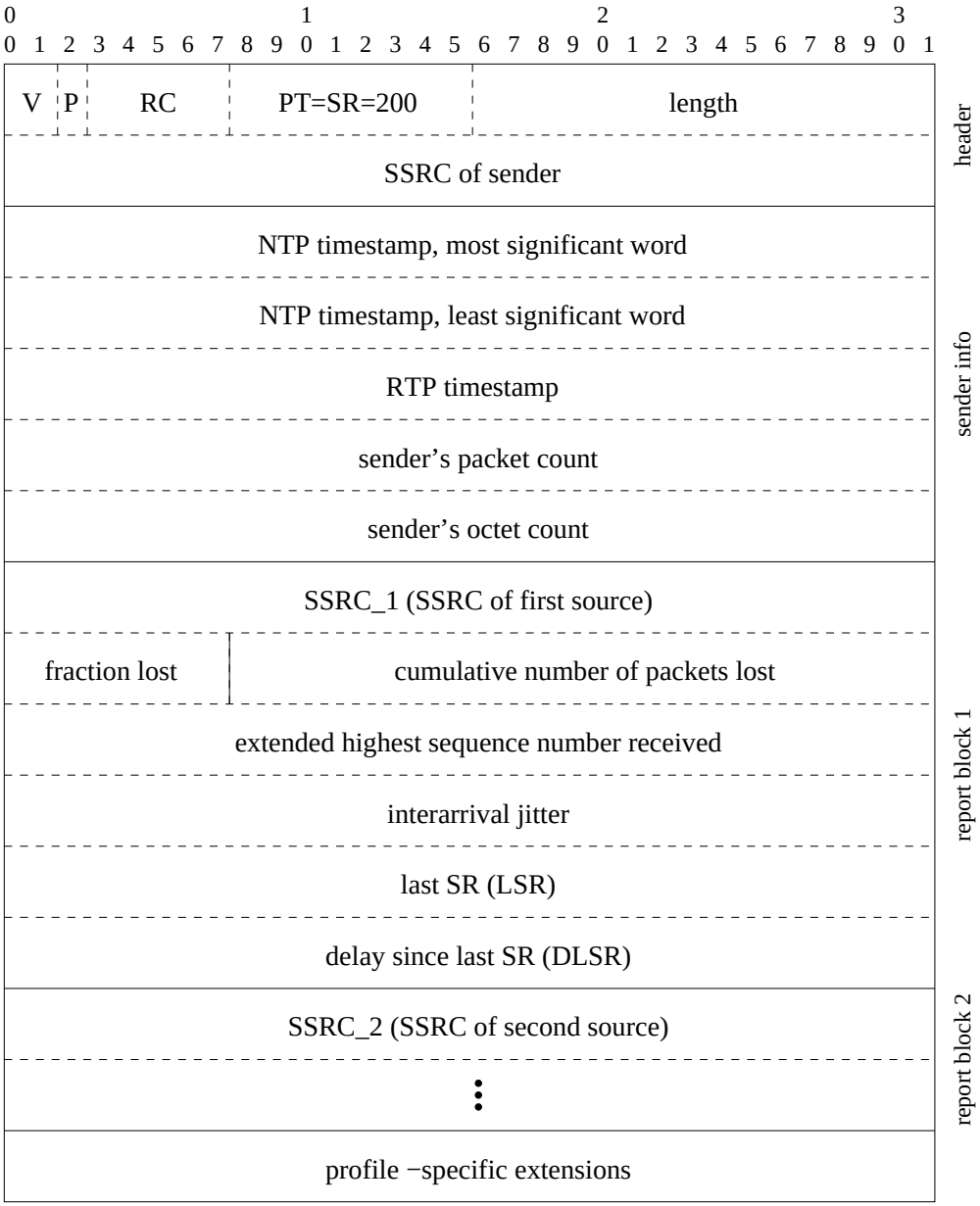


Abbildung 1.7: SR-Format

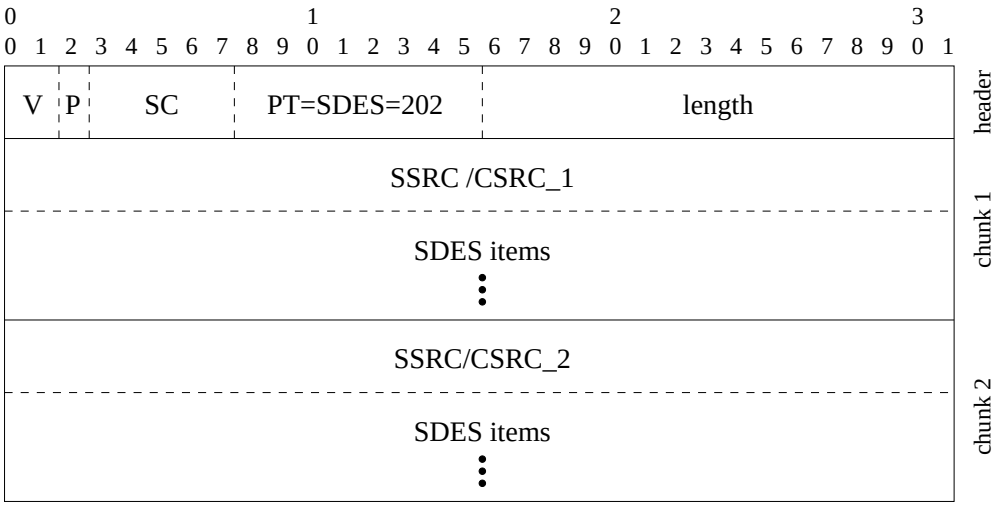


Abbildung 1.8: SDES-Format

RTP Timestamp: 32 Bits

Diese Zeitangabe stellt den gleichen Zeitpunkt dar wie der NTP-Timestamp, benutzt aber dasselbe Format, wie es in RTP-Paketen verwendet wird. Dies kann dazu benutzt werden, mehrere übertragene Medien zu synchronisieren.

Anzahl gesendeter Pakete: 32 Bits

Enthält die Anzahl in dieser Sitzung gesendeter RTP-Pakete, bis zum Zeitpunkt, an dem das SR-Paket generiert wird. Falls der SSRC-Identifikator geändert wird, muss dieser Zähler wieder bei Null beginnen.

Anzahl gesendeter Oktetts: 32 Bits

Die Anzahl gesendeter Payload-Octets (d.h. ohne Header und Padding) bis zu Zeitpunkt der Generierung dieses Pakets. Wie die Anzahl gesendeter Pakete muss dieser Zähler bei einem SSRC-Wechsel zurückgesetzt werden.

Source Description (SDES)

Im Gegensatz zu den SR- und RR-Paketen haben Source-Description-Pakete keinen direkten Zusammenhang mit den übertragenen Daten, sondern vielmehr mit den Teilnehmern einer Sitzung. Das Layout eines SDES-Pakets sieht folgendermassen aus:

Wie einfach ersichtlich ist, haben SDES-Pakete einen verkürzten Header von 4 Bytes, der im Unterschied zu anderen RTCP-Typen keinen SSRC-Identifikator aufweist. Auf diesen Header folgt eine Liste von 0-31 so genannter SDES-Chunks,

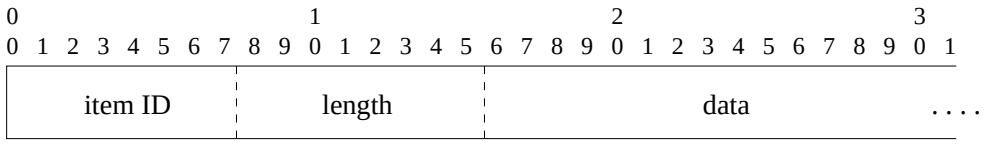


Abbildung 1.9: SDES-Item

die jeweils einen Identifikator und eine Anzahl von SDES-Items enthalten. Diese Items beschreiben je eine Eigenschaft eines Teilnehmers, beziehungsweise der von ihm verwendeten Software. Die üblichen SDES-Items haben alle dasselbe Format: eine 8 Bit lange Identifikation, die den Item-Subtyp kennzeichnet, ein weiteres 8 Bit breites Feld, das die Länge der folgenden Daten in Oktetts angibt und schliesslich die Daten.

Folgende Subtypen sind in [RFC 1889] vordefiniert (in Klammern die dazugehörigen ID-Werte):

Kanonischer Endpunktidentifikator CNAME (1)
: Dieser sollte algorithmisch erzeugt werden und einen Teilnehmer auch zwischen Sitzungen eindeutig identifizieren. Dies kann unter anderem dazu verwendet werden, die SSRC-Identifikatoren aus verschiedenen Sitzungen zu assoziieren, um beispielsweise zwei Ströme mit Ton- und Videodaten in einer Videokonferenz-Anwendung einem Teilnehmer zuordnene zu können.

NAME (2)
: Der echte Name eines Teilnehmers, für die anderen menschlichen Teilnehmer an einer Sitzung gedacht. Dies im Gegensatz zum oben erwähnten CNAME-Item, das nur zur Verarbeitung innerhalb der Anwendung gedacht ist.

EMAIL (3)
: Email-Adresse des Teilnehmers, im üblichen Format.
(Bsp: hans.muster@beispiel.ch)

PHONE (4)
: Telefonnummer des Teilnehmers, im internationalen Format, d.h. mit einem Pluszeichen anstatt der landesabhängigen internationalen Vorwahl.

LOC (5)
: Der Aufenthaltsort. Die Genauigkeit dieser Angabe ist vom Kontext der Anwendung abhängig, innerhalb eines Gebäudes könnte es beispielsweise die Raumnummer sein.

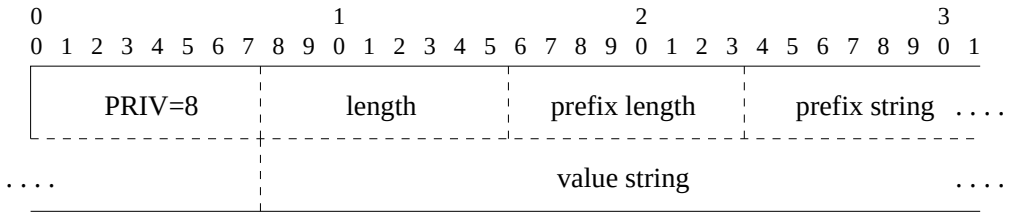


Abbildung 1.10: PRIV-Item

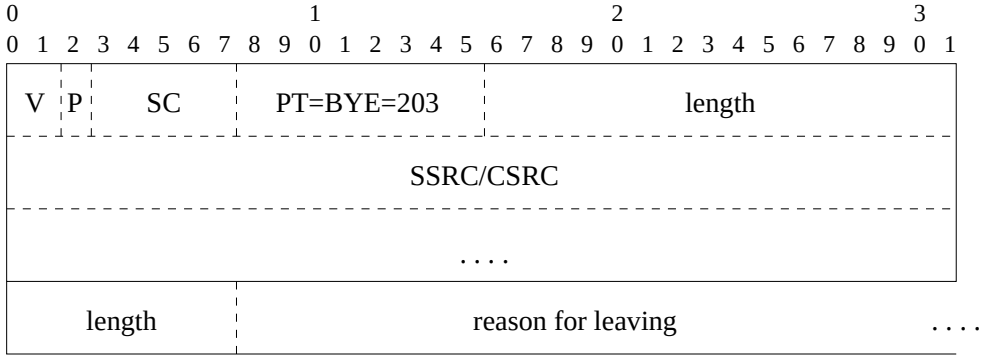


Abbildung 1.11: BYE-Format

TOOL (6)
Ein String, der die verwendete Applikation identifiziert.

NOTE (7)
Eine beliebige Notiz des Teilnehmers, zum Beispiel eine Nachricht wie “bin gleich zurück”.

Für experimentelle oder anwendungsinterne Zwecke gibt es ein erweitertes Item-Format mit der Identifikation PRIV (8), das zusätzlich zwei Felder zur Unterscheidung verschiedener solcher Items aufweist.

End of Participation (BYE)

Wenn eine Partei ihre Teilnahme an einer Sitzung beendet, zeigt sie dies mit einem BYE-Paket an. Das Format besteht aus einem 4 Byte langen Header, einer SSRC-Liste mit Identifikatoren für alle betroffenen Endpunkte und einem optionalen String, der den Grund für die Beendigung angibt.

Application Specific (APP)

Für applikationsspezifische Informationen können im jeweiligen Profile APP-Pakete mit folgendem Grundformat definiert werden.

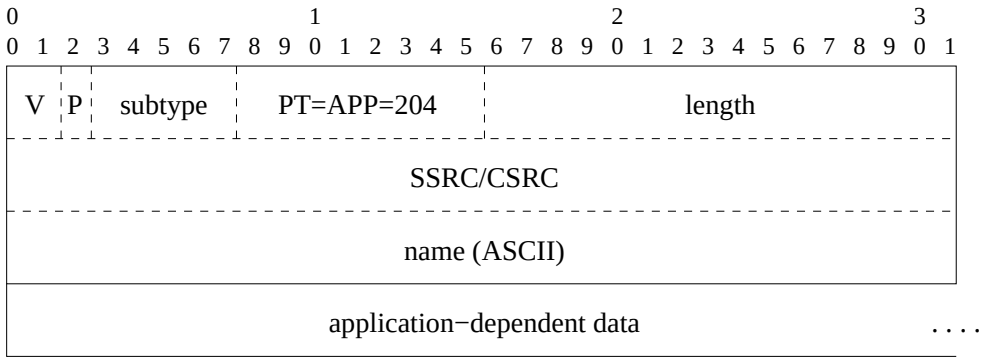


Abbildung 1.12: APP-Format

Version, Padding, Length
Diese Felder sind äquivalent zu denen in RR-Paketen.

Subtype: 5 Bits
Identifiziert den Subtyp eines APP-Pakets innerhalb einer Applikation.

Name: 4 Oktetts
Ein im Profil festgelegter Name, der zu einem Profil gehörende APP-Pakete identifiziert. Dies ist notwendig, da eine Anwendung auch “fremde” APP-Pakete erhalten könnte und diese als solche erkennen muss.

Applikationsabhängige Daten
Der Inhalt dieses Feldes muss im Profil definiert werden. Die Länge ist, innerhalb der durch das Netzwerkprotokoll gegebenen Grenzen, beliebig.

1.2.5.3 Roundtrip-Berechnung mit SR/RR-Paketen

Wenn ein Sender ein SR- oder RR-Paket erhält, dessen LSR-Feld sich auf ein von ihm gesendetes SR-Paket bezieht, kann die Roundtripzeit (Umlaufzeit) zwischen ihm und dem antwortenden Endpunkt folgendermassen berechnet werden.
Sei t_s der Zeitpunkt, in dem ein SR-Paket an den anderen Endpunkt gesendet wird, und t_e der Zeitpunkt, in dem das Antwortpaket empfangen wird. Die Zeit zwischen Empfang des ersten SR-Pakets und Senden der Antwort wird durch das DLSR-Feld übermittelt. Die Roundtripzeit lässt sich nun berechnen durch

$$Roundtrip = (t_e - t_s) - DLSR.$$

Abb. 1.13 zeigt den Ablauf schematisch. Es gilt zu beachten, dass die Verzögerungen in beide Richtungen häufig asymmetrisch sind, wie es auch im Schema zu erkennen ist.

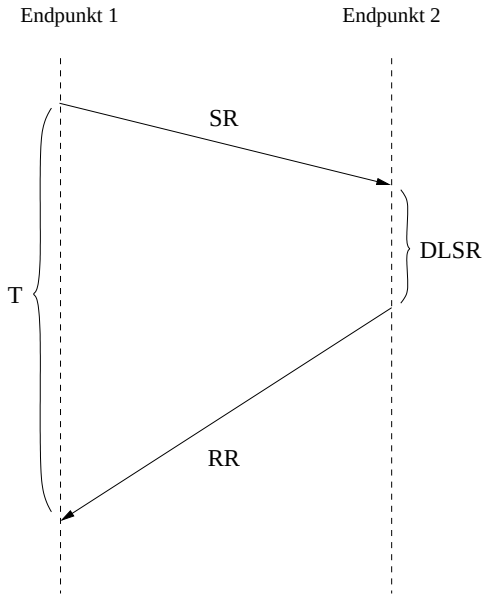


Abbildung 1.13: Roundtrip-Berechnung

1.2.5.4 Weitere Eigenschaften von RTP

Mixer und Übersetzer

Die grundlegende Funktionalität von RTP genügt völlig, wenn sie zum Beispiel innerhalb eines Intranetzes mit genügend grossen Bandbreiten eingesetzt wird. In grösseren Netzen wie dem Internet, ist es wahrscheinlich, dass viele Verbindungen eine zu kleine Bandbreite aufweisen, um alle Pakete einer Konferenz rechtzeitig empfangen zu können, oder dass sie durch eine Firewall vom RTP-Verkehr abgeschnitten sind.
Um diese Probleme zu beheben, wurden *Übersetzer* (Translatoren) und *Mixer* eingeführt. Übersetzer leiten Pakete weiter, ohne ihr SSRC-Feld zu verändern und sind transparent für die RTP-Endsysteme, da sie keine neue Synchronisationsquelle darstellen. Sie können allerdings die Datenkodierung ändern, mehrere Pakete zu einem zusammenfassen, oder Pakete ver- und entschlüsseln. Mit einem Translator lässt sich also das Firewall-Problem einfach beheben.
Die Funktion eines Mixers lässt sich schon aus dem Namen schliessen: Er fasst RTP-Pakete von mehreren Quellen zu einzelnen zusammen, bevor er sie weiterschickt. Wenn er im “Zentrum” einer Konferenz steht (wie beispielsweise die MCU bei H.323-Konferenzen), kann der Mixer die Daten aller Teilnehmer zusammenfassen und zurückschicken, so dass die Endsysteme nur je einen Datenstrom empfangen und senden müssen. Da durch die Kombination verschiedener RTP-Ströme ein neuer Strom erzeugt wird, erhält ein RTP-Mixer einen eigenen SSRC-

Identifikator und versendet seine eigenen RTCP-Pakete. Im Gegensatz dazu leitet ein Translator diese Pakete nur weiter (mit Ausnahme von SR/RR-Paketen, deren Werte er gegebenenfalls korrigieren muss).

Durch Kombination von Übersetzern und Mixern kann die Flexibilität noch erhöht werden. Obiges Beispiel könnte folgendermassen erweitert werden: Der Multicast-Gruppe, an die der Mixer G.711-Daten sendet, wird ein Translator hinzugefügt. Dieser komprimiert die Daten, indem er die Kodierung nach G.729 übersetzt und leitet sie an eine andere Multicast-Gruppe weiter. Damit hätte man zwei Multicast-Gruppen, eine für Endsysteme mit grosser und eine für Endsysteme mit kleiner Bandbreite.

Standardprofil

Speziell für “Audio- und Videokonferenzen mit minimaler Kontrolle” wurde in [RFC 1890] ein Profil definiert, das bei offenen Konferenzen nützlich sein kann, in denen keine Medien- und Mitgliederparameter ausgehandelt werden. Beim definieren eigener Profile kann diese Spezifikation als Grundlage dienen.
Für Header-Formate, Sende-Intervalle und dergleichen werden die in [RFC 1889] vorgeschlagenen Werte übernommen. Spezifischer sind die Angaben hingegen für die transportierten Audiodaten. Unter anderem werden die möglichen Abtastraten, die Reihenfolge bei mehreren Audiokanälen, das Verhalten des Markerbits bei frame-basierten Kodierungen und die übliche Paketrate festgelegt. Der wichtigste Teil ist aber wohl die vorgeschlagene Abbildung von Payload-IDs auf Audiokodierungen, welche in Tabelle 1.1 zu sehen ist. Die IDs einiger Typen werden per Signalisierung festgelegt, sind also in der Tabelle nicht enthalten.

1.2.6 Kodierungsverfahren

Für Audio- und speziell Sprachdaten existieren eine Vielzahl von Kodierungen, die alle ihre Vor- und Nachteile aufweisen. Analog-Digital-Converter (ADCs) liefern die diskretisierte Darstellung des Audiosignals meist in linearer PCM-Kodierung. PCM-Formate sind denn auch die grundlegenden und einfachsten verfügbaren Formate. Komplexere Kodierungen nützen spezielle Eigenschaften des Signals aus. Im Allgemeinen können Audiokodierungen nach den Gesichtspunkten der benötigten Bitrate und der Qualität des resultierenden Signals bewertet werden. Wichtig dabei ist, dass die Qualität durch das (subjektive) menschliche Gehör und nicht durch die (objektive) mathematische Abweichung vom Originalsignal gegeben ist. Man spricht von *perzeptueller Qualität*.

Name	Typ	Bittiefe	ms/frame	Hz	ID	Bemerkung
1016	Frame		30	8000	1	aka CELP
DVI4	Sample	4		8000	5	aka IMA ADPCM
				16000	6	
G.721	Sample	4		8000	2	
G.722	Sample	8		8000	9	
G.728	Frame		2.5	8000	15	
GSM	Frame		20	8000	3	
L8	Sample	8		8000		unsig. 8bit linear
L16	Sample	16		44100	11	signed 16bit linear
					10	stereo
LPC	Frame		20	8000	7	
MPA	Frame			90000	14	bedeutet MPEG-1 oder MPEG-2
PCMA	Sample	8		8000	8	aka G.711 A-Law
PCMU	Sample	8		8000	0	aka G.711 μ-Law
VDVI	Sample	var		8000		Variable-rate DVI4

Tabelle 1.1: RTP-Audio-Kodierungen

Eine wichtige Unterscheidung zwischen den Kodierungen, vor allem im Hinblick auf die Verwendung in VoIP, ist diejenige zwischen sample- und framebasierten Methoden. Bei samplebasierten Kodierungen besteht keine Abhängigkeit zwischen den einzelnen Samples, was den Vorteil hat, dass der Datenstrom beliebig unterteilt und somit die Paketlänge fast beliebig gewählt werden kann. Framebasierte Kodierungen fassen Samplegruppen meist fixer Länge zu Frames zusammen und kodieren diese, indem sie die Abhängigkeiten zwischen den einzelnen Werten ausnutzen. Hiermit lassen sich bessere Kompressionsraten als mit samplebasierten Verfahren erzielen, die Paketlänge hingegen ist fix gegeben, was auch die minimale Verögerung auf einen gewissen Wert festlegt. PCM-Kodierungen sind normalerweise samplebasiert, die meisten anderen framebasiert.

1.2.6.1 PCM

Pulse-Code-Modulation (PCM) rekonstruiert das Signal aus einer Reihe von Abtastwerten (Samples) mit fixem zeitlichen Abstand. Die Qualität dieser Rekonstruktion hängt von zwei Faktoren ab: Der Anzahl Abtastwerte pro Sekunde (Samplerate) und der Genauigkeit dieser Werte (Bittiefe). Die Samplerate begrenzt die maximale rekonstruierbare Frequenz auf $\frac{Samplerate}{2}$ Hz, während die Bittiefe bestimmt, wie genau die Amplitude des rekonstruierten Signals mit der des Originals übereinstimmt. Bei genügender Auflösung kann mit PCM eine sehr

hohe Qualität erreicht werden, was sich jedoch negativ auf die Bitrate auswirkt. Audio-CDs verwenden beispielsweise eine Samplerate von 44.1 kHz bei 16 Bit breiten Samples.

Die im PSTN verwendeten und in [G.711] definierten Kodierungen A-Law und μ-Law verwenden eine Bittiefe von 8, verbessern aber die perzeptuelle Qualität, indem sie die Abtastwerte logarithmisch Gewichten. Damit können die Abschnitte des Signals mit kleiner Amplitude genauer rekonstruiert werden als diese mit grosse Amplitude, was der Tatsache rechnung trägt, dass leise Geräusche genauer wahrgenommen werden als laute. Die folgende Formel zeigt, wie ein linearer PCM-Wert nach μ-Law konvertiert wird (m sei der lineare Abtastwert, mp dessen mögliches Maximum. Weiter sei $u = 100$ oder $u = 255$).

$$y = \frac{sgn(m)}{ln(1+u)}\, ln\left(1 + u\left|\frac{m}{mp}\right|\right)$$

Die entsprechende Formel für A-Law ist etwas komplizierter, folgt aber derselben Grundidee. Statt u wird hier $A = 87.6$ verwendet:

$$y = \begin{cases} \frac{A}{1+lnA}\left(\frac{m}{mp}\right), & \left|\frac{m}{mp}\right| \leq \frac{1}{A} \\ \frac{sgn(m)}{1+lnA}\left(1 + lnA\left|\frac{m}{mp}\right|\right), & \frac{1}{A} \leq \left|\frac{m}{mp}\right| \leq 1 \end{cases}$$

Wie aus Abb. 1.14 ersichtlich, sind die Abstufungen bei kleinen Abtastwerten überproportional kleiner.

Differentielles PCM

Audiosignale sind stetig und haben meist einen “sanften” Verlauf. Somit kann man, wenn n Abtastwerte bekannt sind, den $n + 1$-ten mit einiger Genauigkeit voraussagen. Eine einfache Kodierung, die diese Eigenschaft ausnützt, ist *Differential PCM* (DPCM): Dem ersten Abtastwert folgen nur noch Differenzen zum vorhergehenden Wert. Als Beispiel könnte der Abtastwert 8 Bit breit sein, die Differenzen jedoch nur noch 4 Bit, womit sich bei längeren Signalen eine Verringerung der Bitrate um Faktor 2 ergibt. Die Qualität dieser Methode hängt sehr von der Güte der Voraussage ab, da sonst die 4 Bit der Differenz zur Korrektur nicht ausreichen und die resultierende Wellenform Fehler aufweist.

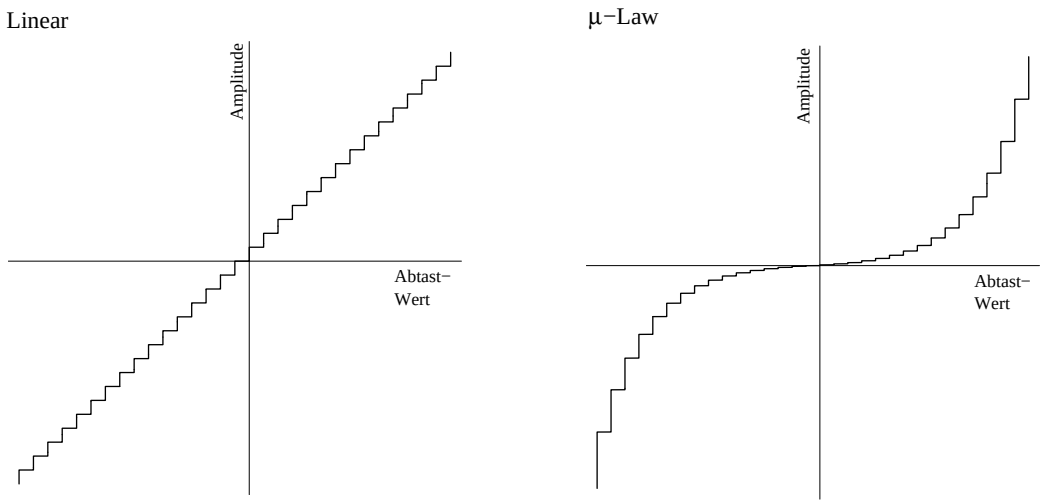


Abbildung 1.14: Amplitudenkurven für lineares und μ-Law-PCM bei Bittiefe 5

Das Verhältnis zwischen Qualität und Bitrate dieser Methode kann durch *Adaptive Differential PCM* (ADPCM) noch verbessert werden. Die Differenzen werden dynamisch skaliert und gewichtet, je nachdem, wie sich das Signal kurz vorher verhalten hat. Wenn beispielsweise das Signal grössere Schwankungen in der Amplitude aufweist, wird auch die Skalierung erhöht, da auch mit grösseren Abweichungen vom vorhergesagten Wert gerechnet werden muss. Der für die Vorhersage und Skalierung verwendete Algorithmus muss nicht nur dem Decoder, sondern auch dem Encoder bekannt sein, da das Signal natürlich nicht mit anderer Skalierung kodiert werden sollte, als es es dekodiert wird. Durch diese Verbesserung wird die Qualität bei gleichbleibender Bitrate entscheidend verbessert oder bei gleichbleibender Qualität die Bitrate stark gesenkt. Zum Beispiel weist eine ADPCM-Kodierung mit 32 kbps nur unmerklich schlechtere Qualität auf als 64 kbit PCM. Es gibt mehrere ADPCM-Kodierungen. Die bekannteste unter ihnen ist die in [G.726] definierte Version von ITU-T.

1.2.6.2 Perzeptuelle Verfahren

Die Wahrnehmung von Geräuschen durch das menschliche Ohr ist nicht linear und von vielen Faktoren abhängig. Die Signalstärke, die notwendig ist, damit ein Ton wahrgenommen werden kann, ist für alle Frequenzen unterschiedlich. Am empfindlichsten ist das Ohr in der Nähe von 1 kHz. Die Signalstärke, die notwendig ist, damit das Ohr eine gewisse Frequenz wahrnehmen kann, nennt man *Hörschwelle*. Abb. 1.15 zeigt deren ungefähren Verlauf über den Frequenzen. Speziell Sprachsignale bewegen sich nur in einem relativ engen Frequenzband.

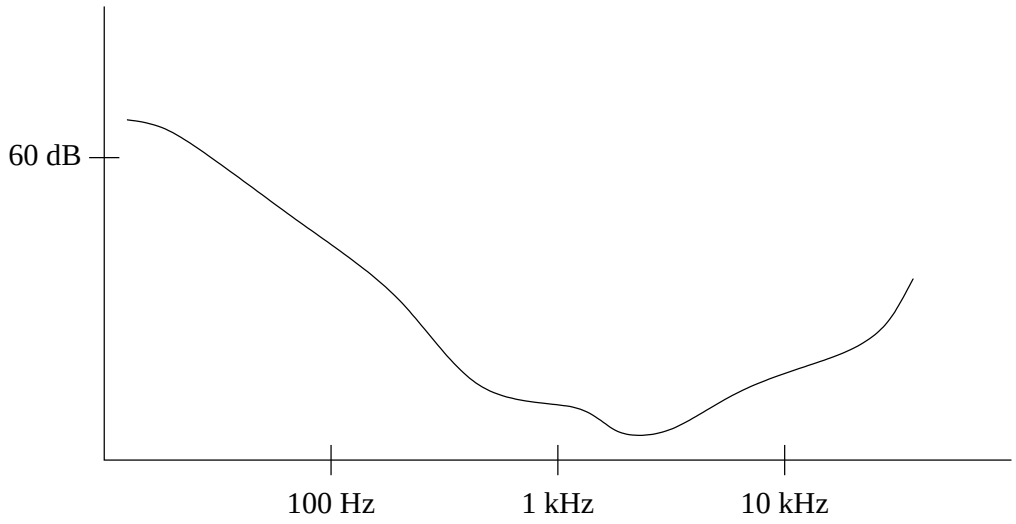


Abbildung 1.15: Die Hörschwelle für verschiedene Frequenzen

Eine andere wichtige Eigenschaft besteht darin, dass durch einen lauten Ton einer bestimmten Frequenz leisere Töne mit benachbarten Frequenzen unhörbar gemacht werden. Dies wird *Maskierung* genannt und kann insofern ausgenutzt werden, als dass die maskierten Frequenzen nicht kodiert werden müssen, weil sie gar nicht wahrgenommen würden.

Wiederum gibt es mehrere Ansätze, diese Eigenschaften auszunutzen. *Subband-Kodierer* berechnen zu Beginn das Frequenzspektrum der zu übertragenden Daten und setzen danach Bandpassfilter für die dominanten Frequenzen ein. Diese bleiben für die Dauer der Kodierung fix und werden dem Decoder zu Beginn der Übertragung mitgeteilt. Weil die von den Filtern kommenden Daten direkt verwendet werden können, ist die Verzögerung beim Kodieren klein, was ein Vorteil sein kann. Der Nachteil ist die begrenzte Anzahl von Filtern, was sich stark auf die Qualität auswirkt.
Adaptive Transformationkodierer analysieren das Spektrum ganzer Frames (z.B. mit Fast Fourier Transform) und übertragen jeweils das Resultat der Analyse. Damit lässt sich eine bedeutend bessere Qualität erreichen, was aber auf Kosten der Verzögerung geht. Diese Methode wird unter anderem auch zum Speichern von Musikdaten verwendet, zum Beispiel in Sonys Minidisc.

Weiter lassen sich *Source-* und *Hybridkodierer* unterscheiden. Sourcekodierer übertragen nur die Parameter des generierten Sprachmodells, was zu starken Verzerrungen führen kann, wenn mehrere Personen gleichzeitig Sprechen. Hybridkodierer fügen den Parametern Teile des Originalsignals hinzu, wodurch Fehler bei der Rekonstruktion stark vermindert werden können. Der bekannteste Vertre-

ter dieser Gattung ist CELP. Die Komplexität dieses Verfahrens ist beträchtlich, weshalb es hier nicht näher betrachtet wird. Siehe [1016] für eine Definition und [Proc94] für eine optimierte Implementierung.

1.2.6.3 Silence Suppression

Während eines Telefonats spricht meist nur eine Partei, weshalb es häufig unsinnig erscheint, die Audiodaten beider Seiten gleichzeitig zu übertragen. In Phasen, in denen eine der Parteien schweigt, kann ihr Datenfluss unterbrochen werden, ohne dass dies das Gespräch stören würde. Man nennt dies *Silence Suppression*.
Um zwischen aktiven Abschnitten, in denen der Anrufer spricht, und passiven unterscheiden zu können, muss ein Lautstärken-Schwellenwert gewählt werden. Sinkt die Lautstärke für eine gewisse Dauer (die Hold-Over-Zeit) unter diesen Wert, wird der Datenfluss unterbrochen. Die genaue Wahl dieses Wertes ist ein wichtiger Faktor, da bei einem zu kleinen Wert die Stille-Phasen nicht korrekt erkannt werden und bei einem zu grossen Teile des Sprachsignals beschnitten werden, was *Clipping* genannt wird (siehe Abb. 1.16).

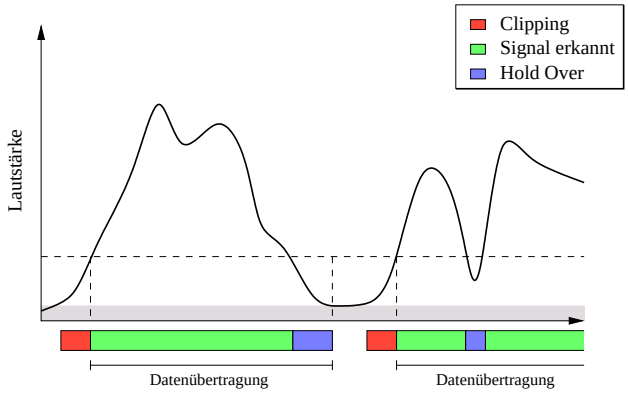


Abbildung 1.16: Clipping

Ein Problem mit Silence Suppression ist der Eindruck einer “toten Leitung”, der bei Signalunterbrüchen entsteht. Um dem entgegen zu wirken, kann auf der Gegenseite in solchen Phasen künstliches Hintergrundrauschen generiert werden, *Comfort Noise* genannt. Die Qualität dieses Rauschens ist dadurch bestimmt, wie sehr es den tatsächlichen Hintergrundgeräuschen auf der Senderseite ähnelt.

Kapitel 2

Probleme mit VoIP und mögliche Lösungsansätze

2.1 Qualitätsfaktoren

Das Ziel einer Telefonanwendung ist es, dem Benutzer so gut wie möglich das Gefühl zu vermitteln, direkt mit dem Gegenüber zu sprechen. Ein zentraler Punkt dabei ist die *Klarheit* der übertragenen Sprache. Es gibt viele Faktoren, die Einfluss darauf haben. Der offensichtlichste davon ist die Qualität der Mikrophone und Lautsprecher auf beiden Seiten. Die Digitalisierung und Kodierung des Signals ist ein weiterer, wichtiger Faktor. Vor allem die Wahl des Kodierungsschemas ist entscheidend für den Rauschabstand und die Verzerrung des Signals. Wenn das Kodierungsschema auf der Strecke zwischen Sender und Empfänger zusätzlich verändert wird (durch einen RTP-Transcoder zum Beispiel) leidet die Qualität zusätzlich. Wird Silence Suppression eingesetzt, haben auch die Güte des Comfort Noise und eventuelles Clipping Einfluss (letzteres beinflusst zwar nicht die Audioqualität als solche, verringert aber die Verständlichkeit, da Silben oder Teile davon verschluckt werden können).
Mit VoIP ergeben sich auch einige neue Probleme. Im nächsten Abschnitt werden diese diskutiert.

2.2 Probleme

Digitale Netze bringen, neben all ihren Vorteilen, auch einige Nachteile mit sich. Da die Korrektheit der übertragenen Daten wichtig ist, muss ein bedeutender Teil der Ressourcen für die Fehlererkennung beziehungsweise Fehlerkorrektur eingesetzt werden. IP-Netze im speziellen haben durch ihre flexible und verbindungslose Struktur einige zusätzliche Eigenheiten, die sich auf die Sprachübertragung

negativ auswirken können. Die drei hauptsächlichen Probleme, Verzögerung (Delay), Verzögerungsschwankungen (Jitter) und Paketverlust (Packet Loss), werden im Folgenden genauer beleuchtet.

2.2.1 Paketverlust

Paketverluste können in IP-Netzen mehrere Gründe haben. Die folgenden sind die wichtigsten:

Übertragungsfehler Daten können bei der Übertragung zwischen zwei Systemen durch Störungen verändert werden, was sie unbrauchbar machen kann, wenn der Empfänger die Daten nicht trotzdem rekonstruieren kann. Das Paket, zu dem die Daten gehören, muss in einem solchen Fall fallgelassen werden.

Stau Da IP verbindungslos ist, kann für einen Router nicht vorausgesagt werden, wie viel Datenverkehr er zu einem bestimmten Zeitpunkt verarbeiten muss. Um auch bei kurzen Überlastungen alle Pakete weiterleiten zu können, hat ein Router eine oder mehrere Warteschlangen (*Queues*) in Form von FIFO-Speichern, in denen Pakete zwischengelagert werden können. Wenn die Überlast aber so lange anhält, dass die Kapazität der Queues nicht mehr ausreicht, müssen Pakete fallengelassen werden. Dies ist bei vielen Netzen der häufigste Grund für Packetverluste.

Misskonfiguration Häufig wird die Infrastruktur eines IP-Netzes nicht zentral verwaltet (das Internet ist das beste Beispiel dafür) und ist daher anfälliger für Misskonfiguration in den Routern. Wenn ein Paket von einem solchen Router in ein falsche Richtung weitergeleitet wird, erreicht es sein Ziel entweder auf Umwegen oder es geht verloren.

IP-Netze sind so ausgelegt, dass solche Paketverluste keine ernstzunehmenden Folgen haben müssen. Durch TCP lassen sich Daten verlustfrei übertragen, auch wenn einzelne der gesendeten Pakete verlorengehen. Leider ist der Preis dafür, dass die Verzögerung stark ansteigt, weshalb TCP für die Sprachübertragung nicht in Frage kommt. Stattdessen wird üblicherweise UDP eingesetzt, das aber das Erkennen von Paketverlusten der Anwendung überlässt.

Paketverlust macht sich meist durch sehr störende “Lücken” in der übertragenen Sprache bemerkbar. Einige Kodierungen versuchen, durch senden redundanter Informationen solche Lücken auszufüllen, allerdings mit niedrigerer Qualität. Wegen den grossen Qualitätseinbussen, die Paketverluste mit sich bringen,

ist es sehr wichtig, diese soweit als möglich unter Kontrolle zu haben. Durch QoS-Mechanismen wie Integrated Services oder Differentiated Services können, zusammen mit entsprechend verwalteter Bandbreite, Paketverluste auf ein Minimum reduziert werden.

2.2.2 Verzögerung

Wenn Audiodaten mit IP gesendet werden, ergibt sich eine Verzögerung zwischen der Aufnahme auf dem sendenden System und dem Abspielen beim Empfänger. Sie entsteht folgendermassen (siehe auch Abb. 2.1):
Die erste Verzögerung ergibt sich durch das Aufnehmen eines Signalabschnitts

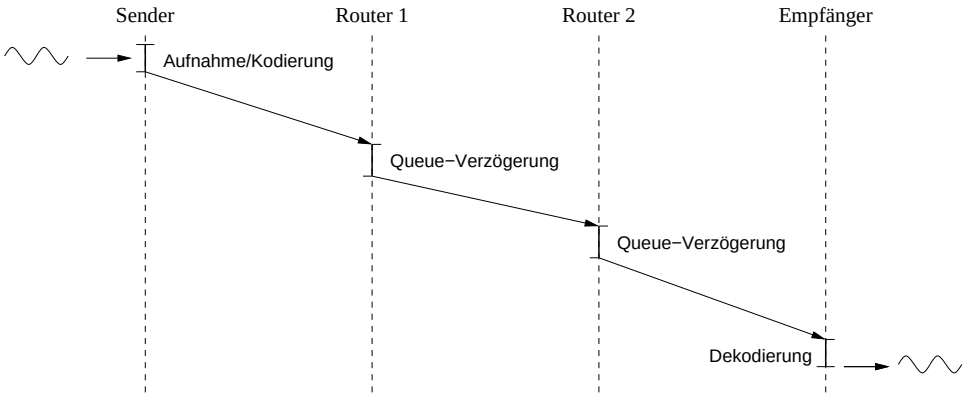


Abbildung 2.1: Entstehen von Verzögerung

und wird bei komplexen Kodierungen durch das aufwendige erstellen eines Kodierungsframes noch grösser. Nun folgt die Übertragung des Pakets, die eine vom Netzwerk abhängige Verzögerung hinzufügt. Schliesslich muss das Signal auf der Gegenseite wieder rekonstruiert werden.
Die am schwierigsten vorauszusagende Verzögerung ist die netzwerkabhängige. Sie hängt von der Bandbreite der Teilstrecken, sowie der Anzahl und Auslastung der Zwischenstationen (Hops) ab. Die durch eine Teilstrecke verursachte Verzögerung ist ungefähr $\frac{Headerlänge+Nutzlast}{Bandbreite}$ (Link-Layer-Protokoll vernachlässigt). Normalerweise muss ein Paket bei jedem Hop vollständig empfangen werden, bevor es weitergesendet werden kann, was man Store-and-Forward-Verzögerung nennt. Je nachdem, wie voll die Queue auf einem Router ist, muss das Paket zusätzlich eine gewisse Zeit warten, bis es weitergeleitet wird.
Sei nun b_i die Bandbreite des i-ten Links, q_i die Queuelänge in Bytes des i-ten Routers (Der Empfänger gilt als letzter Router) und N die Anzahl Links/Router. Weiter sei h die Headerlänge und p die Nutzlastlänge. Dann ergibt sich für die Netzwerkverzögerung nd folgende Formel:

$$nd = \frac{h+p}{b_N} + \sum_{i=1}^{N-1} \frac{h+p}{b_i} + \frac{q_i}{b_{i+1}}$$

Die gesamte Verzögerung ist etwas komplexer. Sei a die Abschnittsgrösse in Samples, r die Samplerate, f die Grösse des kodierten Frames, $k_{\rightarrow}$ und $k_{\leftarrow}$ die Zeiten zur Kodierung und Dekodierung eines Frames und $k_r \leq 1$ die Kompressionsrate der Kodierung. Weiter sollen obige Definitionen weiter gelten. Dann ist die gesamte Verzögerung

$$d = \frac{a}{s} + k_{\rightarrow} + \frac{h + k_r a}{b_N} + \sum_{i=1}^{N-1} (\frac{h + k_r a}{b_i} + \frac{q_i}{b_{i+1}}) + k_{\leftarrow}$$

Es gibt weitere Faktoren, die zur Verzögerung beitragen können, wie etwa Synchronisationspuffer (Jitterbuffer, siehe unten). Diese lassen sich aber durch eine andere Interpretation der Symbole der obigen Formel darstellen. Im Beispiel des Synchronisationspuffers kann etwa $k_{\leftarrow}$ vergrössert werden.

Die Verzögerung, die durch die Aufnahme entsteht, lässt sich natürlich vermindern, indem die Länge der aufgenommenen Abschnitte verkleinert wird. Dies lässt sich aber nur innerhalb enger, durch Bandbreite und Headerlänge gegebener Grenzen realisieren. Abb. 2.2 zeigt die benötigte Bandbreite (in Byte/s) in Abhängigkeit zur Nutzlastgrösse p (in Bytes), unter Annahme von 40 Bytes Headerlänge (entspricht RTP/UDP/IP), einer Datenrate von 8000 Bytes/s und einer direkten Verbindung zwischen Sender und Empfänger.

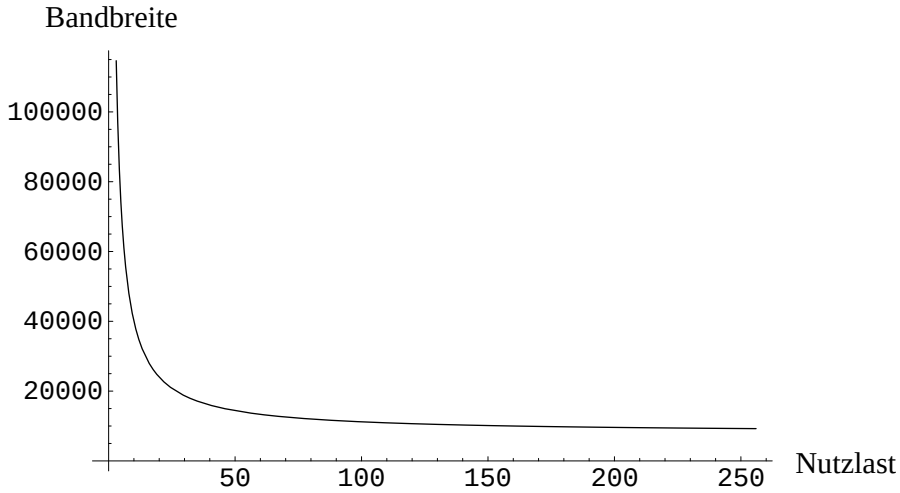


Abbildung 2.2: Benötigte Bandbreite pro Nutzlastgrösse

Verzögerung ist zentral für die subjektive Qualitätswahrnehmung. Die Verständlichkeit bleibt zwar auch bei hoher Verzögerung erhalten, der Charakter der Konversation leidet aber stark darunter. Bei einem normalen Gespräch kann eine Partei die andere unterbrechen, oder auch gleichzeitig zur anderen sprechen, beispielsweise für kurze Einwürfe. Wenn nun die Verzögerung zu gross wird, degeneriert der Charakter der Konversation zunehmend zu dem eines Funkgesprächs, bei dem jeweils eine Partei spricht und dann auf die Antwort der anderen Partei wartet. Messungen haben ergeben, dass Verzögerungen kleiner als 100 ms nicht wahrgenommen werden. Zwischen 100 ms und 300 ms bemerkt der Benutzer eine leichte Verzögerung in der Antwort des Gesprächspartners. Verzögerungen über 300 ms werden als störend empfunden.

2.2.3 Verzögerungsschwankungen

Wenn die Pakete einer Audioverbindung regelmässig eintreffen, können sie jeweils direkt im Anschluss an ein früheres Paket abgespielt werden. Die Verzögerung bei der Übertragung ist aber nicht konstant — Pakete kommen zu früh oder zu spät an — was Lücken im abgespielten Signal verursachen kann. Die Differenz zwischen den Verzögerungen einzelner Pakete wird *Jitter* (Verzögerungsschwankung) genannt. Um Lücken im Signal zu vermeiden, müssen die erhaltenen Daten zwischengespeichert werden, um bei verspäteten Paketen trotzdem Daten abspielen zu können. Die Grösse dieses Zwischenspeichers (Synchronisationspuffer oder Jitterbuffer genannt) kann eines oder mehrere Kodierungsframes umfassen. Solche Puffer sind häufig auch bei RTP-Mixern und -Transcodern nötig. Durch Jitter wird also die Verzögerung noch verstärkt, da die Daten, wenn sie beim Endpunkt angekommen sind, trotzdem noch nicht sofort abgespielt werden können.

2.2.4 Echo

Echo wird durch eine schlechte Trennung zwischen Lautsprecher und Mikrofon, wie zum Beispiel bei einem Telefonhörer, an den Endpunkten einer Verbindung verursacht. Ein Teil des Signals wird so zum Sender zurückgeschickt. Das muss nicht unbedingt zu einer Verschlechterung der Qualität führen; ein Echo mit einer Verzögerung von 16–20 ms ist sogar wünschenswert, weil der Sprecher dadurch den Eindruck einer “lebendigen” Leitung erhält. Bei Verzögerungen grösser als 32 ms wirkt Echo hingegend störend.
Um das Echo zu eliminieren oder zumindest zu dämpfen, können sogenannte Echokompensatoren zwischen den Endpunkten benutzt werden. Normalerweise wird jeweils nahe an einem Endpunkt ein solches Gerät (oder Programm) installiert. Wenn ein Echokompensator aktiviert wird, analysiert er die Verbindung und beginnt damit, das Echo aus dem Signal herauszufiltern. Die Geschwindigkeit,

in der er sich auf die Eigenschaften einer Verbindung einstellen kann, wird *Konvergenzzeit* genannt, die stärke der Dämpfung heisst *Kompensationstiefe* (gemessen in dB). Ein wichtiger Aspekt der Echodämpfung ist die *Double Talk* genannte Situation, wenn beide Teilnehmer gleichzeitig sprechen. Die Fähigkeit, trotz Double Talk die Echodämpfung aufrecht erhalten zu können, wird *Double-Talk-Robustheit* genannt.

2.3 Qualitätsmessung

2.3.1 Metriken für IP-Netzwerke

Viele der Parameter von Verbindungen auf IP-Netzwerken können berechnet oder zumindest geschätzt werden. Obwohl diese Werte nicht direkt die Qualität der übertragenen Sprache messen, ist diese doch von vielen von ihnen abhängig. Einige dieser Parameter werden hier in grösserem Detail besprochen.

2.3.1.1 Bandbreite

Einer der wichtigsten Faktoren für VoIP ist die verfügbare Bandbreite, da sie eine obere Schranke bezüglich der Sprachqualität setzt und Verzögerung wie auch Paketverlustrate stark mit beeinflusst. In einigen Fällen wird diese bekannt sein, in vielen anderen kann sie aber nur geschätzt werden. Nebst der Kapazität der einzelnen Links spielt dabei ihre aktuelle Auslastung eine zentrale Rolle. In IntServ- oder DiffServ-Umgebungen kommt noch die Frage der Prioritäten hinzu.
Eine einfache Methode, die verfügbare Bandbreite zwischen zwei Systemen A und B zu ermitteln ist, von A aus eine kurze Zeit lang soviele UDP-Pakete wie möglich an System B zu senden (man *flutet* das Netzwerk), das diese registriert und die erreichte Bandbreite zurückmeldet. Die so gemessenen Werte sind sehr genau. Allerdings ist diese Methode ausserhalb von Testnetzen nicht tauglich, da sie zuviel Bandbreite verschwendet. Immer, wenn eine Applikation die Bandbreite messen möchte, käme der Datenverkehr praktisch zum Erliegen.

Vernünftiger verhält sich folgende Methode: Anstatt zu Beginn einer Übertragung die Bandbreite mit obiger Methode auszumessen, wird eine erste Schätzung angenommen. Diese kann durch die (bekannte) Bandbreite des Netzanschlusses gegeben sein, oder durch die maximal benötigte Bandbreite der verwendeten Kodierung, falls diese kleiner als der erste Wert ist. Ist die Schätzung zu tief, wird nichts weiter geschehen. Ist sie zu hoch, macht sich das entweder durch grosse Paketverluste und Verzögerung bemerkbar oder dadurch, dass sich die zu sendenen Daten im Endsystem stauen. Die zu Anfang getroffene Annahme muss nun

solange nach unten korrigiert werden, bis sich die Situation normalisiert. Somit erhält man eine Schätzung der momentan verfügbaren Bandbreite, ohne das Netz übermässig zu belasten. Nach dem Vorbild von TCP könnte nun ständig versucht werden, die Bandbreite wieder zu erhöhen, um so auf eventuell zurückgehende Auslastung zu reagieren. Die dadurch aber noch gesteigerten Paketverluste haben für VoIP aber sehr unerwünschte Effekte — die vorgestellte Methode verursacht bei schlechter Schätzung, schon zu viele Verluste —, weshalb meist die zu tiefe, aber verlässliche Bandbreite akzeptiert wird.
Voraussetzung für diese Art der Bandbreitenschätzung ist eine Kodierung mit flexibler Bitrate, oder zumindest eine Reihe von Kodierungen mit fixen, aber unterschiedlichen Bitraten.

Offensichtlich ist es sehr schwierig, ohne Qualitätseinbussen die zur Verfügung stehende Bandbreite zu messen. Wo immer möglich sollte daher die verwendete Bandbreite zentral kontrolliert werden, um klare Verhältnisse zu schaffen. In dedizierten VoIP-Netzen ist dies gut realisierbar; In Netzen mit verschiedenen Anwendung muss dafür auf QoS-Mechanismen wie IntServ oder DiffServ zurückgegriffen werden.

2.3.1.2 Paketverlust

Das Messen von Paketverlusten hat zwei Aspekte: einerseits muss das Verhältnis zwischen gesendeten und empfangenen Paketen ermittelt werden, andererseits braucht es Kriterien, die darüber entscheiden, wann ein Paket als verloren bezeichnet werden kann.
Die erste Aufgabe ist einfach. Der Sender stattet die von ihm ausgehenden Pakete mit einer Sequenznummer aus, womit der Empfänger fehlende Pakete erkennen kann, indem er nach Lücken in den beobachteten Sequenznummern Ausschau hält. Ist s_{max} die höchste und s_{min} die kleinste in einem Zeitintervall beobachtete Sequenznummer, und ist n die Anzahl zwischen s_{min} und s_{max} liegender, beobachteter Sequenznummern, berechnet sich die Verlustrate r durch

$$r = \frac{n}{s_{max} - s_{min}} \, .$$

Ist die Verlustrate seit Beginn der Übertragung von Interesse, kann s_{min} gleich der ersten Sequenznummer gesetzt und dieselbe Formel verwendet werden.[1]

Schwieriger als das Berechnen der Verlustrate ist die Entscheidung darüber, welche Pakete als verloren gezählt werden sollen und welche nicht. Diese Fra-

[1] Eine wichtige Tatsache ist die, dass die erste Sequenznummer üblicherweise *nicht* gleich 0 ist, sondern im Zufallsverfahren gewählt wird. Dies gilt auch für RTP.

ge kann nicht allgemein beantwortet werden, da sie von der Art der Anwendung abhängt. Bei nicht zeitkritischen Anwendungen kann lange auf ein verspätetes Paket gewartet werden und Reihenfolgevertauschungen haben geringen Einfluss. Ein Paket wird hier erst als verloren betrachtet, nachdem ein gewisses Zeitintervall überschritten ist. VoIP-Anwendungen hingegen sind meist sehr empfindlich auf Verzögerung und Jitter, weshalb verspätete Pakete meist als verloren gewertet werden, selbst wenn sie ihr Ziel erreichen. Ähnlich verhält es sich mit vertauschten Paketen: Reihenfolgevertauschungen können höchstens innerhalb des Synchronisationspuffers korrigiert werden. Ist dieser nicht gross genug, wird auch ein solches Paket als verloren betrachtet.
[RFC 1889] schreibt sogar für kurz- und langfristige Statistiken verschiedene Kriterien vor. In der kurzen Frist gelten alle nicht benutzbaren Pakete als verloren, langfristig werden sie aber als erhalten gezählt.

2.3.1.3 Verzögerung

Die Verzögerung (englisch: Delay) ist der am schwierigsten zu messende Wert. Eine Näherung erhält man dadurch, dass man die Roundtrip-Zeit durch zwei teilt. Diese wird folgendermassen berechnet: Der Sender schickt ein Paket an den Empfänger und merkt sich die Zeit, an dem es abgeschickt wurde. Der Empfänger schickt das Paket umgehend zurück und vermerkt darin die Zeit, die zwischen Empfang und Senden der Antwort verstrichen ist. Wenn die Antwort den ursprünglichen Sender erreicht, kann dieser durch die Differenz zwischen Empfang der Antwort und der gespeichert Zeit, abzüglich der im Paket vermerkten Dauer, die Roundtrip-Zeit berechnen. Dieses Verfahren wird in RTP verwendet (siehe Seite 25).
Leider erhält man so nur eine grobe Schätzung der Verzögerung in einer der beiden Richtungen, da diese stark asymmetrisch sein können. Die Auslastung der Verbindung auf dem Hinweg kann sehr stark von derjenigen auf dem Rückweg abweichen.

Sind die Systemuhren beider Systeme genau synchronisiert, kann natürlich durch vermerken von Timestamps im Paket ein viel genaueres Resultat erzielt werden. Indem die Uhren der Endpunkte durch das Network Time Protocol (NTP) an eine Referenzuhr angeglichen werden, können die Differenzen im Bereich einiger Millisekunden gehalten werden. Wo NTP nicht zur Verfügung steht, weichen die Systemuhren mit zunehmender Laufzeit voneinander ab, wodurch auch die Verzögerungsberechnung zunehmend ungenauer wird.

2.3.1.4 Verzögerungsschwankungen

Bedeutend einfacher als die Verzögerung sind die Schwankungen zwischen den Verzögerungen einzelner Pakete, der Jitter, zu messen. Für zwei aufeinanderfolgende Pakete p und q mit Sendezeiten p_s und q_s und Empfangszeiten p_e und q_e ist der Jitter gegeben durch $|(q_e - p_e) - (p_s - q_s)|$.
Darüber, wie der Jitter für mehrere Pakete definiert ist, gibt es widersprüchliche Informationen. In [RFC 1889] ist er definiert als die Varianz des Jitters zwischen je zwei aufeinanderfolgenden Paketen. RTP benutzt dazu einen laufend berechneten Schätzwert, der auf Seite 19 erläutert wird. [RFC 2598] hingegen verlangt den Median derselben Werte.

2.3.2 PSQM

Ein schwierig zu beurteilender Faktor bei der Qualitätsmessung einer Sprachübertragung ist der Eindruck auf die subjektive, menschliche Wahrnehmung. Perceptual Speech Quality Measurement (PSQM[2]) beschreibt ein Verfahren, diese zu modellieren.
Darin vergleicht man ein mit VoIP übertragenes Signal mit seinem Original, indem die Frequenzspektren und Lautstärken beider Signale durch ein mathematisches Model der menschlichen Wahrnehmung verarbeitet und verglichen werden. Das Ergebnis ist eine positive reelle Zahl. Höhere Werte bedeuten dabei eine schlechtere Bewertung.
Bislang ist PSQM hauptsächlich in Form von Hardware-Implementierung erhältlich.

2.4 Lösungsansätze

2.4.1 Kodierungsadaption

In vielen Fällen kann die Qualität der Übertragung verbessert werden, indem die Bitrate der gesendeten Daten verkleinert wird. Damit sinkt entweder die Paketrate oder die Grösse der einzelnen Pakete, wodurch das Netzwerk entlastet und dadurch zuverlässiger wird. Da kleinere Pakete schneller weitergesendet werden können, werden auch die Router-Queues schneller geleert und überlaufen deshalb weniger häufig. Eine kleinere Paketrate wirkt sich auf der anderen Seite dadurch aus, dass sich die Queues langsamer füllen, was zum selben Effekt führt.

[2] [P.861]

Ob und in welcher Art sich die Bitrate senken lässt, hängt vom verwendeten RTP-Profil und der darin definierten Audiokodierungen ab. Als Beispiel soll hier das Profil aus [RFC 1890] dienen. Angenommen, es werden Daten im Format PCM μ-Law mit 8000Hz Samplerate (entspricht 64 kbit/s) gesendet, und die erzielte Qualität sei ungenügend, dann kann beispielsweise auf eine ADPCM-Kodierung (G.726) mit 32 kbit/s Bitrate umgeschaltet und damit die Netzwerkbelastung um nahezu die Hälfte verringert werden. Der Empfänger erkennt die neue Kodierung am Payload-Type-Feld im RTP-Header und kann die Daten entsprechend dekodieren. Alternativ dazu kann das Kodierungsschema beibehalten und dessen Bitrate gesenkt werden, natürlich unter der Voraussetzung, dass das Kodierungsschema variable Bitraten unterstützt. Beispiele dafür sind die samplebasierten Kodierungsschemen und einige framebasierte, wie VDVI (Variable-rate-DVI4, eine Abwandlung von IMA ADPCM).

Es kann von Vorteil sein, solche variablen Schemen zu verwenden. Dies einerseits deshalb, weil dadurch ein konsistenteres Hörempfinden ensteht (die Signalverzerrung ändert nur ihre Stärke, nicht ihren Charakter, was weniger überraschend wirkt als eine plötzliche Änderung des Klangcharakters), vor allem aber wegen der verminderten Komplexität (und damit der Fehleranfälligkeit) der Implementierung.

2.4.2 Redundante Kodierung

Mit Kodierungsadaption wird ein Audiostrom an die Gegebenheiten der Verbindung angepasst, paketverluste führen jedoch immer noch zu Lücken im resultierenden Audiosignal. Man kann dieses Problem angehen, indem man in jedem Paket Redundante Kodierung geht dieses Problem an, indem in jedem Paket einen Block mit stärker komprimierten Daten aus dem vorherigen Paket vor den aktuellen Audiodaten einfügt. Geht nun ein Paket verloren, kann der Empfänger die verlorenen Daten dem nächsten Paket entnehmen. Solange nur einzelne Pakete verlorengehen, entsteht also nur eine kurzfristige Qualitätsschwankung.
Eine Voraussetzung für diese Ansatz ist, dass der Empfänger ankommende Daten vor dem Abspielen zumindest so lange zwischenspeichert, bis er die Daten verlorener Pakete durch die redundanten Daten des nächsten ersetzen kann. Diese Funktionalität kann auch den häufig eingesetzten Synchronisationspuffern hinzugefügt werden. Abb. 2.3 zeigt eine schematische Darstellung des Prinzips.

2.4.3 Headerkomprimierung

Je kleiner die verfügbare Bandbreite, desto grösser wird der Einfluss der Headerlänge von Paketen auf die Verzögerung. Wenn für eine RTP-Anwendung UDP

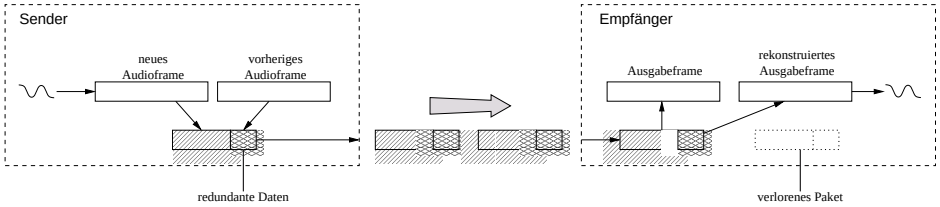


Abbildung 2.3: Redundante Kodierung

als darunter liegendes Protokoll verwendet wird, ergibt sich eine Headerlänge von 40 Bytes (IP-Header: 20 Bytes, UDP-Header: 8 Bytes, RTP-Header: 12 Bytes), was auf einem 10 Mbit/s Ethernet zwar nur 32 μsec zusätzliche Verzögerung ergibt; aber im Falle eines mit 28.8 kbit/s sendenden Modems ergeben sich schon 11 ms Verzögerung, was sich zusammen mit den anderen Faktoren schnell bemerkbar macht[3]. Es bietet sich also an, nach einem Weg zu suchen, die Länge des Headers zu beschränken.

In [RFC 2508] ist eine Methode definiert, die dieses Ziel in vielen Fällen erreicht. Die Beobachtung, auf der die Idee basiert, ist die, dass sich die meisten Felder in den Headern während der Übertragung nicht oder nur wenig verändern und somit redundante Information tragen. Wenn beide Seiten einer (physikalischen) Verbindung darüber informiert sind, kann ein verkürzter Header, in dem die redundanten Felder fehlen, alle benötigten Informationen übertragen. Die sendende Seite wird *Kompressor*, die empfangende *Dekompressor* genannt. Eine weitere Voraussetzung ist der Einsatz eines Link-Layer-Protokolls, das die durch Headerkomprimierung entstehende, zusätzliche Kommunikation zwischen Kompressor und Dekompressor unterstützt. Abb. 2.4 zeigt die Felder in den drei Headern, die sich nicht oder nur wenig ändern[4]. Im einfachsten Fall verändert sich der Wert eines Feldes nach dem ersten Paket gar nicht und kann daher in den folgenden Paketen übernommen werden. Andere ändern sich in vorhersehbarer Art. Sequenznummern werden beispielsweise immer um eins inkrementiert. Nachdem das erste Paket übertragen wurde, kann nun der Verbindung eine Identifikation zugeordnet werden, eine sogenannte *Context session ID* (CID). Mit den immer noch gesendeten Prüfsummen kann ermittelt werden, ob der rekonstruierte Header dem ursprünglichen entspricht. Ist dies nicht der Fall, oder sind Pakete verlorengegangen, muss der Dekompressor dem Kompressor über das Link-Layer-Protokoll mitteilen, welche Felder neu übertragen werden müssen. Im schlimmsten Fall muss ein Paket mit vollem Header geschickt werden.

[3]Bei diesen Berechnungen wurde der Link-Layer-Header nicht berücksichtigt

[4]In dieser Abbildung ist das TOS-Feld als konstant bezeichnet, was aber in DiffServ-Netzen nicht der Fall ist

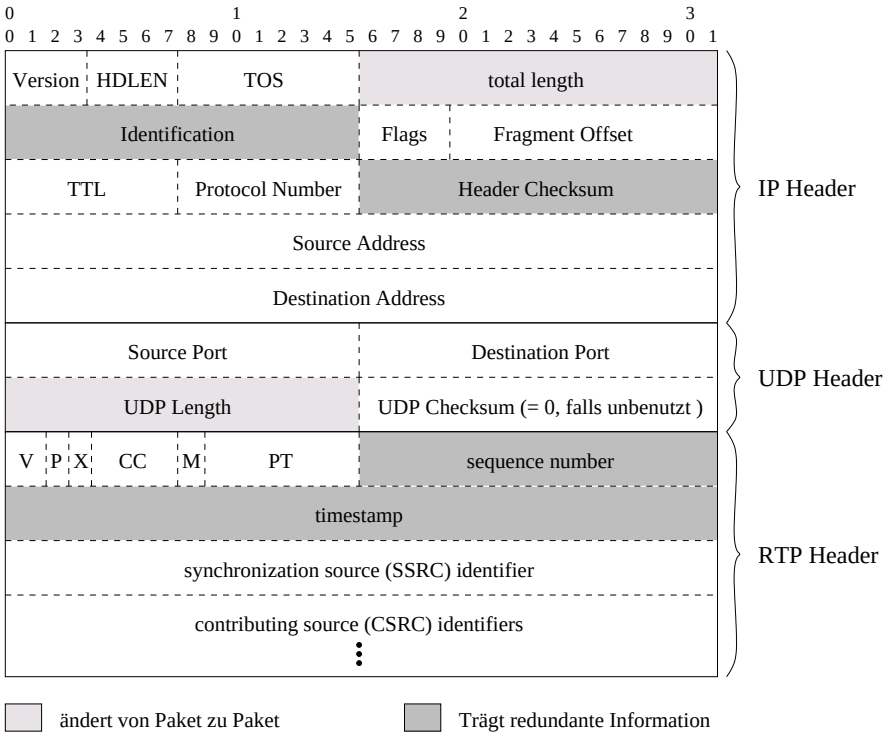


Abbildung 2.4: Redundante Felder in IP/UDP/RTP-Headern (Quelle: [DousB])

Je nachdem, welche Header komprimiert werden, können Kompressor und Dekompressor näher oder weiter voneinander entfernt sein. Bei voller Kompression dürfen beide nur durch einen Link mit entsprechendem Link-Layer-Protokoll verbunden sein. Ist der IP-Header unkomprimiert, können die Pakete zwischen zwei beliebigen Systemen in einem IP-Netz kursieren, die UDP/IP-Stacks beider Systeme müssen aber Unterstützung für komprimierte UDP-Header bieten. In der einfachsten Variante wird nur der RTP-Header komprimiert, in welchem Fall einzig die RTP-Anwendung entsprechend konfigurierbar sein muss. Allerdings ist die Verzögerungs-Reduktion dann nur minimal. Ausserdem können bei Konferenzen oder bei komplexen Profilen nicht soviele Felder vernachlässigt werden, wie bei einer einfachen Punkt-zu-Punkt-Verbindung.

Header-Compression bringt offensichtlich nur bei niedriger Bandbreite echte Vorteile. Die hauptsächlichen Anwendungsgebiete sind denn auch die “letzte Meile”, die häufig aus langsamen seriellen Verbindungen besteht, und Funknetze. Im Backbone ist der Aufwand der Komprimierung und Dekomprimierung zu gross, um den kleinen Vorteil einer teilweisen Kompression zu rechtfertigen.

2.4.4 RSVP

Es ist sehr schwierig — in vielen Fällen sogar unmöglich — vorauszusagen, welche Dienstqualität ein Paket in einem dynamischen IP-Netz erhalten wird. Wenn ein Anruf aufgebaut wird, sollte dem MGC aber doch eine gute Schätzung darüber vorliegen, damit überzählige Verbindungen abgewiesen werden können.
Dieses Problem wurde von der IETF unter anderem mit der Integrated-Services-Architektur (kurz: IntServ) angegangen, deren bekannteste Realisierung das Re-Source reserVation Protocol (RSVP[5]) ist. Die Grundidee dieses Protokolls ist es, den Pfad, den die Pakete zwischen den zwei beteiligten Endpunkten zurücklegen, festzusetzen und entlang diesem Pfad Bandbreite für die Verbindung zu reservieren. Indem der Datenpfad festgelegt wird, können viele Unsicherheiten ausgeschaltet werden, die eine Voraussage der Dienstqualität erschweren. Durch die Reservationen werden ausserdem die Pakete einer Anwendung vor den Auswirkungen von unkontrolliertem Datenverkehr geschützt. Die Dienstqualität wird im Router dadurch verwirklicht, dass zu verschiedenen Datenflüssen gehörende Pakete verschieden behandelt werden. Dies geschieht meist im *Packet Scheduler*, der über die Reihenfolge entscheidet, in der die Pakete aus der router-internen Queue weitergesendet werden.
RSVP bietet volle Unterstützung für Multicasting, was natürlich einen Einfluss auf die Komplexität des Protokolls hat. Eine der Auswirkungen ist, dass die Reservationen vom Empfänger ausgehen, eine Eigenschaft, die es auch erlaubt, jedem Teilnehmer einer Multicast-Sitzung eine unterschiedliche Bandbreite zuordnen zu können. Reservationen können allerdings nur vom Empfänger ausgehen.
Da die Router entlang dem Datenpfad für jede Reservation Tabelleneinträge verwalten, müssen sie auch sicherstellen können, dass "vergessene" Reservationen wieder rückgängig gemacht werden, da sonst die Bandbreite für neue Anfragen nicht verfügbar ist. Dies wird dadurch gelöst, dass Einträge nach einer gewissen Frist verfallen, wenn sie nicht vorher erneuert worden sind. Man spricht von einem *Soft State* der Reservationen. Indem bei der Erneuerung andere Werte angegeben werden, können Reservationen auch dynamisch verändert werden.

2.4.4.1 Reservationsarten

Reservationen werden durch eine Flussbeschreibung (*Flow Descriptor*) charakterisiert, der wiederum in Flussspezifikation (*Flow Specification*) und Filterspezifikation (*Filter Specification*) zerfällt. Die Flussspezifikation definiert dabei die

[5] [RFC 2205]

Art, wie der Packet Scheduler die zum Fluss gehörenden Pakete behandeln soll, während die Filterspezifikation bestimmt, welche Pakete überhaupt zum Fluss gehören sollen.

Es gibt mehrere Arten von Filterspezifikationen. Die einfachste unter ihnen definiert einen Fluss von einem Endpunkt zum anderen und eignet sich daher für normale Telefongespräche. Um die Anzahl der Filter in Multicast-Sitzungen zu verringern, existieren zwei weitere Spezifikationsarten, die Pakete von mehreren Systemen oder von ganzen System-Gruppen als zum Fluss gehörig einstufen können.

2.4.4.2 Reservationsaufbau

Der Aufbau einer RSVP-Reservation soll hier anhand eines Beispiels erklärt werden, das in Abb. 2.5 illustriert ist. Ausgangslage sind zwei Endsysteme, der Sender A und der Empfänger B, die über zwei Router R1 und R2 verbunden sind. A beginnt den Aufbau, indem er eine PATH-Nachricht an B schickt, die Informationen über die Pakete enthält, die er zu senden plant. Wenn das Paket einen der Router erreicht, speichert dieser einen PATH STATE, der die Adressen der benachbarten Hops enthält, damit die nachfolgenden Pakete über denselben Pfad geschickt werden können. Schliesslich erreicht das Paket den Endpunkt B, welcher aufgrund der im Paket enthaltenen Informationen eine RESV-Nachricht mit einer Liste von Reservations-Anfragen generiert und sie zurückschickt. Die Router auf dem Pfad überprüfen darauf, ob noch genug Bandbreite vorhanden ist, um die Reservation tätigen zu können. Ausserdem ermitteln sie die Berechtigung von B, überhaupt Reservationen zu tätigen. Wenn beide Tests positiv verlaufen, wird eine Bestätigung (RESV CONF), andernfalls eine Fehlermeldung (RESVERR) zurückgesendet. Die Reservationen können durch eine PATH TEARDOWN-Nachricht von

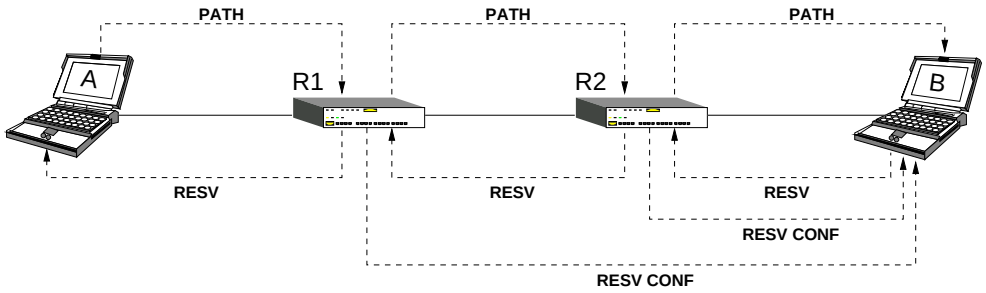


Abbildung 2.5: Aufbau einer RSVP-Reservation

A und eine RESV TEARDOWN-Nachricht von B wieder rückgängig gemacht werden. Wenn ein Router eine solche Nachricht erhält, entfernt er die jeweiligen Reservationen aus seinen Tabellen.

2.4.4.3 Nachteile

Obwohl sich RSVP für den Einsatz in kleineren Netzen gut eignet, weist es in grossen oder unkontrollierten Netzen Nachteile auf. Zum einen kann es vorkommen, dass nicht alle Router auf dem Pfad zwischen den Endsystemen RSVP-fähig sind. Das verhindert zwar nicht den Einsatz von RSVP, da PATH- und RESV-Nachrichten auch von nicht RSVP-fähigen Routern weitergeleitet werden, schränkt aber die Verlässlichkeit einer Reservation stark ein, da ein Paket von einem solchen Router so umgeleitet werden kann, dass es mehrere RSVP-Router überspringt.
Weiter muss alle Software, die RSVP nutzen soll, RSVP-fähig sein. Da PATH- und RESV-Nachrichten zwischen Quelle und Ziel des Datenflusses hin- und hergeschickt werden müssen, kann keine externe Instanz diese Aufgabe für sie übernehmen.
Das Hauptproblem ist jedoch das der Skalierbarkeit: Der allergrösste Teil der VoIP-Datenflüsse wickelt sich zwischen zwei Endpunkten ab, was in den Routern einen Tabelleneintrag für jeden durch sie verlaufenden Fluss erfordert. Zusammen mit der Anzahl Teilnehmer eines normalen Telefonnetzes ergeben sich für die zentralen Router extrem grosse Tabellen, die sie verwalten müssen. Der benötigte Speicher und vor allem die Verzögerung, die durch das Suchen nach passenden Tabelleneinträgen ensteht, kann unakzeptable Ausmasse annehmen. Für dieses Problem muss eine Lösung gefunden werden, wenn RSVP in grossen VoIP-Netzen zum Einsatz kommen soll.

2.4.5 Differentiated Services

Ein alternativer Ansatz, der die Skalierungsprobleme von Integrated Services beziehungsweise RSVP lösen könnte, ist Differentiated Services (kurz DiffServ genannt, siehe [RFC 2475]). Er basiert darauf, dass die vielen Mikroflüsse, die bei RSVP selbst in den Backbone-Routern alle einzeln vermerkt sind, an den Netzwerkgrenzen zu einigen wenigen Klassen aggregiert und innerhalb des Netzwerks nur noch durch ihre zugewiesene Klasse unterschieden werden. Die einzelnen Router behandeln dann die Pakete nach einem ihrer Klasse zugeordeten Muster, dem Per-Hop-Behaviour (PHB). Bei DiffServ steht also nicht der gesamte Pfad eines Flusses im Mittelpunkt, sondern die Art, wie die Pakete auf den einzelnen

Hops behandelt werden. Dadurch sinkt die Komplexität, die ein einzelner Router bewältigen muss, auf ein niedriges und nicht von der Anzahl Mikroflüsse abhängiges Niveau.

2.4.5.1 Terminologie

Die folgenden Begriffe haben spezielle Bedeutung für DiffServ:

Behaviour Aggregate (BA) bezeichnet eine Menge von Paketen mit demselben DS-Codepunkt, die in eine gewisse Richtung fliessen.

Classifier Eine Einheit, die Pakete aufgrund ihres Headers und vordefinierter Regeln auswählt. Ein BA-Classifier wählt Pakete nur aufgrund ihres DS-Feldes aus; die allgemeine Form wird MF-Classifier genannt (MF steht für Multi Field).

Grenzknoten Ein DS-Knoten, der eine DS-Domäne mit einer anderen oder mit einer nicht DS-fähigen Domäne verbindet.

DS-Codepunkt Ein spezifischer Wert des DSCP-Teils im DS-Feld.

DS-Domäne Zusammenhängendes Netz von Knoten mit einer gemeinsamen DiffServ-Konfiguration.

DS-Knoten Ein DiffServ-fähiger Netzwerkknoten. Spezielle Knoten sind der Ingress-Knoten, der sich am "Eingang" einer DS-Domäne befindet und der am "Ausgang" befindliche Egress-Knoten. Die dazwischen liegenden Knoten werden innere Knoten (Interior Nodes) genannt.

DS-Feld Das TOS-Feld in IPv4- oder das Traffic-Class-Feld in IPv6-Headern, interpretiert nach [RFC 2474].

Dropping "Wegwerfen" von Paketen, im Zusammenhang mit Verkehrsprofilen auch Policing genannt.

Marking Das Setzen des DS-Codepunkts eines Pakets nach definierten Regeln. Pre-marking meint das markieren, bevor ein Paket in eine DS-Domäne eintritt, Re-marking das ummarkieren eines bereits gesetzten Codepunkts.

Metering Messen der momentanen Eigenschaften des von einem Classifier ausgewählten Verkehrsstroms. Der resultierende Messwert kann von Markern, Shapern oder Droppern verwendet werden.

Mikrofluss Der Datenfluss zwischen zwei Anwendungen.

Per-Hop-Behaviour (PHB) Das Verhalten eines DS-Knotens gegenüber einem Behaviour Aggregate. Lassen sich mehrere PHBs konzeptionell nicht trennen (z.B. Dropping-Prioritäten), spricht man von einer PHB-Gruppe.

Service Das gesamte Verhalten einer DS-Domäne gegenüber einem Teil des Verkehrs eines Kunden.

Service Level Agreement (SLA) Vertrag, der den Service definiert, den ein Kunde erwarten kann. Er kann auch ein TCA enthalten.

Shaping Das Begrenzen eines Datenstroms auf eine gewisse Paket- oder Bitrate durch verzögertes Senden der Pakete.

Traffic conditioning Erzwingen gewisser Regeln bezüglich eines Datenstroms durch Metering, Marking, Shaping und/oder Policing. Die Regeln werden in einem Traffic Conditioning Agreement (TCA) festgelegt, das Teil eines Service Level Agreements sein kann.

Zusätzlich zu diesen Begriffen wird in den weiteren Kapiteln der Begriff der *Serviceklasse* benutzt, der den *Service*-Begriff etwas präzisiert. Eine Serviceklasse bezeichnet das gesamte Verhalten einer DS-Domäne gegenüber dem zu einem bestimmten Behaviour Aggregate gehörenden Teil des Verkehrs eines Kunden.

2.4.5.2 Struktur eines DiffServ-Netzes

Das DiffServ-Modell unterteilt die Welt in DS-Domänen und Domänen ohne DiffServ-Unterstützung, die rechtlich oder konzeptuell unabhängigen Parteien gehören oder von ihnen verwaltet werden. Datenströme treten über Ingress-Knoten in eine DS-Domäne ein, wo sie in Behaviour Aggregates aufgeteilt werden. Dies geschieht durch einen MF-Classifier und einen Marker, der den Codepunkt der Pakete entsprechend kennzeichnet. Falls ein Datenstrom die im SLA festgelegten Parameter übersteigt, muss er gemäss dem TCA geglättet werden, was üblicherweise mit einem Policer oder Shaper geschieht.
Innerhalb der Domäne werden die Behaviour Aggregates von den Interior-Knoten mit einem BA-Classifier identifiziert und gemäss dem vordefinierten Per-Hop-Behaviour weitergeleitet, bis sie einen Egress-Knoten erreichen. Dieser kann die Ströme normalerweise an die nächste Domäne weiterleiten, ohne sie zu verändern. Wenn das TCA dies aber erfordert, muss er sie erst gemäss dem TCA mit der nachfolgenden Domäne glätten, bevor er sie weiterschicken kann. Abb. 2.6 stellt diesen Ablauf schematisch dar.

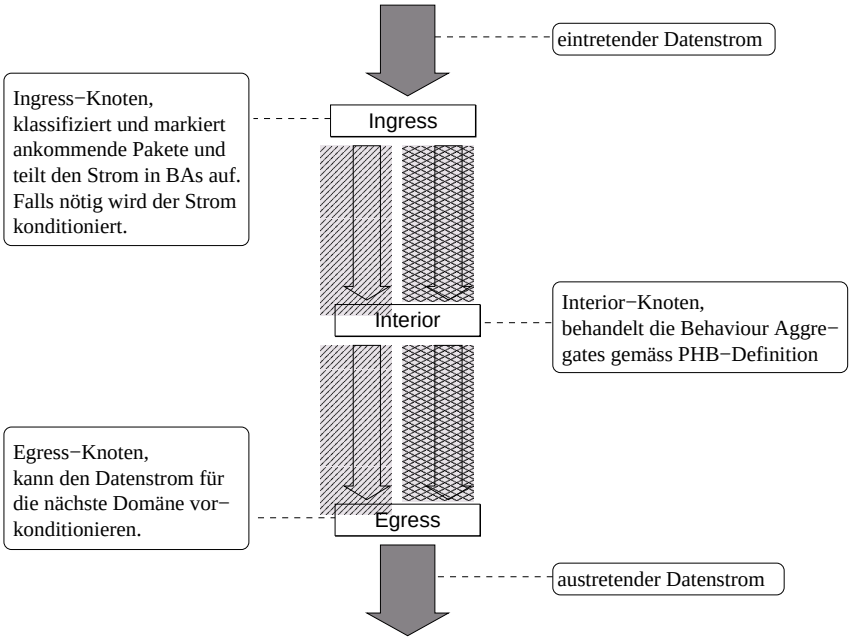


Abbildung 2.6: Das DiffServ-Modell

2.4.5.3 Mechanismen und PHBs

Es gibt einen wichtigen Unterschied zwischen dem Begriff des Per-Hop-Behaviour und dem des Mechanismus. Ein PHB beschreibt nur den von aussen sichtbaren Effekt auf die Behaviour Aggregates, behandelt einen DS-Knoten also als Blackbox. Mechanismen hingegen werden innerhalb des Knotens eingesetzt, um das geforderte PHB zu erreichen und sind in diesem Sinne austauschbar. Solange der gewünschte Effekt erzielt wird, ist jede Kombination von Mechanismen erlaubt.
Innerhalb einer Domäne ist eine Abbildung der möglichen Codepunkte auf eine Menge von PHBs definiert; für diese Abbildung und die verwendeten PHBs existieren Kodierungsregeln ([RFC 2474]) und eine Reihe von Vorschlägen, die Wahl ist aber grundsätzlich frei. Im folgenden werden einige dieser Vorschläge vorgestellt.

Per-Hop-Behaviours

Um dem Verhalten einer Standleitung möglichst nahe zu kommen, wurde *Expedited Forwarding*[6], auch Premium Service genannt, erfunden. Das Ziel ist, einen Datenstrom, der sich innerhalb einer festgelegten Bandbreite bewegt, mit möglichst

[6][RFC 2598]

kleiner Verzögerung und ohne Paketverluste oder Reihenfolgevertauschungen weiterzuleiten. Wenn die Queues eines Knotens gleichzeitig Pakete eines EF-Stromes und solche eines anderen Stromes enthalten, müssen die EF-Pakete immer zuerst gesendet werden. Erst wenn die EF-Queue leer ist, dürfen andere Pakete weitergeleitet werden. An den Domänengrenzen kann es vorkommen, dass der EF-Strom die vordefinierte Bandbreite überschreitet, worauf er geglättet werden muss. Das kann durch zwei Methoden erreicht werden: die bevorzugte ist es, einen Shaper einzusetzen. Überzählige Pakete werden nicht direkt weitergeleitet, sondern bleiben solange in der Queue, bis die resultierende Bitrate der gesendeten Pakete die gewünschte Bandbreite nicht übersteigt. Diese Vorgehensweise funktioniert gut, solange die Bandbreitenüberschreitungen nur von kurzer Dauer sind. Ziehen sie sich aber über so lange Zeit hinweg, dass die Queue gefüllt wird, müssen zusätzlich ankommende Pakete gedroppt werden.
Alternativ zu einem Shaper kann auch ein Policer eingesetzt werden. Dieser misst die Bandbreite des Stromes beim Eintritt in den Knoten und verwirft überzählige Pakete, ohne sie zuvor in der Queue abgelegt zu haben. Diese Methode findet vor allem in Fällen Verwendung, wo das Traffic Conditioning Agreement vorschreibt, dass der beim Ingress-Knoten ankommende Strom bereits dem SLA entspricht. In einem solchen Fall ist es akzeptabel, überzählige Pakete einfach zu verwerfen.

Eine weitere beliebte Art von PHB ist *Assured Forwarding*[7], das auch unter dem Namen Assured Service bekannt ist. Die Überlegung hinter AF ist, dass eine Organisation, die zwei geographisch getrennte lokale Netze verbindet, die Gewissheit haben will, dass die Daten zwischen ihnen mit hoher Wahrscheinlichkeit korrekt übertragen werden, solange die benötigte Bandbreite unter einem gewissen Wert bleibt. Zusätzlich sollen aber auch Überschreitungen dieses Wertes möglich sein, wobei aber die Wahrscheinlichkeit einer korrekten Übertragung sinken darf. Assured Forwarding definiert nun vier PHB-Gruppen, die AF-Klassen 1–4, die weiter in drei Dropping-Prioritäten unterteilt sind. Pakete einer AF-Klasse werden vom Ingress-Knoten mit einer der drei Dropping-Prioritäten markiert, je nachdem, ob und wie stark die vorgeschriebene Bandbreite überschritten wird. Abb. 2.7 veranschaulicht dies.

Solange kein Stau entsteht, leiten Interior-Knoten auch Pakete mit hoher Priorität einfach weiter. In Stausituationen hingegen werden zuerst Pakete mit hoher Priorität, danach solche mit mittlerer Priorität verworfen. Gute Planung vorausgesetzt, sollten auch bei starkem Stau keine Pakete mit tiefer Priorität verworfen werden müssen. Dieses Verhalten kann durch mehrere Mechanismen erzielt werden. Eine Bedingung dabei ist jedoch, dass durch den verwendeten Mechanismus keine Reihenfolgevertauschungen verursacht werden. Eine Möglichkeit

[7] [RFC 2597]

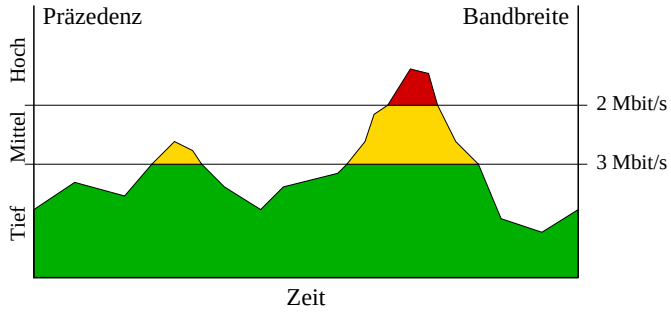


Abbildung 2.7: Unterteilung eines Stromes in Dropping-Prioritäten

ist es, einen modifizierten Random-Early-Detection-Algorithmus ([RED]) zu verwenden. Dieser verwirft ankommende Pakete mit einer vom Füllstand der Queue abhängigen Wahrscheinlichkeit. Indem jeder der drei Dropping-Prioritäten eine unterschiedliche Wahrscheinlichkeitsverteilung zugeordnet wird, erhält man das gewünschte Verhalten. Ein Beispiel findet sich in Abb. 2.8.

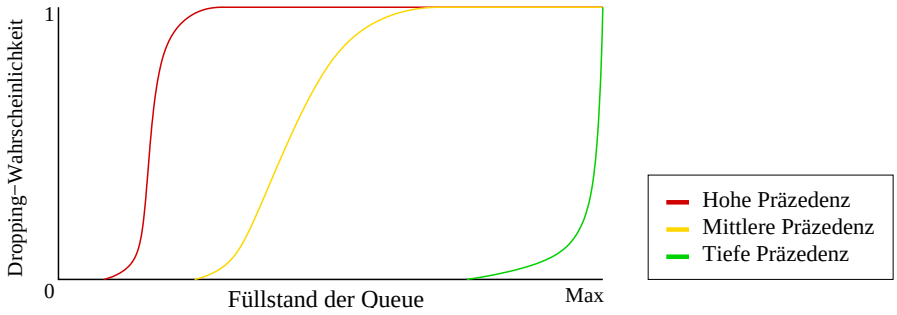


Abbildung 2.8: RED-Verteilungsfunktionen für drei Dropping-Prioritäten

2.4.5.4 Vergleich mit RSVP

Durch die Reduktion auf Behaviour Aggregates erreicht der DiffServ-Ansatz eine ungleich bessere Skalierbarkeit als RSVP, was vor allem beim Einsatz auf WANs grosse Vorteile bringt. Andererseits wird die Berechenbarkeit eines RSVP-Stroms nicht erreicht, da keine Pfadreservation stattfindet. In Sonderfällen kann dieser Nachteil aber vermindert werden, etwa indem für EF-Pakete statische Routingtabellen verwendet werden. Allgemein kann man sagen, dass RSVP ausgereifter ist, DiffServ aber grosse Flexibilität besitzt und sich aufgrund der oben genannten Vorteile in grossen ISP-Netzen durchsetzen könnte.
Es ist auch möglich, die Vorteile beider Ansätze zu kombinieren: [RSVPmap] schlägt vor, die Signalisierung von RSVP und die Reservationsart von DiffServ

zu kombinieren, indem DS-Domänen als einzelne Elemente in einem RSVP-Netzwerk agieren. Dadurch würde das Hauptproblem von RSVP, die schlechte Skalierbarkeit, verringert oder zumindest auf wenige Knoten eingeschränkt.

Kapitel 3

Lösungsansätze mit DiffServ

3.1 Auswirkungen der Serviceklassen

3.1.1 Grundlegende Argumentation

Die in [RFC 2475] vorgeschlagene DiffServ-Architektur lässt vieles offen. Auch für die Definition des Per-Hop-Behaviours einer DS-Domäne schreibt sie nur grobe Richtlinien vor. Es können aber “vernünftige” Annahmen darüber gemacht werden, welche Dienste ein Provider seinen Kunden anbieten wird.
Verschiedene Kunden haben auch verschiedene Bedürfnisse und finanzielle Möglichkeiten. Andererseits kann ein Provider aus Effizienzgründen nicht jedem Kunden eine massgeschneiderte Lösung anbieten. Es bietet sich also an, eine Anzahl grundlegender PHBs zu definieren, deren Parameter variiert werden können. Das dem Kunden angebotene Service-Level-Agreement beschränkt sich dadurch auf eine Kombination dieser PHBs und ihrer Parameter.
Weiter wird ein Provider seine Dienste in mehreren Preisklassen anbieten; beispielsweise kann die Angebotspalette “Basic”-, “Professional”- und “Deluxe”-Klassen enthalten. Dadurch ergibt sich Hierarchie der Dienstgüten.

Als konkretes Beispiel einer solchen Aufteilung in grundlegende PHBs sei hier die DiffServ-Implementierung der Universität Bern[1] herangezogen, mit der auch die weiter unten beschriebenen Experimente durchgeführt wurden. Die hier verwendeten PHBs sind Expedited Forwarding, die Assured Forwarding PHB-Gruppe, zwei spezielle PHBs für Netzwerk-Kontroll-Verkehr für den Domäneninternen Gebrauch und Best Effort. Man kann also zwischen drei grundlegenden PHBs wählen: EF für klar definierte Bandbreiten und kleine End-zu-End-Verzögerung, AF für unscharf definierte Bandbreitenanforderungen mit garantierter Mindestbandbreite und Best Effort für unsichere, aber billige Datenübertragung.
Die Assured-Forwarding-PHB-Gruppe unterscheidet vier Klassen, deren grund-

[1] [LinuxDS]

legendes Verhalten identisch ist. Die Parameter können aber für jede AF-Klasse frei gewählt werden, wodurch beispielsweise Angebote mit garantierter Mindest- und fester Maximalbandbreite möglich sind.
Die Eigenschaften einzelner PHBs können sich unter Umständen überschneiden. So kann bei einem wenig ausgelasteten Router eine Assured-Forwarding-Queue ähnliche Verzögerungswerte aufweisen wie die Expedited-Forwarding-Queue. Aufgrund des Scheduling-Algorithmus bietet die EF-Queue zu einem bestimmten Zeitpunkt jedoch immer gleiche oder bessere Verzögerungswerte als die anderen Queues im Router. Falls aber AF-Verkehr billiger als EF-Verkehr ist, oder die im SLA festgelegte EF-Bandbreite bereits ausgeschöpft ist, kann es sich durchaus lohnen, eigentlich für EF vorgesehene Pakete als AF-Pakete zu senden.

Die These, dass verschiedene PHBs unter Umständen vergleichbare Eigenschaften aufweisen können, wird durch die Publikation “Performance Evaluation of a Linux DiffServ Implementation[2]” unterstützt. Nebst dem eigentlichen Resultat, der Bestätigung der Leistungsfähigkeit von DiffServ mit AF- und EF-PHBs, zeigen die in dieser Arbeit enthaltenen Messungen, dass diese beiden PHBs zumindest dann sehr ähnliche Verzögerungs- und Jitter-Werte aufweisen, wenn die entsprechenden Serviceklassen nicht selbst überlastet sind.

Anwendungen mit DiffServ-Unterstützung können Eigenschaften konkreter Service-Level-Agreements, wie sie in diesem Abschnitt beschrieben wurden, mittels geeigneter Strategien ausnutzen und dadurch den gebotenen Service optimaler ausnutzen.

3.2 Trivialer Ansatz

Beim Einsatz von DiffServ für VoIP bietet es sich an, für alle anfallenden Daten eine einzige Serviceklasse zu benutzen, ähnlich wie bei RSVP. Premium Service ist schon vom Konzept her genau für solche Anwendungen vorgesehen. Ein Endpunkt wird in einem solchen Fall die benötigte Bandbreite per Signalisierung reservieren und dadurch eine Verbindung mit garantierter Bandbreite und kleiner Verzögerung erhalten.

Die Reservierungen innerhalb des Netzwerks können dabei statisch oder dynamisch geschehen. Im ersten Fall werden die Reservierungen abhängig von der Netzwerkstruktur statisch verteilt und werden nicht mehr geändert; die Endpunkte erhalten eine feste Bandbreite und müssen für eine Verbindung keine weiteren Re-

[2] [Stattb]

servierungen tätigen. In der Hierarchie weiter oben stehende Router erhalten dann eine Bandbreite zugeteilt, die der Summe der Endsysteme entspricht, die über sie senden.
Diese Lösung verursacht sehr wenig Verwaltungsaufwand und skaliert auch sehr gut, ist aber ineffizient. Solange nicht alle Endpunkte aktiv sind, wird die reservierte Bandbreite nicht ausgenutzt, was zwar angesichts der Netzwerkauslastung nicht weiter schlimm ist, da die Bandbreite von DiffServ-Routern dann für andere Serviceklassen verwendet werden kann. Innerhalb eines Firmennetzes mag diese Lösung also durchaus brauchbar sein. Sobald jedoch der Datenverkehr über einen ISP fliesst, wird dieser auch für nicht ausgelastete Reservierungen Geld verlangen, weshalb diese Reservationen möglichst klein gehalten werden sollten.

Dynamische Allokation von Bandbreiten vermindert dieses Problem signifikant. Wenn ein Endpunkt eine Verbindung aufbauen will, richtet er eine Reservierungsanfrage an den zuständigen MGC, welcher dann prüft, ob die Reservationen auf der Strecke zwischen den Endpunkten ausreichen. Ist dies nicht der Fall, kann er entweder auf den betroffenen Routern durch entsprechende Nachrichten mehr Bandbreite bereitstellen und dem Endpunkt die Reservation bestätigen, oder die Anfrage ablehnen. Wenn sich die Verbindung über mehrere Netze erstrecken soll (beispielsweise von Firmennetz A über ISP I zu Firmennetz B), gestaltet sich der Ablauf komplexer: Der für A zuständige MGC hat nur Kontrolle über das eigene Netz. Für Reservationen in den Netzen I und B muss er die jeweils zuständigen MGCs kontaktieren und zusätzliche Bandbreite aushandeln. Nur wenn die Antwort von beiden positiv ist, kann die Reservation bestätigt werden. Nach Abschluss der Verbindung meldet der Endpunkt dem MGC, dass die Reservationen rückgängig gemacht werden können, was dieser entsprechend weiterleitet. Zur Veranschaulichung siehe Abb. 3.1.
Die Aufgabe, die die MGCs hier übernehmen, wird allgemein als “Bandwidth Brokering” bezeichnet; Man könnte sie auch unabhängigen Bandwidth-Brokern übertragen.

Mit diesem Ansatz werden die überflüssigen Reservationen stark reduziert, zum Preis eines grossen Signalisierungsaufwandes. Die auftretenden Skalierungsprobleme sind zwar mit denen von RSVP nicht vergleichbar — das Problem betrifft schliesslich nur die Signalisierung und nicht das Routing — sie können sich aber merklich in der Dauer des Verbindungsaufbaus niederschlagen. Es gibt mehrere Wege, diesen Aufwand zu reduzieren. Zum einen kann der MGC Reservationen in grösseren Einheiten und zum Voraus tätigen, womit nur jede n-te Anfrage eines Endpunkts eine zusätzliche Reservation nötig macht. Damit tauscht man Reservations-Effizienz gegen Signalisierungs-Effizienz. Zum anderen können oft

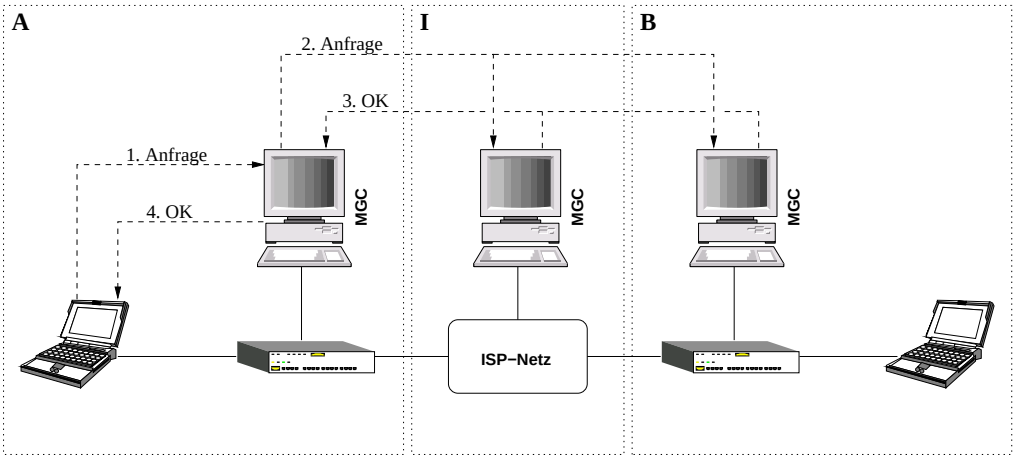


Abbildung 3.1: Allokation über mehrere Netze

benutzte Routen statisch reserviert werden, während alle anderen dynamisch bleiben, womit der Signalisierungsaufwand gezielt an den “Hot Spots” verringert werden kann.

Die in diesem Abschnitt besprochenen Ansätze haben mehrere Vorteile gegenüber einer Lösung mit RSVP. Der grösste dieser Vorteile ist, wie erwartet, die bessere Skalierung. Während RSVP-Router Tabelleneinträge für jede durch sie gehende Verbindung verwalten müssen, kann ein (intermediärer) DiffServ-Router die Pakete schlicht nach ihrem DSCP-Feld klassifizieren und entsprechend behandeln. Der Signalisierungsaufwand wird sich, bei “vernünftiger” Implementierung, im schlechtesten Fall in der Grössenordnung desjenigen von RSVP bewegen. Problematisch ist bei RSVP auch, dass die Endpunkte die Bandbreiten entlang der Route selber reservieren müssen, was dem MGC die Kontrolle über die verfügbare Bandbreite sehr erschwert. Auch das kann mit diesen Ansätzen gelöst werden, zusammen mit einem weiteren Problem: Nicht RSVP-fähige Endpunkte, die an ein RSVP-Netz angeschlossen werden, können keine Reservierungen tätigen und sind somit von diesem System ausgeschlossen. Nicht DiffServ-fähigen Endpunkten hingegen können von entsprechend konfigurierten First-Hop-Routern Serviceklassen zugeteilt werden. Eine DiffServ-Anwendung auf einem solchen Endpunkt kann sogar den First-Hop-Router durch Signalisierung rekonfigurieren und somit bewirken, dass ihre Best-Effort-Pakete mit einem anderen DSCP markiert werden.

Durch diese Methoden werden allerdings nicht alle Möglichkeiten von DiffServ ausgeschöpft. Es besteht zum Beispiel kein Zwang, dass alle Pakete eines Datenflusses denselben DSCP-Wert tragen müssen, er kann sogar von Paket zu

Paket ändern. Weiter ist die Beschränkung auf Expedited Forwarding häufig nicht optimal. Im weiteren werden deshalb komplexere Methoden vorgestellt, die diese Eigenschaften auszunutzen versuchen.

3.3 Adaptieren der Serviceklasse eines Flusses

3.3.1 Voraussetzungen

Wie in Abschnitt 3.1.1 erörtert, kann mit einer gewissen Wahrscheinlichkeit davon ausgegangen werden, dass ein Provider seinen Kunden vordefinierte Servicepakete anbietet, die nur innerhalb gewisser Grenzen an die speziellen Bedürfnisse eines Kunden angepasst werden können. Um die Verfügbarkeit der im Service-Level-Agreement spezifizierten Dienste garantieren zu können, muss der Provider Ressourcen reservieren, auch wenn diese vom Kunden nicht ausgelastet werden. Daher wird er dem Kunden zumindest eine Grundpauschale berechnen.
Es ist nun unwahrscheinlich, dass der vom Kunden erzeugte Datenstrom eine Verteilung von Serviceklassen aufweist, die das SLA optimal ausnutzt. Es kann durchaus vorkommen, dass eine Serviceklasse überlastet ist, währenddem eine andere fast brachliegt. Hier kommt die in 3.1.1 beschriebene Substituierbarkeit der Serviceklassen ins Spiel: Indem die Anwendungen die Serviceklasse der von ihnen produzierten Datenflüsse dynamisch den aktuellen Verhältnissen anpasst, kann die Auslastung auf mehrere Klassen verteilt und damit die Effizienz gesteigert werden.

Es ist auch denkbar, dass der Provider dem Kunden nicht einen Pauschalbetrag, sondern die übertragene Datenmenge pro Klasse berechnet. In diesem Fall gäbe es billigere und teurere Serviceklassen, und der Kunde würde für die Übertragung seiner Daten die jeweils billigste Serviceklasse wählen, die seine Anforderungen erfüllt. Da sich die Eigenschaften einer Serviceklasse während einer Übertragung ändern können, kann sich dynamische Anpassung auch hier lohnen.
Im einfachsten Fall umfasst der Spielraum für eine solche Adaption zwei Klassen, beispielsweise Best Effort und EF. Eine Anwendung kann nun Kosten sparen, indem sie ihre Daten solange mit Best Effort schickt, wie die erhaltene QoS genügt. Sinkt sie unter ein bestimmtes Niveau, wechselt die Anwendung auf EF, kehrt aber sofort wieder zu Best Effort zurück, sobald sich die Situation normalisiert hat.

3.3.2 Strategie

Wie kann eine solche dynamische Adaption in einer auf RTP basierenden VoIP-Anwendung implementiert werden? Eine der Anforderungen ist durch die Verwendung von RTP bereits erfüllt: Die Anwendung muss die Dienstqualität, die ihr Datenstrom erhält, beobachten, um gegebenenfalls darauf reagieren zu können. Eine solche Reaktion besteht normalerweise darin, die Kodierung der gesendeten Daten oder die Paketrate zu verändern. Dieses Verhalten kann nun durch das Adaptieren der Serviceklasse ersetzt oder ergänzt werden.

Der Anwendung steht eine begrenzte Anzahl möglicher Klassen zur Auswahl, von denen immer nur eine aktiv sein kann. Um im Fall einer Verschlechterung der QoS einen sinnvollen Klassenwechsel durchführen zu können, benötigt sie die Fähigkeit, die Klassen abhängig von der Situation in "besser" und "schlechter" einzuteilen. Damit dies algorithmisch geschehen kann, muss man also Relationen definieren:
Sei nun A die Menge aller berücksichtigten QoS-Anforderungen und S die Menge der wählbaren Serviceklassen. Wir definieren also für jede Anforderung $a \in A$ eine Relation $>_a$ auf S durch

$$k >_a l \;:\Leftrightarrow\; \text{“}k \text{ erfüllt Anforderung } a \text{ besser als } l\text{”},\; k, l \in S$$

Die verwandten Operatoren erhalten ihre übliche Bedeutung:

- $k \geq_a l \;:\Leftrightarrow\; \neg\, l >_a k$
- $k =_a l \;:\Leftrightarrow\; \neg\, k >_a l \;\wedge\; \neg\, l >_a k$

Die Umkehrungen $<_a$ und $\leq_a$ werden analog definiert. $k >_{a,b} l$ kann als Kurzschreibweise für $k >_a l \;\wedge\; k >_b l$ benutzt werden.

Eine Anwendung wird oft berechnen müssen, welche Serviceklassen bezüglich einer Anforderung die nächsthöhere oder nächsttiefere ist. Durch einen gerichteten Graphen kann ein System von Relationen auf einer Menge von Serviceklassen auf eine Weise dargestellt werden, die diese Operationen optimiert. Der Aufbau eines solchen Graphen geht folgendermassen vor sich:

Jede Serviceklasse bildet ein Element in der Menge K der Knoten. Für jeden Knoten $k \in K$ und jede Anforderung $a \in A$ führt eine mit a markierte Kante zu allen Knoten der Menge $\{l \in K \mid l >_a k \;\wedge\; \neg\exists\, m \in K : \; l >_a m >_a k\}$.

Abb. 3.2 zeigt den Graph zu folgendem Beispiel: $S = \{K_1, K_2, K_3\}$, $A = \{a, b\}$, $K_3 >_a K_2 >_a K_1$ und $K_3 =_b K_2 >_b K_1$.

Um die bezüglich der Anforderung a nächsthöhere Serviceklasse zu finden, kann nun einfach der mit a markierten Kante gefolgt werden. Existiert keine solche Kante, ist die Serviceklasse erreicht, die Anforderung a am besten erfüllt. Tiefere Klassen können durch verfolgen der Kanten in der umgekehrten Richtung gefunden werden.

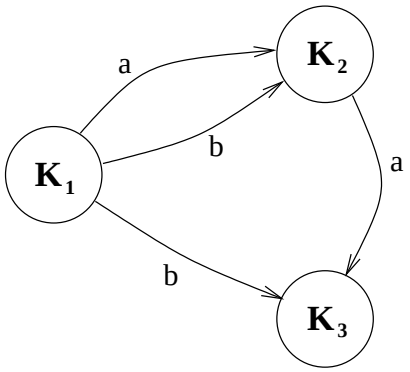


Abbildung 3.2: Beispielgraph für drei Serviceklassen

Für ein gegebenes SLA müssen nun sinnvolle Relationen gefunden werden. Gute Kandidaten dafür sind das Jitter- und Verzögerungsverhalten, die Paketverlustrate und die (finanziellen) Kosten der einzelnen Serviceklassen. Leider lassen sich kaum allgemeine Aussagen über das Verhalten einzelner PHBs machen. Einzig in Abschnitt 4.2.2.2 von [RFC 2474] lässt sich etwas dazu finden:

> We refer to a Class Selector Codepoint with a larger numerical value than another Class Selector Codepoint as having a higher relative order while a Class Selector Codepoint with a smaller numerical value than another Class Selector Codepoint is said to have a lower relative order. The set of PHBs mapped to by the eight Class Selector Codepoints MUST yield at least two independently forwarded classes of traffic, and PHBs selected by a Class Selector Codepoint SHOULD give packets a probability of timely forwarding that is not lower than that given to packets marked with a Class Selector codepoint of lower relative order, under reasonable operating conditions and traffic loads. A discarded packet is considered to be an extreme case of untimely forwarding. In addition, PHBs selected by codepoints '11x000' MUST give packets a preferential forwarding treatment by comparison to the PHB selected by codepoint '000000' to preserve the common usage of IP Precedence values '110' and '111' for routing traffic.

Somit kann zumindest davon ausgegangen werden, dass PHBs höherer Ordnung (d.h. mit einem numerisch grösseren Codepunkt) zumindest gleich gut sind wie solche mit niedriger Ordnung. Erfüllt also eine Serviceklasse die Anforderungen einer Anwendung nicht, kann diese die Suche nach einer besseren Klasse auf diejenigen höherer Ordnung beschränken. Alle weiteren Eigenschaften müssen dem SLA entnommen werden.

Um einer optimalen Ausnutzung des SLA möglichst nahe zu kommen, sollte eine Anwendung immer die Klasse wählen, die ihre Anforderungen gerade erfüllen kann. Diese Überlegungen führen zum folgenden, grundlegenden Algorithmus:

```
FOREVER
        Warte auf neue Übertragungsdiagnose
        Erstelle Liste mit Anforderungen, die nicht erfüllt werden
        IF Liste ist leer
                Erstelle Diagnose für nächsttiefere Klasse
                IF Diagnose gut
                        Wechsle zu dieser Klasse
                ELSE
                        Behalte aktuelle Klasse bei
                ENDIF
        ELSE
                Suche die nächsthöhere Klasse, die diese Anforderungen besser erfüllt
                IF gefunden
                        Wechsle zu dieser Klasse
                ELSE
                        Behalte aktuelle Klasse bei
                ENDIF
        ENDIF
LOOP
```

Dieser Pseudocode lässt offen, wie die Klassen und ihre Relationen intern dargestellt werden. Verschiedene Datenstrukturen sind denkbar, allen voran die oben eingeführte Graphendarstellung. Ist die Klassenaufteilung fest vorgegeben, können die Übergänge sogar fest in den Code integriert werden.
Es kann vorkommen, dass ein Klassenwechsel nicht klar definiert ist, da mehrere gleichwertige Klassen zur Verfügung stehen. Eine mögliche Lösung dafür besteht darin, die Relationen so zu wählen, dass dies nicht passiert. Alternativ kann auch ein nichtdeterministischer Übergang durchgeführt werden. In vielen Fällen können auch problemspezifische Informationen zur Auflösung des Konflikts verwendet werden.

Es ist wichtig, Folgendes zu realisieren: Die Anzahl von einem solchen Algorithmus berücksichtigter Serviceklassen muss nicht der Anzahl im SLA definierter PHBs oder PHB-Gruppen entsprechen. In einfachen Anwendungen kann es durchaus angebracht sein, nur die minimale Anzahl von zwei verschiedenen Klassen zu unterscheiden.

3.3.3 Das Problem des Zurückfallens

Wie oben erwähnt, kann davon ausgegangen werden, dass Datenflüssen mit einem PHB hoher Ordnung eine bessere oder zumindest gleichwertige Behandlung zuteil wird als solchen mit einem PHB kleinerer Ordnung. Anders ausgedrückt: Wenn es eine Serviceklasse mit besserem Verhalten als die zur Zeit benutzte gibt, dann ist sie sicher auch höherer Ordnung als diese. Es genügt somit, die Übertragungsqualität des gesendeten Datenflusses zu beobachten, um über eine eventuelle Erhöhung der Serviceklasse entscheiden zu können.

Anders verhält es sich in der umgekehrten Richtung, dem Zurückfallen auf eine Klasse tieferer Ordnung. Zwar kann für eine gegebene Serviceklasse ausgeschlossen werden, dass Klassen tieferer Ordnung eine bessere QoS aufweisen. Das genügt aber nicht, um über einen Klassenwechsel entscheiden zu können. Ist die mit der aktuellen Serviceklasse erhaltene QoS schlecht, kann ein Zurückfallen ausgeschlossen werden. Ist sie hingegen gut, kann über die QoS der nächsttieferen Klasse keine Aussage gemacht werden.

Eine naive Lösung zu diesem Problem wäre es, in periodischen Abständen versuchsweise zur tieferen Klasse zu wechseln, um sofort wieder zur alten Serviceklasse zurückzukehren, falls sich die Übertragungsqualität als zu niedrig erweist. In den meisten Anwendungen kann das ein gangbarer Weg sein, für VoIP ist er jedoch nicht akzeptabel, da jeder erfolglose Versuch einen merklichen Einbruch der Klangqualität oder sogar Signalunterbrüche bewirkt.

Offensichtlich muss ein Weg gefunden werden, mit genügender Sicherheit vorauszusagen, ob ein Zurückfallen erfolgreich wäre. In Netzen mit einer kleinen, übersichtlichen Struktur könnte ein einzelnes System dazu eingesetzt werden, die Auslastung aller Router zu überwachen, um den Anwendungen die für sie optimale Serviceklasse zuzuordnen. Eine solche zentralisierte Lösung ist aber in grösseren Netzen undenkbar, da sie sehr schlecht skaliert. Das Problem muss also von den Endpunkten her gelöst werden.

Der hier begangene Weg ist folgender: Eine kleine Anzahl von Paketen wird unabhängig vom eigentlichen Datenfluss zwischen den Endpunkten hin und her geschickt und ihr Verhalten beobachtet. Diese Pakete, Sondenpakete genannt, tragen den DS-Codepunkt der Serviceklasse, auf die die Anwendung zurückfallen würde. Durch das beobachtete Verhalten der Pakete kann sich die Anwendung ein grobes Bild davon machen, was für eine QoS bei einem Zurückfallen zu erwarten wäre, um dann aufgrund dieser Schätzung zu entscheiden.

Die beiden kritischen Parameter bei diesem Verfahren sind die Anzahl hin und her gesendeter Pakete (die Grösse der Stichprobe) und die Zeitspanne, in der die Messwerte ermittelt werden. Grundsätzlich wächst die Genauigkeit eines Schätzwertes mit der Grösse der Stichprobe, und je kürzer das Ermitteln der Messwerte dauert, desto schneller kann auf Veränderungen der Netzwerkauslastung reagiert werden.
Andererseits besteht bei zu kurzer Messdauer das Risiko, dass kurzfristige Veränderungen der Netzwerkauslastung nicht als solche erkannt werden, was zu ähnlichen Effekten wie das oben erwähnte versuchsweise Wechseln der Serviceklasse führt. Die durch das Verhältnis zwischen Anzahl und Zeitspanne gegebene Paketrate ist ein weiterer begrenzender Faktor: Durch das Senden von Sondenpaketen werden genau jene Werte verfälscht, die gemessen werden sollen. Das Ausmass dieser Verfälschung ist direkt von der Paketrate abhängig. Es gilt also, optimale Werte für Paketanzahl und Messdauer zu finden.

Um diese Optimierungsaufgabe zu vereinfachen, kann die von den Sondenpaketen beanspruchte Bandbreite, und damit die Paketrate, festgelegt werden. Sinnvollerweise kann sie, analog zu RTCP, einen festen Anteil der Gesamtbandbreite einer RTP-Sitzung einnehmen. Somit bleibt nur noch *ein* freier Parameter, wahlweise Paketanzahl oder Messdauer. Da die Paketanzahl inhärent ganzzahlig ist, bietet es sich an, sie als Parameter zu wählen. Welche Werte aber für diesen Parameter gewählt werden sollen, muss im konkreten Fall durch Experimente ermittelt werden. Das Ziel ist dabei immer, den Gesamteindruck auf die Gesprächsteilnehmer zu optimieren.

Es wäre denkbar, auch andere Klassen als die nächsttiefere mit Sondenpaketen zu analysieren, um so direkt zur optimalen Serviceklasse wechseln zu können. Da die Bandbreite für Sondenpakete aber beschränkt ist, würde sich die ohnehin schon im Bereich mehrerer Sekunden liegende Messzeit, und damit die Reaktionszeit, noch einmal stark erhöhen. Es erscheint daher nicht sinnvoll, das Verhalten höherer Klassen zu messen. Die Reaktionszeit auf Messungen des eigentlichen Datenstroms ist in solchen Fällen schneller und genauer, selbst wenn mehrere Wechsel nötig sind. Auch bei den tieferen Klassen lohnt es sich kaum, gleichzeitig

Messreihen für mehrere von ihnen durchzuführen, da die durch das Überspringen einer Klasse gewonnene Zeit die verlängerte Reaktionszeit nicht wettmacht.
Gibt man die Annahme auf, dass eindeutige Relationen bezüglich der Dienstqualität der Serviceklassen bestehen, erscheint die Situation jedoch anders. Da dann nicht mehr klar ist, welche Klassen “besser” sind, muss man sich durch mehrere gleichzeitige Messreihen einen Überblick verschaffen, um bei Bedarf zur optimalen Klasse wechseln zu können. Die von den Messreihen beanspruchte Zeit bewegt sich in diesem Fall aber wahrscheinlich im Bereich mehrerer zehn Sekunden, weshalb kurzfristige Entwicklungen nicht mehr erkannt werden können. Ausserdem müssen Anwendungen lange Zeit laufen, bevor sie eine Situationsbeurteilung erstellen können. Tritt ein Qualitätseinbruch auf, bevor dies geschehen ist, muss die Anwendung die Klasse auf gut Glück wechseln.

3.3.4 Kombination mit herkömmlichen Ansätzen

Das in diesem Abschnitt beschriebene Verfahren kann mit herkömmlichen Adaptionstechniken kombiniert werden. Anstatt die von den Teilnehmern einer RTP-Sitzung gesendeten Rückmeldungen nur dafür zu verwenden, über einen eventuellen Klassenwechsel zu entscheiden, können sie auch zum Optimieren der Übertragungsqualität mit Mitteln wie der Kodierungsadaption dienen. Kleinere Qualitätseinbussen, die noch keinen Klassenwechsel rechtfertigen, können so ausgeglichen werden.
Dabei ist zu beachten, dass die durch diese Adaptionstechniken veränderten Übertragungsparameter bei einem Klassenwechsel wieder rückgängig gemacht werden sollten, damit sie sich schneller an die plötzlich veränderte Situation anpassen können. Die erste nach einem Klassenwechsel eintreffende Rückmeldung jedes Empfängers muss ausserdem ignoriert werden, da die darin enthaltenen Informationen wahrscheinlich durch die Situation vor dem Wechsel beeinflusst sind und dadurch unnötige Parameterveränderungen provozieren können.

3.4 Markieren prioritärer Pakete

3.4.1 Idee und Voraussetzungen

Ein vollständig anderer Ansatz ist der, die einzelnen Pakete eines Datenflusses mit verschiedenen DS-Codepunkten zu markieren, wodurch sich die Wahrscheinlichkeit erhöht, dass einzelne Pakete korrekt und mit kleiner Verzögerung übertragen werden. Voraussetzung dafür ist, dass die Kodierung der Daten zwischen “wichtigen”, sprich prioritären, und “unwichtigen” Paketen unterscheiden kann.

Bei einigen Videokodierungen ist dies gegeben: Die einzelnen Frames werden in Form von Differenzen zum vorhergehenden Frame übertragen, ergänzt durch in regelmässigen Abständen gesendete Vollbilder, die eventuelle, durch verlorene Pakete verursachte Fehler korrigieren. Der Verlust eines solchen Vollbild-Pakets hat also eine grössere Auswirkung auf die Videoqualität als der Verlust eines normalen Pakets, wodurch eine natürliche Unterteilung in “wichtige” und “unwichtige” Pakete entsteht.
Das Ziel eines solchen Verfahrens ist, eine gewisse Mindestqualität der Übertragung garantieren zu können. Für die Sprachübertragung kann das beispielsweise heissen, dass Qualitätsschwankungen in Kauf genommen, Signalunterbrüche aber tunlichst vermieden werden.
Durch den verringerten Einsatz hoher Serviceklassen wird der Datenstrom vermehrt über die niedrigeren, billigeren Serviceklassen abgewickelt, was die Kosten optimiert.

3.4.2 Umsetzung

3.4.2.1 Wahl der Codepunkte

Damit der gewünschte Effekt erzielt wird und prioritäre Pakete tatsächlich einen besseren Service erhalten, müssen die verwendeten DS-Codepunkte geschickt gewählt werden. Dabei müssen zwei Punkte beachtet werden:
Die DiffServ-Spezifikation erlaubt, die acht möglichen Codepunkte auf nur zwei Per-Hop-Behaviours abzubilden. Deshalb muss man sicherstellen, dass die Wahl der Codepunkte auch tatsächlich in zwei verschiedenen PHBs resultiert, da sich sonst ein Wechsel nicht wie erwünscht auswirkt.
Weiter muss unbedingt vermieden werden, dass es zu Reihenfolgeumkehrungen kommt, was normalerweise bedeutet, dass die Pakete beider Behaviour Aggregates von den intermediären DS-Knoten in derselben Queue zwischengespeichert werden müssen. Dies ist beispielsweise zwischen den verschiedenen Dropping-Prioritäten innerhalb einer Assured-Forwarding-Klasse der Fall.

3.4.2.2 Wahl des Kodierungsschemas

Das grösste Problem bei der Umsetzung dieses Verfahrens ist die Wahl des Kodierungsschemas. Für die Echtzeit-Sprachübertragung sind keine Kodierungen gebräuchlich, die die gewünschte natürliche Unterscheidung in Pakete verschiedener Wichtigkeit aufweisen. Man könnte nun einfach einen gewissen Teil der Pakete, beispielsweise jedes vierte, zu einem prioritären Paket erklären, was aber nicht zum gewünschten Resultat führt. Die Wahrscheinlichkeit, dass Pakete verloren

gehen, sinkt zwar, jedes verlorene Paket führt aber nach wie vor zu einem Signalunterbruch.
Um das Verfahren umsetzen zu können, muss also ein neues Kodierungsschema definiert, oder zumindest ein existierendes abgeändert werden, so dass aus den prioritären Paketen eine unterbrechungsfreie, wenn auch qualitativ schlechte Rekonstruktion des ursprünglichen Signals gewonnen werden kann. Hierfür kommen verschiedene Ansätze in Frage, das Grundmuster bleibt aber das gleiche.
Nehmen wir der Einfachheit halber an, die Wahrscheinlichkeit, dass ein prioritäres Paket verloren geht, sei Null (Verspätete Ankunft eines Paketes kann als Sonderfall eines Paketverlustes betrachtet werden). Dann muss im Extremfall ausschliesslich aus den in prioritären Paketen enthaltenen Daten ein unterbrechungsfreies Signal rekonstruiert werden können. Beide, prioritäre und nicht-prioritäre Pakete, sollten also die gesamte Signaldauer abdecken. Aus ersteren sollte ausserdem eine gute Annäherung an das ursprüngliche Signal rekonstruiert werden können. Solche Kodierungen werden im Allgemeinen “hierarchisch” genannt. Abb. 3.3 zeigt ein Beispiel dazu.

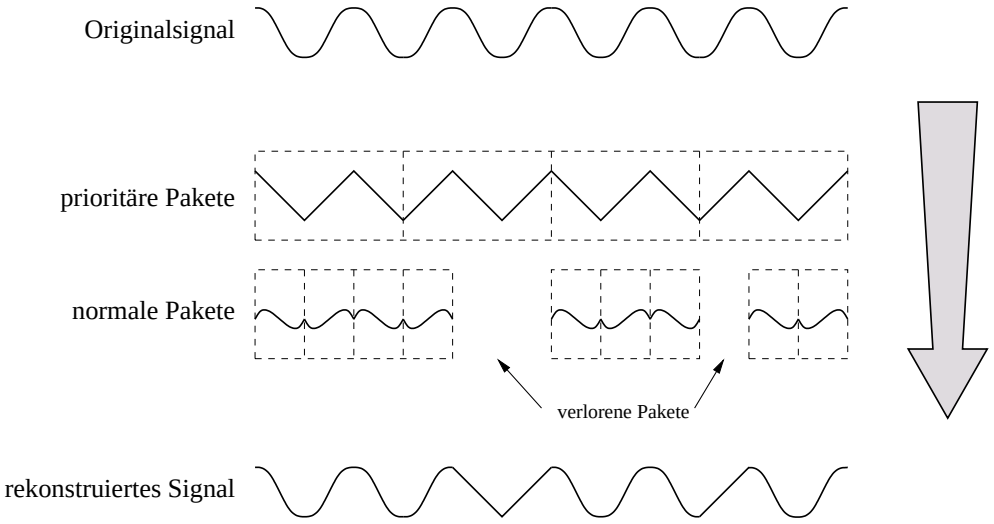


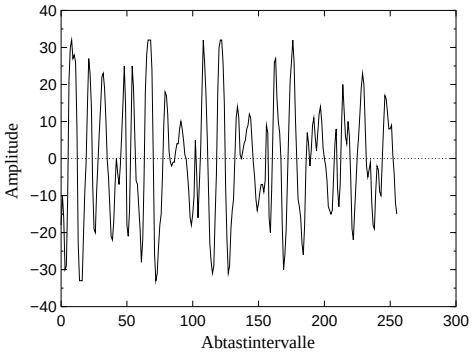
Abbildung 3.3: Beispiel für hierarchische Kodierung

Index- und Nibble-Splitting

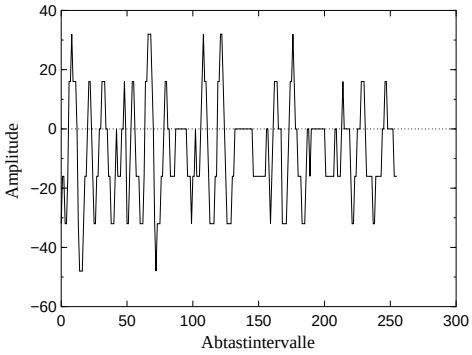
In diesem Abschnitt werden zwei einfache Kodierungsschemata vorgestellt, die speziell für den Ansatz des Markierens prioritärer Pakete entworfen wurden. Ihre gemeinsame Idee ist, samplebasierte Audiodaten so aufzuteilen, dass immer zwei Pakete denselben Zeitabschnitt abdecken.

Das erste dieser Schemata heisst *Index-Splitting* und basiert auf folgender Grundidee: Die in einem bestimmten Zeitintervall t aufgenommenen Samples $\{s_0, ..., s_{n-1}\}$ (n ein ganzzahliges Vielfaches von 2) werden in zwei Gruppen aufgeteilt, indem die Samples mit geradem bzw. ungeradem Index hintereinander abgelegt werden. Daraus ergeben sich zwei Datenblöcke mit den Inhalten $\{s_0, s_2, ..., s_{n-2}\}$ und $\{s_1, s_3, ..., s_{n-1}\}$, die beide das Zeitintervall t abdecken, und die in zwei getrennten Paketen gesendet werden. Geht nun eines der Pakete verloren, können die fehlenden Werte durch Interpolation rekonstruiert werden. Bei linearer Interpolation würde also für ein verlorenes Sample s_i der Wert $\frac{s_{i-1}+s_{i+1}}{2}$ eingesetzt werden.
Beide aus diesem Verfahren hervorgegangenen Pakete sind gleichwertig, es liegt also keine eigentliche hierarchische Kodierung vor. Indem aber beispielsweise die Pakete, die "gerade" Daten enthalten, markiert werden, erhält man das oben geforderte Verhalten. Das Verfahren kann ausserdem auf jedes samplebasierte Kodierungsschema angewandt werden.

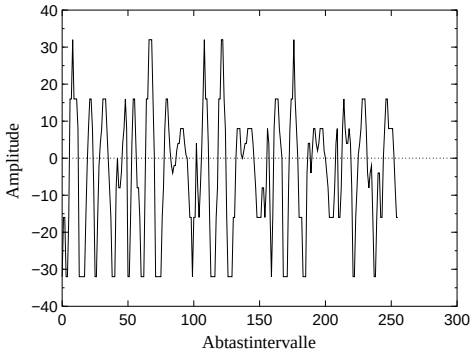
Nibble-Splitting wählt einen ähnlichen Ansatz. Auch hier werden die in einem Zeitintervall t aufgenommenen Samples in zwei Datenblöcke aufgeteilt, als Basis verwendet dieses Verfahren aber μ-Law. Die Samples $\{s_0, ..., s_{n-1}\}$ (n beliebig) werden weiter in Halbbytes (Nibbles) unterteilt, können also durch $\{s_0^h, s_0^l, ..., s_{n-1}^h, s_{n-1}^l\}$ dargestellt werden, wobei h und l für High- und Low-Nibble, also höher- und niederwertiges Halbbyte, stehen. Die resultierenden Datenblöcke haben die Form $\{s_0^h, s_1^h, ..., s_{n-1}^h\}$ und $\{s_0^l, s_1^l, ..., s_{n-1}^l\}$ und werden wiederum in getrennten Paketen versendet. Im Gegensatz zum Index-Splitting ist hier aber offensichtlich, welches Paket prioritär behandelt werden muss. Ersetzt man die niederwertigen Halbbytes mit 0, erhält man eine zwar schlechtere, aber immer noch verständliche Rekonstruktion des Stimmsignals, was unter anderem auf die μ-Law-spezifische, spezielle Gewichtung der Abtastwerte zurückzuführen ist. Abb. 3.4 verdeutlicht dies anhand einer Wellenform mit 8000 Hz Abtastrate, die einer realen Sprachaufnahme entstammt: Fallen bei Daten mit linearer PCM-Kodierung die unteren 4 Bits weg, ergeben sich extreme Abstufungen im Signal, die Sprache ist kaum noch verständlich. Passiert dasselbe bei einer μ-Law-Wellenform, sind zwar auch Abstufungen zu erkennen, die Sprache bleibt aber verständlich. Durch Interpolation kann dieses Resultat zusätzlich verbessert werden.
Setzt man hingegen alle höherwertigen Halbbytes auf 0, ist das ursprüngliche Signal nicht mehr zu erkennen. Einzige Ausnahme bilden sehr leise Signale, bei denen alle höherwertigen Halbbytes gleich Null sind. Diese können ohne Einbussen rekonstruiert werden.



(a) Original-Wellenform



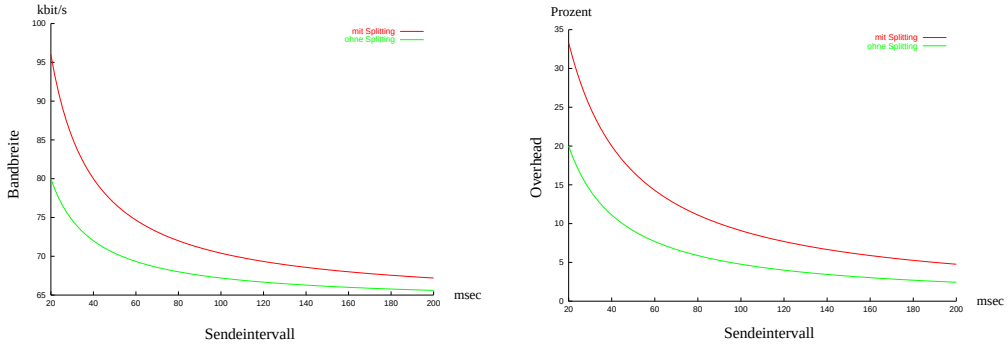
(b) Bei linearem PCM



(c) Bei μ-Law

Abbildung 3.4: Signalrekonstruktion bei wegfallendem Low-Nibble

Beide hier beschriebenen Kodierungsschemen erfüllen die zu Beginn des Abschnitts formulierte Forderung nach möglichst lückenloser Signalrekonstruktion, die Qualität der von ihnen erzeugten Rekonstruktion variiert jedoch. Index-Splitting erzeugt ein rauscharmes, aber dumpfes Signal, was sich vor allem bei Zischlauten störend bemerkbar macht. Mit Nibble-Splitting rekonstruierte Signale rauschen stark, erzielen aber eine bessere Verständlichkeit der Zischlaute, da der Frequenzbereich des Signals nicht beschnitten wird. Abb. 3.7 zeigt die mit beiden Methoden aus den höherwertigen Halbbytes generierten Rekonstruktionen der Original-Wellenform.



(a) Bandbreite (b) Overhead

Abbildung 3.5: Auswirkungen von Splitting

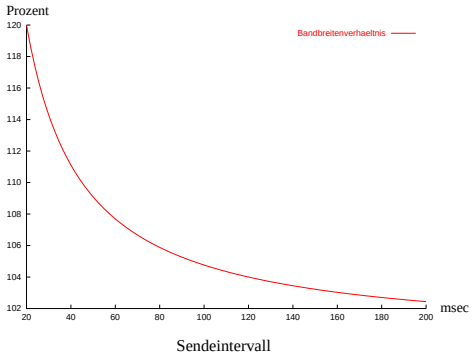


Abbildung 3.6: Verhältnis zu normalerweise benötigter Bandbreite bei Splitting

3.4.3 Bewertung

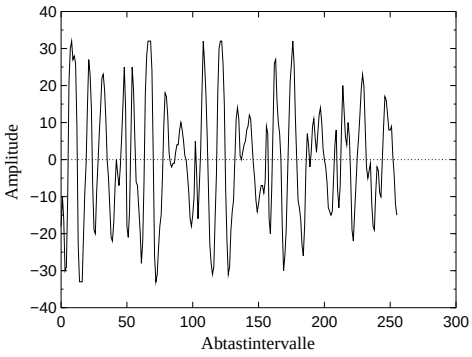
Der Ansatz des Markierens prioritärer Pakete erfüllt die an ihn gestellte Forderung, ein mit hoher Wahrscheinlichkeit unterbrechungsfreies Signal zu übermitteln. Er hat aber auch einige gewichtige Nachteile.

Durch die beiden oben besprochenen Splitting-Methoden verdoppelt sich die Anzahl zu übertragender RTP-Pakete und damit auch die Anzahl zu übertragender Paket-Header, was sich vor allem bei hohen Paketraten stark auf den Bandbreitenverbrauch auswirkt. Die Abbildung 3.5 zeigt die Auswirkungen des Splitting-Verfahrens auf Bandbreite und Overhead. Dabei wurde von einer Datenrate von 64 kbit/s und einer Headerlänge von 40 Bytes (was einem normalen RTP/UDP/IP-Header entspricht) ausgegangen. Abb. 3.6 verdeutlicht diese Auswirkungen mit einem prozentualen Vergleich der Bandbreitenanforderungen. Wie daraus hervor-

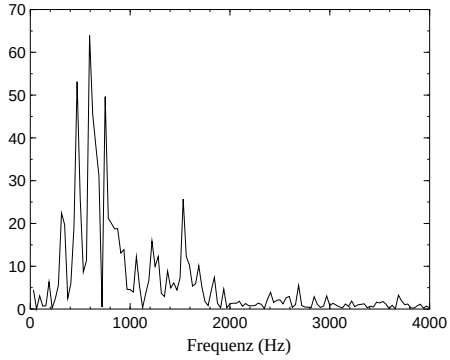
geht, sind die Effekte der Headerverdopplung umso stärker, je grösser die Paketrate ist. Bei 10 Paketen pro Sekunde (100ms Sendeintervall) verbraucht Splitting 4.8%, bei 16 (62.5ms) 7.4% und bei 40 Paketen (25ms) 16.6% mehr Bandbreite. Für Internettelefone der heutzutage verbreiteten Art mag das akzeptabel erscheinen, da deren Paketrate klein ist. Um die gewohnte Telefonqualität zu erreichen muss die Paketrate hoch sein, wodurch die zusätzlich benötigte Bandbreite 10% übersteigt.
Der Einsatz von Index- oder Nibble-Splitting könnte also bedeutende Mehrkosten verursachen, was einen der Hauptgründe für die breite Einführung von VoIP, die Kostenersparnis, neutralisieren würde.

Benützt man andere hierarchische Methoden, kann der zusätzlich generierte Overhead verringert werden, indem die Rate prioritärer Pakete gesenkt wird. Beispielsweise könnten auf ein prioritäres Paket immer vier normale kommen, wodurch sich der zusätzliche Bandbreitenbedarf um 75% verringert. Diese Lösung birgt aber ein anderes Problem: Da ein prioritäres Paket dasselbe Zeitintervall abdecken muss wie mehrere normale Pakete, vervielfacht sich die Verzögerung, die durch das Aufnehmen der Audiodaten entsteht. Bei 30ms Sendeintervall und einem Verhältnis zwischen prioritären und normalen Paketen von wiederum 1/4, bedeutet das eine Verzögerung von 120ms. Dazu kommt die durch Kodierung und Übertragung entstehende Verzögerung. Somit würde also ein kritischer Parameter bei Telefonanwendungen massiv verschlechtert.

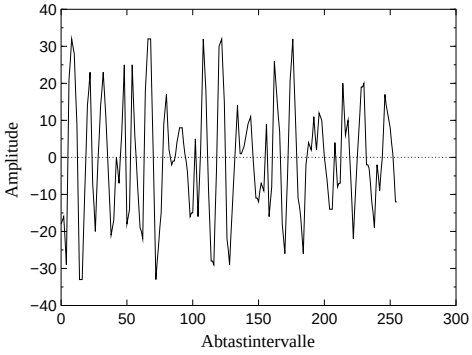
Fazit: Das Markieren prioritärer Pakete kann zwar helfen, eine lückenlose Übertragung zu sichern, bewirkt aber eine starke Erhöhung der benötigten Bandbreite oder der Verzögerung. Somit ist das Verfahren für die IP-Telefonie kaum verwendbar. Für unidirektionale Streaming-Applikationen wie Internetradio, in denen die Verzögerung keine wichtige Rolle spielt, könnte das Verfahren hingegen durchaus Verwendung finden, allerdings in der oben angetönten Form mit seltenen prioritären Paketen und nicht mit einer der Splitting-Methoden.
Beide Splitting-Methoden sind in der im nächsten Kapitel behandelten Implementierung teilweise enthalten. Da die genauere Betrachtung des Verzögerungsverhaltens aber zu obigen Resultaten führte, gibt es keinen Modus im Programm, der sie aktiv benutzt. Kapitel 5 enthält auch keinen entsprechenden Test.



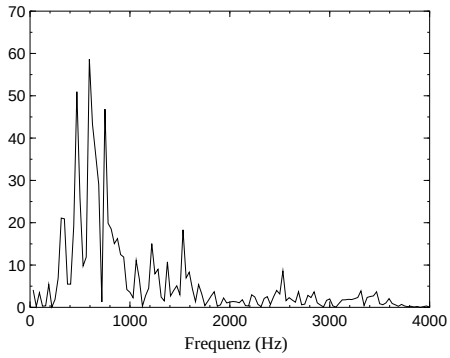
(a) Originalwellenform



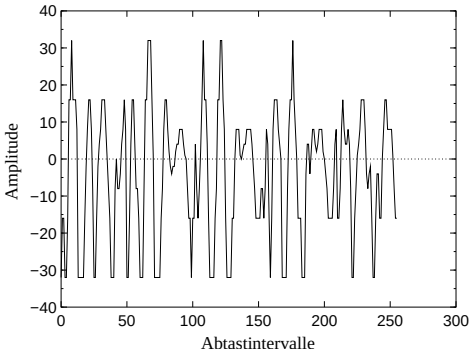
(b) Originalspektrum



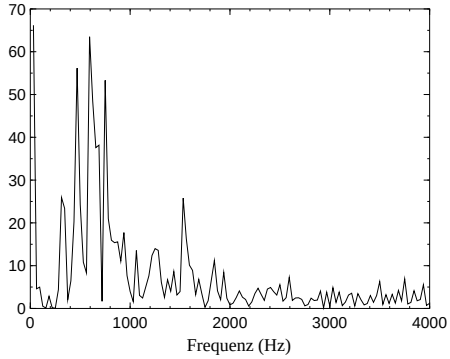
(c) Index-Splitting, Wellenform



(d) Index-Splitting, Spektrum



(e) Nibble-Splitting, Wellenform



(f) Nibble-Splitting, Spektrum

Abbildung 3.7: Vergleich der Splitting-Verfahren

Kapitel 4

Implementierung

4.1 Programme

4.1.1 DSPhone

Um die vorgeschlagenen Ansätze zur Optimierung von VoIP mit DiffServ experimentell testen zu können, musste ein Prototyp entwickelt werden. Sein Name ist DSPhone, was für DiffServ Phone steht. Dieser Abschnitt bietet eine kurze Übersicht über die Voraussetzungen der Implementation und die Fähigkeiten des Programms. Details zu Design und Implementation können in den nachfolgenden Abschnitten nachgelesen werden.

4.1.1.1 Voraussetzungen

Die Grundlage für die Implementation bildet die DiffServ-Variante der Universität Bern[1] mit den dazugehörigen Tools, die auf Intel-basierten Linuxsystemen installiert ist. Als Programmiersprache fand C++ Verwendung, ergänzt durch einige Perl-Skripten zur Auswertung der Messwerte. Die Ansteuerung der Audiohardware kann über die Bibliotheken des Enlightened Sound Daemon (ESD), der Advanced Linux Sound Architecture (ALSA) und des Open Sound Systems (OSS) erfolgen. Für die GUI-Programmierung wurde die Bibliothek Gtk-- verwendet, die ein klassenorientiertes Interface zur C-basierten Bibliothek GTK+ bietet. Verwaltung und Konfiguration geschieht mit CVS, GNU make und autoconf.

4.1.1.2 Fähigkeiten

DSPhone ermöglicht das Übermitteln von Audiodaten zwischen zwei Endsystemen und beinhaltet eine einfache RTP-Implementierung, die nur Sitzungen mit

[1] [LinuxDS]

zwei Parteien unterstützt. Da es sich um einen Prototypen handelt, wurden die wichtigen Schnittstellen flexibel gehalten, um das Experimentieren mit verschiedenen Komponenten zu ermöglichen. Speziell die Wahl des Medientransport-Protokolls[2], der verwendeten Audioformate und -geräte sowie die Parameter der einzelnen Komponenten innerhalb eines Medientransport-Protokolls sind dynamisch gehalten. Um die Synchronisation zwischen den einzelnen Teilen zu vereinfachen und deren Komplexität zu verringern, benutzt das Programm Multithreading.
Das Programm besteht aus mehreren Subsystemen (siehe Abb. 4.1 für einen Überblick). Das *Basissystem* stellt mit einem einfachen, proprietären Signalisierungsprotokoll Verbindungen zwischen zwei Endpunkten her, tauscht Konfigurationsparameter zwischen ihnen aus und erstellt das Backend der Verbindung. Zu diesem Zweck enthält es eine dynamische Konfigurationsdatenbank inklusive einem Parser und Generator für Konfigurationsdateien.
Das *Backend* übernimmt das Aufnehmen, Konvertieren, Mischen und Abspielen

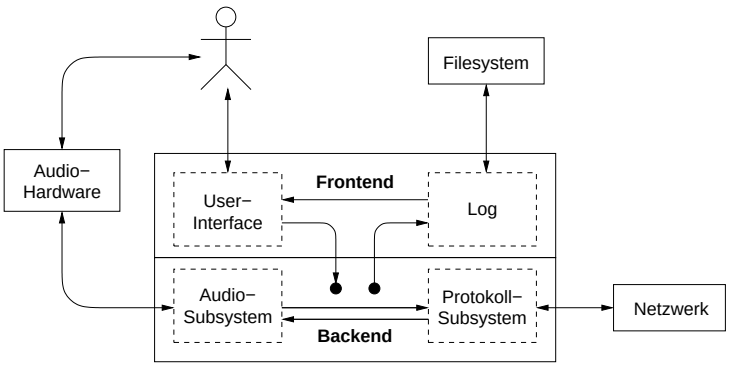


Abbildung 4.1: Architektur von DSPhone

der Audiodaten auf mehreren möglichen Geräten. Darunter sind auch logische Geräte wie das Null-Interface, das ein inaktives Audiogerät simuliert, oder das File-Interface, das Audiodaten aus einer Datei liest, beziehungsweise sie in eine Datei schreibt. Vor allem letzteres ist für Experimente unverzichtbar.
Damit das Backend den Eigenheiten eines Transportprotokolls angepasst werden kann, wird es auch vom *Protokoll*-Subsystem kontrolliert. Dieses ist auswechselbar (das RTP-Subsystem ist nur eine mögliche Variante) und übernimmt ausserdem die Aufgabe des Aufbaus und Betriebs der Medientransport-Kanäle.
Backend und Protokoll kennen 4 Modi: In *Send*- und *Receive*-Modi werden nur Audiodaten aufgenommen, beziehungsweise abgespielt. Ist ein Endsystem im Send-Modus, muss sein Gegenüber im Receive-Modus sein und umgekehrt.

[2] z.B. RTP

Der *Simple-Duplex*-Modus kann als Kombination von Send- und Receive-Modus betrachtet werden; Audiodaten werden sowohl aufgenommen als auch abgespielt, die beiden Ströme sind jedoch unabhängig voneinander. Insbesondere hört sich der Sprecher also nicht selbst, was aber für den nicht-experimentellen Einsatz eine durchaus wünschenswerte Eigenschaft sein kann. Im *Duplex*-Modus wird denn auch das empfangene Signal mit dem aufgenommenen gemischt, bevor es abgespielt wird. Abb. 4.2 verdeutlicht diese Betriebsarten.

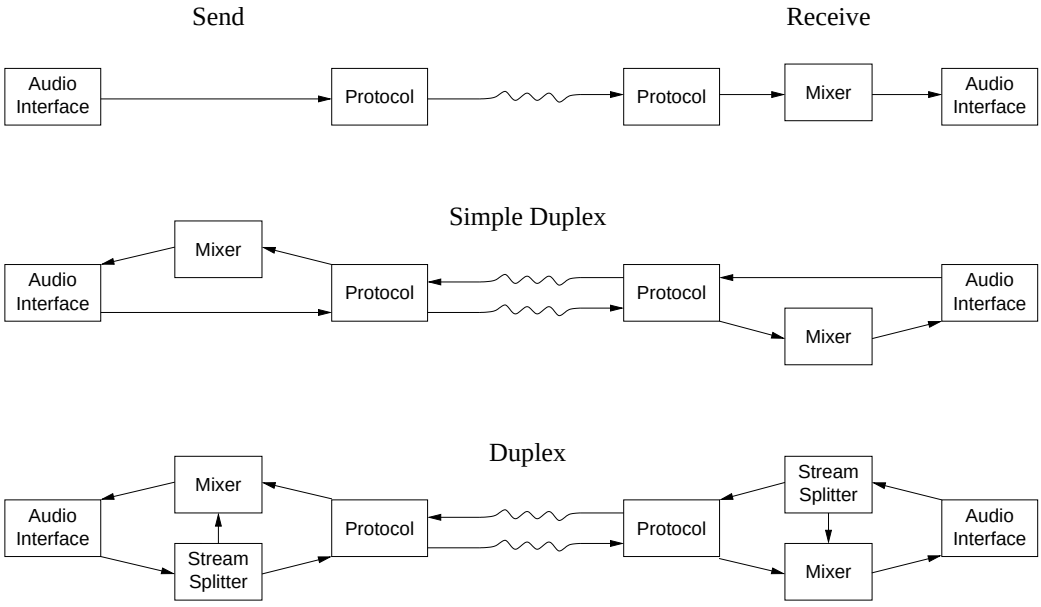


Abbildung 4.2: Backend-Modi

Zwei verschiedene *Frontends* übernehmen die Interaktion mit dem Benutzer, ein graphisches Interface und ein einfaches, kommandozeilenorientiertes. Das GUI besteht aus einem Verbindungsaufbau-Dialogfenster (Abb. 4.3), einem Konfigurationsfenster (Abb. 4.4) und einem Verbindungskontrollfenster (Abb. 4.5). Durch das Verbindungsaufbau-Dialogfenster kann der Benutzer einen Anruf einleiten, Adresse und Port der DSPhone-Instanz angeben, die angerufen werden soll, und auf einen Anruf warten. Ausserdem kann von hier aus das Konfigurationsfenster erreicht werden, wo die Parameter der verschiedenen Subsysteme verändert und in die Konfigurationsdatei geschrieben werden können. Sobald ein Anruf aufgebaut oder auf einen Anruf gewartet wird, erscheint das Verbindungskontrollfenster. Es besteht aus einem Feld, in dem Meldungen über den Anruf-Status und eventuelle Fehler dargestellt werden können. In der Konfiguration kann die Art

und Ausführlichkeit dieser Meldungen eingestellt werden. Das Fenster besitzt nur einen einzelnen Button, mit dem das Fenster geschlossen und damit der laufende Anruf beendet werden kann.

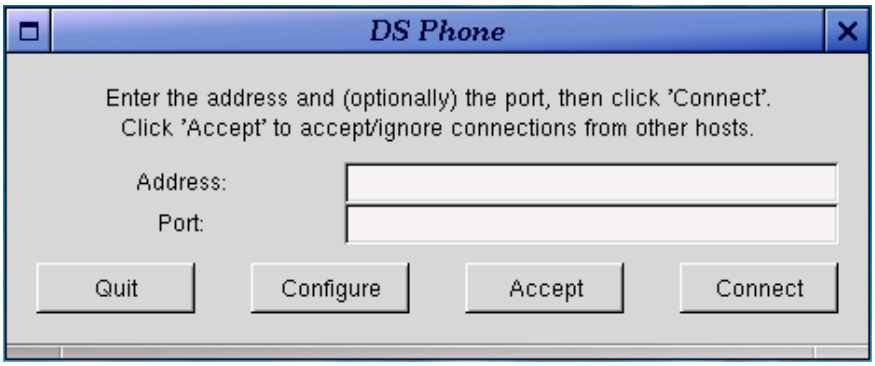


Abbildung 4.3: Verbindungsaufbaudialog

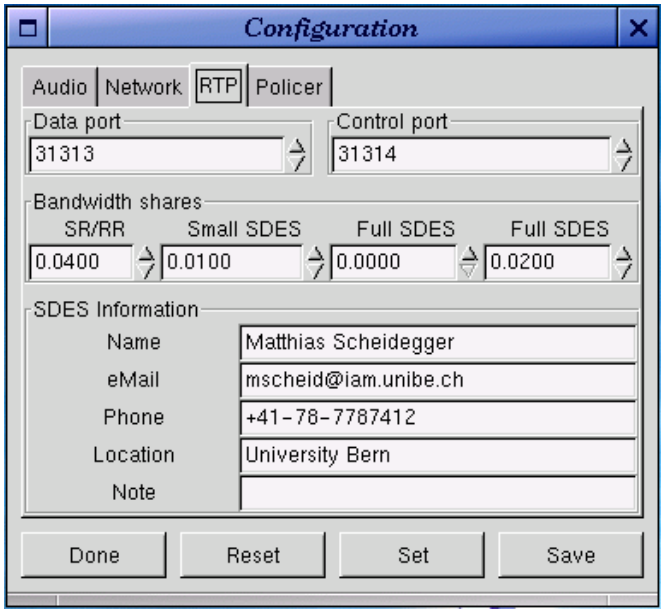


Abbildung 4.4: Konfigurationsfenster

Das ausführbare Programm der Kommandozeilenversion trägt den Namen *txt-phone* und erwartet die hauptsächlichen Anweisungen, wie “rufe host.mydomain auf Port 31313 an” oder “warte auf einen Anruf” in Form von Argumenten beim Programmaufruf. Die detaillierte Konfiguration wird über eine Konfigurationsdatei eingelesen, die frei gewählt werden kann. Wird keine Datei angegeben, ver-

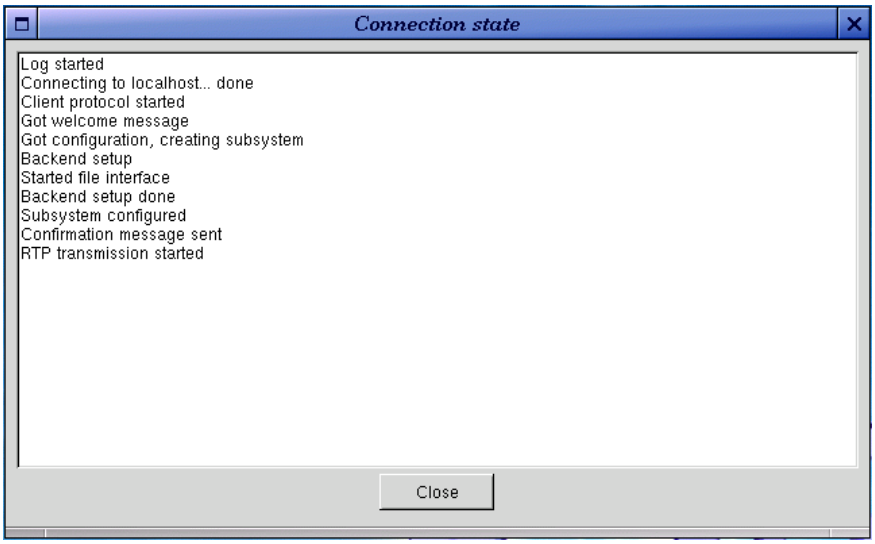


Abbildung 4.5: Verbindungskontrollfenster

sucht das Programm `~/.dsphonerc` zu lesen, falls existent. Andernfalls werden interne Standardwerte verwendet.
Wenn das Programm mit der Option `__DEBUG__` kompiliert wird, kann über die Konfigurationsvariable `debug_context` zusätzlich die Art der ausgegebenen Diagnosemeldungen spezifiziert werden. Dies wurde unter anderem dazu verwendet, die internen Werte und Entscheidungen im RTP-Subsystem auszugeben und für die durchgeführten Tests verfügbar zu machen.

4.1.1.3 Eigenschaften des RTP-Subsystems

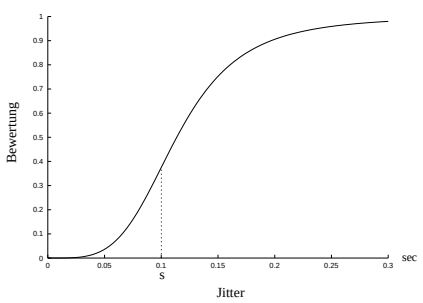
Die RTP-Implementierung von DSPhone erfüllt einen Grossteil der von [RFC 1889] geforderten Eigenschaften. Weil sich diese Arbeit aber auf Punkt-zu-Punkt-Verbindungen konzentriert, ist die Implementierung auf solche spezialisiert und unterstützt nur Sitzungen mit zwei Teilnehmern, was sich auch vorteilhaft hinsichtlich der Komplexität auswirkt. So können zum Beispiel einige Algorithmen ausgelassen werden, die sich mit Besonderheiten von grossen Sitzungen beschäftigen, wie zum Beispiel die SSRC-Kollisions-Erkennung oder die zufällig verteilten Sendezeiten für RTCP-Pakete, die verhindern sollen, dass zuviele Teilnehmer gleichzeitig RTCP-Pakete senden.
Um mit den verschiedenen in Kapitel 3 besprochenen Ansätzen experimentieren zu können, ist es notwendig, die Algorithmen, die über die verwendeten Serviceklassen entscheiden, auswechselbar zu machen. Diese Flexibilität wird erreicht, indem der gewünschte Algorithmus mitsamt seinen Parametern per Konfigurationsdatei gewählt werden kann.

Intern werden die aktuellen Übertragungsparameter in einer Instanz der Klasse *Policy* abgelegt, die verschiedenen Entscheidungsalgorithmen werden durch die abstrakte Klasse *Policer* gekapselt (nicht zu verwechseln mit der Router-Funktionalität des Policing). Zu Beginn einer Sitzung und immer dann, wenn neue Werte zur Übertragungsqualität vorliegen, entscheidet der gewählte Policer darüber, ob und wie die Policy-Parameter zu verändern sind. Bei einfachen Policern ist dies dann der Fall, wenn ein RR- oder SR-Paket empfangen wurde. Policer, die zwischen verschiedenen Serviceklassen wechseln können, greifen zusätzlich auf die Klasse *Scout* zurück, die mittels sogenannter Probe-Pakete versucht, die Übertragungsqualität einer anderen Serviceklasse als der gerade benutzten zu schätzen.
Der nächste Abschnitt gibt einen Überblick über die verschiedenen Policer und deren Entscheidungsalgorithmen.

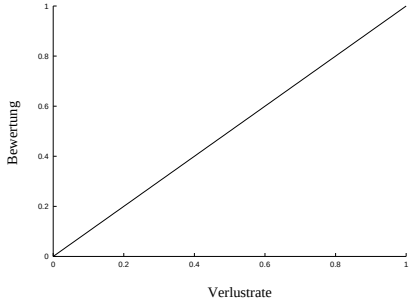
Policer

Alle Policer entscheiden über eine bestimmte Menge von Parametern, die in der *Policy*-Klasse zusammengefasst sind. Diese umfassen die aktuelle Länge und das Audioformat der gesendeten Paketinhalte sowie die DiffServ-Klasse, mit der die Pakete gesendet werden sollen. Als Audioformate stehen die vier im RTP-Profil definierten Formate zur Auswahl, die DiffServ-Klassen entsprechen den in [LinuxDS] verwendeten. Zusätzlich steht die Pseudo-Klasse `IGNORE` zur Verfügung, welche das System dazu veranlasst, den alten TOS-Wert zu übernehmen.
Alle bekannten Messwerte und Statistiken, die sich auf die aktuelle RTP-Sitzung beziehen, sind in der *RTPStats*-Klasse abgelegt. Auf ihre Attribute greift der aktive Policer zu, um die Situation beurteilen zu können. Die dabei hauptsächlich berücksichtigten Messwerte sind Jitter, Verzögerung und Paketverlustrate. Leider sind diese drei Werte nicht ohne weiteres vergleichbar, da sie völlig verschiedene Zahlenbereiche darstellen. Mithilfe von *Bewertungsfunktionen* können sie jedoch alle auf das Zahlenintervall [-1,1] abgebildet werden, das als eine Benotung zwischen “übertroffen” (-1), “erfüllt” (0) und “ungenügend” (1) aufgefasst werden kann. Je nach Entscheidungsalgorithmus kann so eine überdurchschnittlich gute Bewertung eines Parameters die nicht ganz genügende Bewertung eines anderen aufwiegen.
Die Funktionen heissen e_j, e_d und e_{pl}, für “Evaluate Jitter”, “Evaluate Delay” und “Evaluate Packet Loss” und wurden so gewählt, dass sie die Auswirkungen des jeweiligen Parameters auf die Übertragungsqualität berücksichtigen. Sie sind folgendermassen definiert:

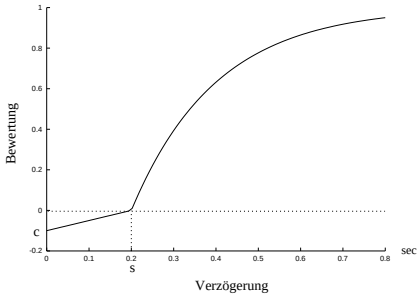
$$e_j(x) = \begin{cases} 1 - \frac{1}{1+\left(\frac{x \cdot tp}{s}\right)^4}, & x > 0 \\ 0, & sonst \end{cases}$$



(a) Jitter



(b) Paketverlustrate



(c) Verzögerung

Abbildung 4.6: Graphen der Bewertungsfunktionen

Die Konstante s (für Skalierung) wählt den Wendepunkt der Funktion, tp ist der Wendepunkt der Funktion im Falle $\frac{tp}{s} = 1$, d.h. sie skaliert die Funktion so, dass bei $s = 1$ der Wendepunkt auch tatsächlich 1 ist. Durch die Form der Funktion wird ausgedrückt, dass wenig Jitter kaum ein Problem darstellt, ab einem gewissen Ausmass aber zu grösserer Verzögerung führt, da der Synchronisationspuffer vergrössert werden muss. s ist der Punkt der grössten Steigung von e_j und sollte deshalb dem Wert entsprechend gewählt werden, ab dem ein grösserer Synchronisationspuffer nötig wird.

$$e_d(x) = \begin{cases} 1 - e^{-\frac{x-s}{s}}, & x > s \\ \frac{c(x-s)}{s}, & sonst \end{cases}$$

hat zwei Konstanten: s wählt die Nullstelle der Funktion, den Sollwert der Verzögerung. c (für Compensation) gibt an, wie stark sich eine Verzögerung unter dem Sollwert auswirkt. Es gilt $e_d(0) = c$. Die Funktion wurde so gewählt, weil

sich bei kleiner Verzögerung auch ein relativ kleiner Anstieg störend bemerkbar macht. Ist die Verzögerung aber schon gross, hat eine kleine Veränderung kaum mehr Einfluss auf die wahrgenommene Qualität des Gesprächs.

$$e_{pl}(x) = \begin{cases} m \cdot x, & x \geq 0 \\ 0, & sonst \end{cases}$$

Diese einfache Funktion hat nur eine Konstante, die Steigung m, die ausserdem meist gleich 1 gewählt wird. Die Einfachheit der Funktion erklärt sich daraus, dass sich die Verlustrate schon im Intervall [0,1] bewegt und direkt als Bewertungsfunktion übernommen werden kann. m kann in Einzelfällen dazu dienen, die Bewertung zu verschärfen.

Die in 3.3.3 erwähnten Sondenpakete werden in DSPhone von einem so genannten *Scout*-Objekt erzeugt. Dieses generiert applikationsspezifische RTCP-Pakete, die mit einer bestimmten Serviceklasse zwischen den Endpunkten hin und her gesendet werden, um damit Verzögerungsverhalten, Jitter und Verlustrate dieser Klasse abschätzen zu können. In Anlehnung an ihre ICMP-Entsprechung heissen sie Ping-Pakete. Ist eine Testreihe zu Ende, teilt das Scout-Objekt dem Policer die Resultate mit, die dieser dann in seinen Algorithmus einbeziehen kann.

DSPhone implementiert alle im Kapitel 3 erwähnten Ansätze. Da sich aber die Nachteile der beiden Splitting-Verfahren als zu gross herausstellten, wurden diese nicht weiter verfolgt. Es bleiben folgende Policer:
DefaultPolicer implementiert keinen Entscheidungsalgorithmus, sondern setzt die Policy auf festgelegte Werte: μ-Law-Format mit 8000Hz Abtastrate, DiffServ auf `IGNORE`. Dieser Policer soll Vergleichswerte für die mit anderen Policern erzielten Messwerte liefern.
Auch *StaticPolicer* ändert die Policy während einer Sitzung nicht. Im Unterschied zu DefaultPolicer ist jedoch auch die DiffServ-Klasse der gesendeten Pakete konfigurierbar, wodurch sich ermitteln lässt, wie sich Anwendungen unter den in 3.2 beschriebenen Bedingungen verhalten.

Gleich zwei Policer implementieren das in 3.3 vorgeschlagene Verfahren. Der einfachere davon ist *SwitchingPolicer*, der nur zwei Serviceklassen unterscheidet und je nach Bedarf zwischen ihnen hin und her wechselt. Somit eignet er sich für Situationen, in denen zwischen einer billigen und einer teuren Serviceklasse gewählt werden kann, und wo nur dann die teure Klasse benutzt werden soll, wenn die Dienstqualität der billigen nicht mehr ausreicht. Die billige Klasse wird hier "tiefe Klasse" die teure "hohe Klasse" genannt. Die Situation wird beurteilt, indem die Summe der Bewertungen der aktuellen Klasse mit einem Schwellenwert

verglichen wird. Ist die Summe grösser, gilt die Situation als schlecht. Ist die aktuelle Klasse hoch und die Übertragungsqualität gut, kommt eine Bewertung der durch den Scout ermittelten Situation in der tiefen Klasse hinzu. Um zu häufige Klassenwechsel zu vermeiden, wird eine solche Bewertung nur nach jedem dritten[3] SR- oder RR-Paket durchgeführt. Der in SwitchingPolicer implementierte Algorithmus hält sich eng an denjenigen von Seite 61 und ist in Abb. 4.7 abgebildet.

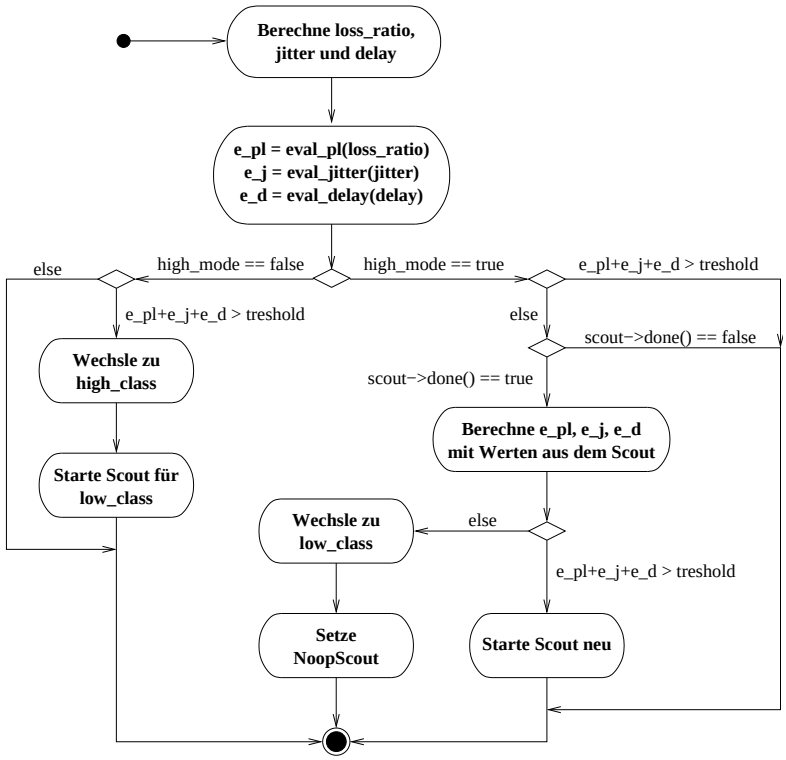


Abbildung 4.7: Action-State-Diagramm von SwitchingPolicer

Komplexer gestaltet sich der Algorithmus, den *ServicePolicer* implementiert. Dieser berücksichtigt alle von [LinuxDS] vorgesehenen Serviceklassen und deren Eigenheiten und beurteilt Jitter, Verzögerung und Verlustrate getrennt, durch Vergleich mit drei verschiedenen Schwellenwerten (t_j, t_d und t_{pl}). Auch hier wird die Situation nur nach jedem dritten SR- oder RR-Paket neu bewertet.
Die Relationen für die einzelnen Anforderungen wurden folgendermassen gewählt (j steht für Jitter, d für Verzögerung, pl für Verlustrate): Premium Service $>_{d,pl}$ Assured Service 4 $>_{d,pl}$ ... $>_{d,pl}$ Assured Service 1 $>_{d,pl}$ Best Effort, und

[3]Dieser Wert ist konfigurierbar

Premium Service $>_j$ Assured Service 4 $=_j$ ... $=_j$ Assured Service 1 $=_j$ Best Effort[4] (Diese Relationen sind Annahmen und vor allem dann unrealistisch, wenn die Premium-Service-Klasse überlastet ist). Für Verzögerung und Paketverlust besteht also eine klare Ordnung. Anders für Jitter, wo nur zwischen Premium Service und den restlichen Klassen unterschieden werden kann, sich also zwei Äquivalenzklassen ergeben. Steigt Jitter zu stark an, muss also von jeder anderen Serviceklasse direkt zu Premium Service gesprungen werden. Dadurch ist aber nicht mehr eindeutig, zu welcher Klasse allenfalls zurückgekehrt werden soll. Der ursprüngliche Algorithmus muss hier präzisiert werden: Die "nächsttiefere Klasse", zu der zurückgefallen werden soll, ist bestimmt durch die vom aktuellen Scout-Objekt untersuchte Serviceklasse. Ein solches Scout-Objekt beobachtet nach jeder Klassenerhöhung die gerade verlassene Klasse. Springt ServicePolicer also von Assured Service 1 direkt zu Premium Service, muss er auch wieder auf Assured Service 1 zurückfallen.
Nach dem Zurückfallen muss die neu zu beobachtende Serviceklasse ermittelt werden. Diese ist aber glücklicherweise durch die Relation $>_{d,pl}$ eindeutig bestimmt. Im obigen Beispiel wäre die "nächsttiefere Klasse" nach dem Zurückfallen also Best Effort.
Zustandsdiagramm und Algorithmus von ServicePolicer sind in Abb. 4.8 und Abb. 4.9 zu finden.

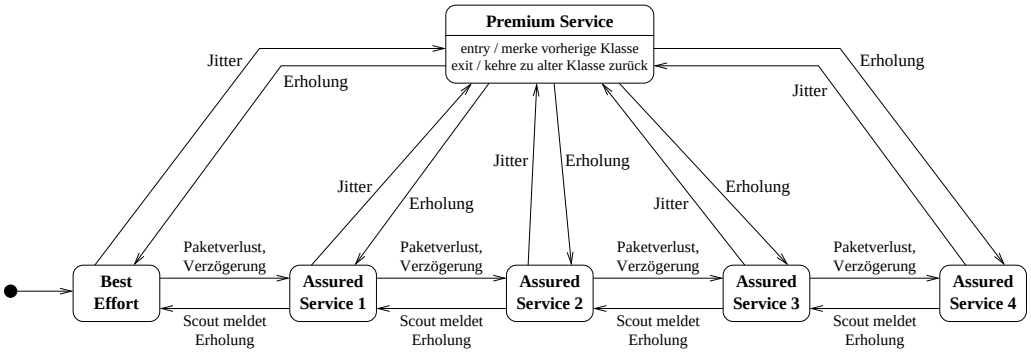


Abbildung 4.8: Zustandsdiagramm von ServicePolicer

[4]Zur Erinnerung: Premium und Assured Service werden als Synonyme für Expedited und Assured Forwarding verwendet.

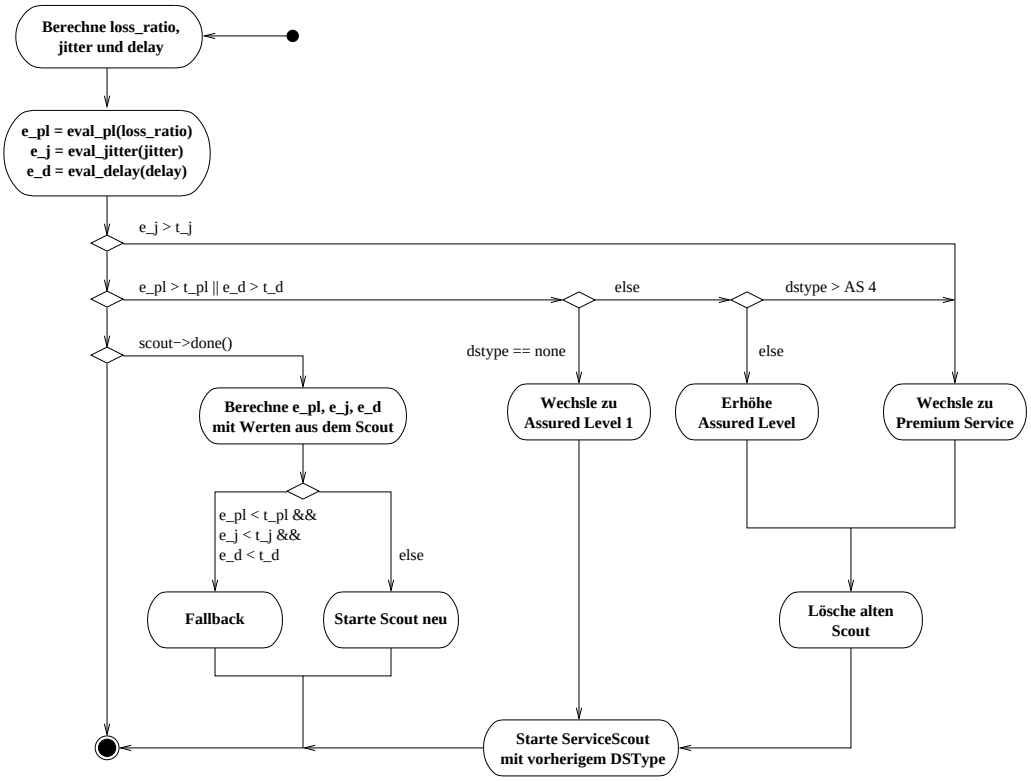


Abbildung 4.9: Action-State-Diagramm von ServicePolicer

RTP-Profil

DSPhone verwendet folgendes RTP-Profil: Die Protokoll-Version, wie sie im Headerfeld erscheint, ist 2.
Das X-Bit wird nicht verwendet, und die Bits des Payload-Feldes haben die Bezeichnungen __TQSNP, mit folgender Bedeutung:

T	Type	0 - lineares PCM, 8 Bit vorzeichenbehaftet
		1 - μ-Law
Q	Quality	0 - Standard, 8000Hz
		1 - Extended, 22050Hz
S	Split	0 - normales Paket
		1 - Paket enthält aufgeteilte Daten
N	Splitting-Type	0 - Index-Splitting
		1 - Nibble-Splitting
P	Split-Part	0 - Gerade Samples / höherwertige Nibbles
		1 - Ungerade Samples / niederwertige Nibbles

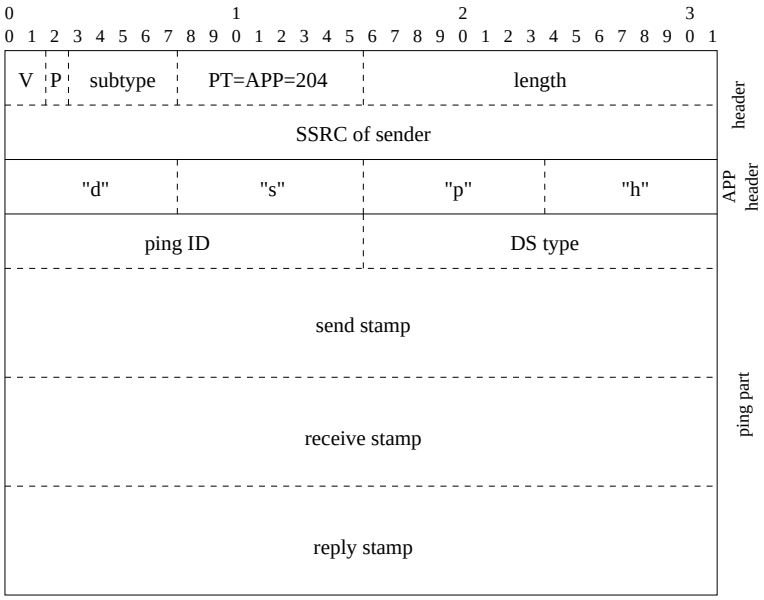


Abbildung 4.10: Ping-Paket-Format

Aktive Teilnehmer reservieren einen festen Anteil ihres Datenstromes für RTCP-Pakete, passive nehmen die empfangenen Pakete als Referenz. Die Bandbreiten sind 5% für normale RTCP-Pakete und weitere 2% für Sondenpakete.

APP-Pakete identifizieren sich durch die Zeichenkette “dsph” und haben zwei Subtypen: Ping (Identifikator 0) und Ping Reply (Identifikator 1). Bis auf diese Typeninformation ist das Format der beiden identisch. Siehe Abb. 4.10. Die Felder dieser Pakete haben folgende Bedeutung:

Subtype: 5 Bits
Der APP-Subtyp des Pakets, entweder 0 oder 1, für Ping und Ping-Reply.

Ping ID: 16 Bits
Identifiziert das Paket in einer Messreihe. Ping- und Ping-Reply-Pakete werden über diesen Wert einander zugeordnet.

DS Type: 16 Bits
Der in 16 Bit kodierte DiffServ-Typ eines Ping-Pakets. Ping-Replies sollten mit diesem Typ zurückgeschickt werden.

Send Stamp: 64 Bits
NTP-Timestamp der Sendezeit des Ping-Pakets. Wird in Ping-Replies übernommen.

Receive Stamp: 64 Bits
: Nur in Ping-Replies gesetzt: NTP-Timestamp der Empfangszeit des Ping-Pakets, auf das geantwortet wird.

Reply Stamp: 64 Bits
: Nur in Ping-Replies gesetzt: NTP-Timestamp der Sendezeit des Ping-Reply-Pakets.

Empfängt ein Teilnehmer einer Sitzung ein Ping-Paket, muss er so schnell als möglich ein entsprechendes Ping-Reply-Paket zurücksenden.

4.1.1.4 Bedienung

DSPhone besteht aus zwei Programmen, dem GUI-orientierten `dsphone` und dem im Textmodus arbeitenden `txtphone`. Das erstere wird mit der Kommandozeile

dsphone [-r *Konfigurationsdatei*]

gestartet. Wird keine Konfiurationsdatei angegeben, verwendet das Programm die Datei `~/.dsphonerc`, falls diese existiert. Die Bedienung der Benutzer-Schnittstelle ist intuitiv und sollte selbsterklärend sein.
`txtphone` erwartet eine Kommandozeile der Form

txtphone [-r *Konfigurationsdatei*] {server [*port*] — client *server* [*port*]}

wobei Option `-f` dasselbe bewirkt wie bei `dsphone`. Das Schlüsselwort `server` lässt das Programm auf einen Anruf warten. `client` startet eine Verbindung zum angegebenen Server. Der zu verwendende Port kann bei beiden optional angegeben werden.

Konfigurationsdatei

Die von DSPhone erwartete Konfigurationsdatei hat das folgende Format:
Zeilen, die mit einer beliebigen Anzahl von Leerzeichen beginnen, die wiederum von einem '#' gefolgt werden, gelten als Kommentarzeilen und werden ignoriert. Leere Zeilen und solche, die nur aus Leerzeichen bestehen, werden ignoriert. Alle anderen Zeilen haben das Format

name = wert

wobei *name* ein bekannter Variablenname und *wert* der Variableninhalt vom entsprechenden Typ ist. Werte können Integer oder Fliesskommazahlen, Boolesche Werte oder von Anführungszeichen eingefasste Strings sein.
Die Tabellen 4.1 und 4.2 enthalten eine Liste der bekannten Variablen. Diese ist nicht ganz vollständig, da einige Variablen nur der Fehlersuche dienen oder Parameter in internen Algorithmen definieren.

In den folgenden Tests werden zusätzlich folgende, eigentlich für die Fehlersuche vorgesehenen, Variablen verwendet: debug_context bestimmt, welche internen Fehler- und Diagnosemeldungen auf `stderr` ausgegeben werden, was auch für die Analyse der Policer-Algorithmen verwendet werden kann.
Mittels debug_use_redirection kann eine RTP-Sitzung mit zwei Teilnehmern auf demselben System simuliert werden, indem die gesendeten UDP-Pakete per `dlo` umgeleitet und gegebenenfalls verzögert werden.

4.1.2 Tools

Verschiedene kleinere Programme waren notwendig, um das Verhalten von DSPhone testen zu können und Fehler zu suchen.

DLO

Damit eine RTP-Sitzung auf einem einzelnen System analysiert werden kann, dürfen nicht beide Instanzen dieselben UDP-Ports wählen. Ausserdem sollten die Pakete leicht verzögert werden, um den Test realistischer zu machen. Zu diesem Zweck werden die RTP- und RTCP-Pakete beider Parteien durch `dlo` (Delayed LOopback) umgeleitet und verzögert. Abb. 4.11 zeigt, wie diese Umleitung aufgebaut ist. In DSPhone kann durch die Variable debug_use_redirection bewirkt werden, dass RTP und RTCP nicht die normalen Ports, sondern Umleitungen auf `dlo` verwenden.

DLO wird von der Kommandozeile mit folgendem Aufruf gestartet:

dlo [-d *Verzögerung*] *port-c1 port-w1 port-w2 port-c2*

Verzögerung bestimmt die Dauer in Millisekunden, die zwischen Empfang und Weiterleiten vergehen soll. *port-c1* und *port-c2* sind die Ziel-Ports, die den beiden Clients gehören, *port-w1* und *port-w2* die Weiterleitungs-Ports, über die `dlo` Pakete empfängt und weiterleitet.

UDPgen / UDPrcv

Um das Verhalten der verschiedenen Algorithmen unter veränderlichen Netzauslastungen testen zu können, muss auf dem Testnetzwerk Datenverkehr generiert

Name	Typ	Beschreibung
ctrl_port	Integer	eigener Signalisierungsport
peer_port	Integer	Signalisierungsport der Gegenseite
audio_dev	String	Zu benutzendes Audiogerät: “null”, “loopback”, “file”, “oss”, “alsa” oder “esd”
audio_bits	Integer	Bittiefe der aufgenommenen Audiodaten: 8 oder 16
audio_rate	Integer	Samplerate der Daten in Hz, z.B. 8000
audio_signed	Bool	Sollen die Daten vorzeichenbehaftet sein
audio_infile	String	Eingabedatei für FileIface
audio_outfile	String	Ausgabedatei für FileIface
msg_normal	Bool	Zeige normale Log-Nachrichten
msg_verbose	Bool	Zeige detaillierte Log-Nachrichten
dbg_normal	Bool	Zeige normale Fehler-Nachrichten
dbg_verbose	Bool	Zeige detaillierte Fehler-Nachrichten
protocol	String	Medientransport-Protokoll: “rtp” oder “simpletcp”
proto_mode	String	Protkoll-Modus (Backend-Modus): “duplex”, “simple duplex”, “send”, “receive”
bandwidth	Integer	Bandbreite des Netzanschlusses, in Byte/sec
wanted_delay	Integer	Gewünschte End-zu-End-Verzögerung, in msec
rtp_port	Integer	Datenport der RTP-Sitzung
rtcp_port	Integer	Kontrollport der RTP-Sitzung, normalerweise rtp_port+1
diag_share	Float	Bandbreitenanteil der RR- und SR-Pakete
sdes_small_share	Float	Bandbreitenanteil der kleinen SDES-Pakete (enthalten nur CNAME)
sdes_full_share	Float	Bandbreitenanteil der grossen SDES-Pakete (enthalten alle bekannten Felder)
probe_share	Float	Bandbreitenanteil der Sondenpakete
sdes_name	String	Name des Teilnehmers
sdes_email	String	Email
sdes_phone	String	Telefonnummer
sdes_loc	String	Standort
sdes_tool	String	Benütztes Programm (Default “dsphone”)
sdes_note	String	Notiz

Tabelle 4.1: Konfigurationsvariablen

Name	Typ	Beschreibung
policer	String	Verwendeter Policer: “default”, “static”, “switching” oder “service”
packetization_interval	Integer	Senderate der Pakete in msec
static_dstype	String	Von StaticPolicer verwendete DiffServ-Klasse: “ignore”, “none”, “assured” oder “premium”
low_type	String	Tiefe DS-Klasse von SwitchingPolicer, wie in static_dstype
high_type	String	Hohe DS-Klasse von SwitchingPolicer
switching_treshold	Float	Schwellenwert von SwitchingPolicer
packet_loss_treshold	Float	Schwellenwert der Verlustrate in ServicePolicer
jitter_treshold	Float	Schwellenwert des Jitter
delay_treshold	Float	Schwellenwert der Verzögerung
evaluation_interval	Integer	Anzahl empfangener SR- oder RR-Pakete zwischen den Neubewertungen in SwitchingPolicer und ServicePolicer

Tabelle 4.2: Konfigurationsvariablen

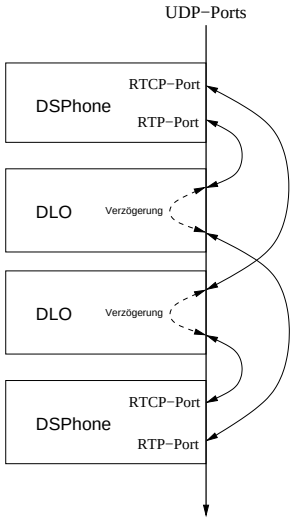


Abbildung 4.11: Umleitungsschema von DLO

werden. Herkömmliche Datenverkehrs-Generatoren können nur mit vorher festgelegten Bandbreiten senden, was aber nicht genügt, um kurzfristige Datenstaus zu simulieren. Aus diesem Grund entstand UDPgen und sein Gegenstück UDPrcv. UDPgen sendet UDP-Pakete an ein festgelegtes Ziel und verändert die generierte Bandbreite im Verlauf der Zeit. Die Bandbreitenentwicklung wird einer aus linearen Segmenten bestehenden Funktion entnommen. Diese Segmente werden durch Wertepaare der Form (*Dauer, End-Bandbreite* definiert, die entweder von `stdin` oder von einer Skriptdatei gelesen werden können (siehe Abb. 4.12 für ein Beispiel).
UDPrcv empfängt die von UDPgen gesendeten Pakete und gibt in gewissen Abständen die von diesen Paketen benutzte Bandbreite aus. Kommen die Pakete in grösseren Abständen an, als Meldungen ausgegeben werden sollen, gibt UDPgen die Bandbreite für jedes empfangene Paket aus.

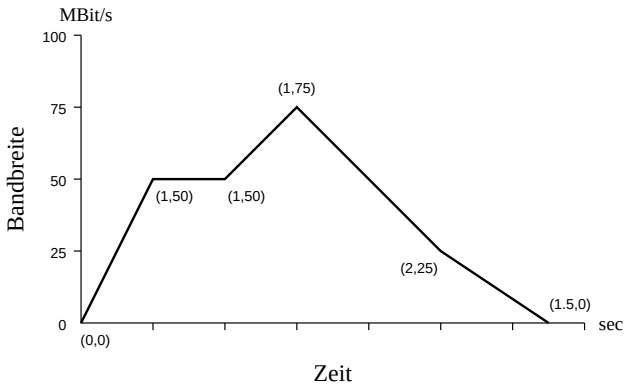


Abbildung 4.12: Beispiel für einen UDPgen-Bandbreitenverlauf

Um den Bandbreitenverlauf so genau wie möglich wiedergeben zu können, implementiert UDPgen einen speziellen Algorithmus. Dieser benutzt die folgenden Variablen und Konstanten:

f	-	die Verlaufsfunktion
i	-	zu betrachtendes Intervall der Funktion
i_{min}	-	minimales Intervall
A	-	Fläche unter einem bestimmten Intervall von f
t_s	-	absolute Startzeit des Algorithmus
t_d	-	seit t_s vergangene Zeit
t_w	-	zu wartende Zeit
s	-	Anzahl in einem Intervall zu sendender Bytes
s_{min}, s_{max}	-	Minimale und maximale Paketgrösse

Der Algorithmus ist folgender:

t_s = jetzt()
t_d = 0

WHILE (Skript nicht zu Ende) {
    t_d = jetzt() - t_s
    $i = i_{min}$
    $A = \int_{t_d}^{t_d+i} f(t)\ dt$

    IF (A = 0) {
        Finde ein maximales, ganzzahliges c so, dass
        $\int_{t_d}^{t_d+c \cdot i_{min}} f(t)\ dt\ =\ 0$
        und setze $i = c \cdot i_{min}$
    } ELSIF ($A < s_{min}$) {
        Finde ein minimales, ganzzahliges c so, dass
        $\int_{t_d}^{t_d+c \cdot i_{min}} f(t)\ dt\ \geq\ s_{min}$
        und setze $i = c \cdot i_{min}$
    }

    $A = \int_{t_d}^{t_d+i} f(t)\ dt$
    s = floor(A)

    IF ($s > s_{max}$) {
        Sende s div s_{max} Pakete der Grösse s_{max}
        $s = s$ mod s_{max}
        IF ($s \geq s_{min}$) Sende ein Paket der Grösse s
    }
    ELSE Sende ein Paket der Grösse s

    $t_w = t_d + i$ - (jetzt() - t_s
    Schlafe während t_w
}

ParseLog

Aus den Diagnosemeldungen von DSPhone muss der Verlauf eines oder mehrerer Parameter extrahiert werden können, um graphische Darstellungen zu erstellen. Durch das Perl-Skript `parselog` wird diese Aufgabe automatisiert. Der Aufruf lautet

parselog [-h] [-d] [-e] *Feld* [*Feld2*...]

Das Programm liest die DSPhone-Ausgabe über `stdin` und extrahiert die angegebenen Felder, die aus den folgenden ausgewählt werden können: loss_ratio, total_loss, jitter und dlsr aus RR-Empfangsmeldungen, Dloss_ratio, Droundtrip,

Dlast_roundtrip und Djitter aus Diagnosemeldungen, Eloss_ratio, Edelay und Ejitter aus Bewertungsmeldungen. Die gefundenen Werte werden in einem für Gnuplot verständlichen Format ausgegeben.
Mit der Option -h kann der Ausgabe ein Kopf mit allgemeinen Informationen über die Sitzung hinzugefügt werden. Werden die Optionen -d oder -e benutzt, beschränkt sich die Suche auf Diagnose- bzw. Bewertungszeilen (e steht für Evaluation).

4.1.3 LinuxDS

Die in den Tests benötigte DiffServ-Funktionalität der Router ist in der DiffServ-Variante der Universität Bern[5], im weiteren nur LinuxDS genannt, realisiert, deren Basis der um einige Module erweiterte Linux-Kern 2.2.17 bildet. Diese Module werden in einer dynamischen Baumstruktur kombiniert, deren Wurzel einem Netzwerk-Gerät zugeordnet ist und deren Blätter aus Queues bestehen. Ankommende Pakete traversieren den Baum bis zu einer Queue, wo sie zwischengespeichert werden. Soll ein Paket gesendet werden, entscheiden die Module des Baumes, aus welcher Queue dieses stammen soll. Da mehrere Module auf dieselben Datenfluss-Beschreibungen zugreifen, existiert ein weiteres Modul, `dstable`, das Tabellen solcher Beschreibungen verwaltet. Hier ein kurzer Überblick über die einzelnen Module:

serv_handler (Service Handler) Ordnet die ankommenden Pakete den in einer Tabelle aufgeführten Flussbeschreibung zu und ändert gegebenenfalls ihr DSCP-Feld, um sie dann an das nächste Modul weiterzuleiten. Dadurch können Pakete von nicht DS-fähigen Systemen einer Serviceklasse zugeordnet werden. Alternativ kann die Zuordnung von Paketen, die von einem DS-fähigen System kommen, verändert werden. Dieses Modul wird nur in Ingress-Knoten verwendet, wo es zwischen der Baumwurzel und dem DS-Classifier zu liegen kommt.

dsclsfr (DS-Classifier) Teilt ankommende Pakete abhängig von ihren DSCP-Feldern einem von sieben Unterbäumen zu. Beim Senden entscheidet dieses Modul mit einem speziellen Round-Robin-Algorithmus, aus welchem Unterbaum das Paket stammen soll. Assured-Service-Pakete werden den Unterbäumen 1–4, Premium-Service-Pakete dem Unterbaum 5 und Best-Effort-Pakete dem Unterbaum 7 zugeteilt. Pakete, deren DSCP-Felder einen der in [RFC 2474] für Kontrollverkehr reservierten Wert enthalten, werden den Unterbäumen 6 und 7 zugeteilt.

[5] [LinuxDS]

In inneren Knoten bildet dieses Modul die Wurzel des Baumes, in Ingress-Knoten bildet es das einzige Kind von Service Handler.

prec_handler (Precedence Handler) Teilt Paketen, die vorher vom Service Handler einer der vier Assured-Service-Klassen zugeordnet wurden, eine Dropping-Priorität zu und leitet sie an das nächste Modul weiter. Wie der Service Handler findet dieses Modul nur in Ingress-Knoten Verwendung, wo es sich an der Wurzel der Unterbäume 1–4 des DS-Classifiers befindet.

trio (Triple RIO Queue) Implementiert eine RED-Queue[6] mit drei verschiedenen Dropping-Prioritäten, wie sie in 2.4.5.3 beschrieben wird. In inneren Knoten bestehen die Unterbäume 1–4 des DS-Classifiers aus Instanzen dieses Moduls. In Ingress-Knoten werden sie je einem Precedence-Handler-Modul angehängt.

premium_shaper Bildet die Wurzel des fünften Unterbaumes des DS-Classifiers. Ordnet die ankommenden Pakete einer Flussbeschreibung zu und leitet sie an die entsprechenden Unterbäume weiter. Deren Anzahl entspricht den in der Tabelle enthaltenen Premium-Service-Flussbeschreibungen. In der umgekehrten Richtung werden die Unterbäume, die Gelegenheit zum Senden erhalten, in einem Round-Robin-Verfahren ausgewählt.
Der Name Premium Shaper ist etwas irreführend, da das eigentliche Shaping an weiter unten im Baum liegende Module delegiert wird. Diese Module sind bereits im Linux-Kern enthalten und sind kein Teil von LinuxDS.

premium_policer Eine Variante von Premium Shaper. Ankommende Pakete werden einer Flussbeschreibung zugeordnet und mit einem Token-Bucket-Algorithmus auf ihre Zulässigkeit geprüft. Schlägt dieser Test fehl, verwirft das Modul das betreffende Paket. Ansonsten wird es an den einzigen Unterbaum weitergeleitet.

Nebst dem Kern enthält das LinuxDS-Paket zwei Programme: `dstab`, mit dem die Flussbeschreibungs-Tabellen im dstable-Modul verwaltet werden können und eine an den LinuxDS-Kern angepasste Version des Programms `tc`, das die Baumstruktur der Module verändern kann. Beispiele für ihre Bedienung können den Tests entnommen werden.

4.2 Design von DSPhone

Das Design von DSPhone wird hier in folgender Form wiedergegeben: Einzelne Klassen und Klassenfamilien werden zuerst kurz in Prosa beschrieben und ge-

[6] [RED]

gebenenfalls motiviert, gefolgt von einer sogenannten Class-Responsibility-Collaborations-Karte (CRC-Karte), die die Verantwortlichkeiten der Klasse auflistet. Die genauen Interfaces und Zusammenhänge zwischen den Klassen stellen die Klassendiagramme dar, die jeweils ein ganzes Subsystem beinhalten[7]. In Fällen, wo weitere Formalismen nötig sind, um das Verständnis zu erleichtern (zum Beispiel wo eigene Protokolle definiert werden), wird diese Form entsprechend ergänzt. Die Reihenfolge der besprochenen Subsysteme folgt der natürlichen Hierarchie, mit Ausnahme des Abschnitts über die Hilfsklassen. Dieser wurde vorgezogen, um allzu häufiges Vorgreifen zu vermeiden.

4.2.1 Hilfsklassen

Manche Klassen sind von allgemeiner Verwendbarkeit und keinem Subsystem zuzuordnen. Einige davon kapseln C-Bibliotheken, andere stellen häufig gebrauchte Konzepte dar, wie zum Beispiel die Producer-Consumer-Klasse (ProdConBuf). Auf Seite 141 befindet sich das Klassendiagramm zu diesen Klassen.

4.2.1.1 Exceptions

Um auftretende Ausnahmezustände in einer allgemeinen Form behandeln zu können und gleichzeitig die Möglichkeit der Standard-C-Biblibothek auszunutzen, textuelle Fehlermeldungen zu generieren, wird eine spezielle Exception-Klasse verwendet.

Klasse: Exception	
Superklassen: -	
Subklassen: ConfigException, ParseException, SocketException, HostException, HostException, RTPPacketException, ProtoException, AudioException, Assertion, FileException, OutOfMemory, CondException, MutexException, ThreadException, LimitedMapException	
Enthält einen Fehlermeldungs-String der Form “Klasse: Fehler”.	`char *msg`
Erfüllt Orthodox Canonical Form.	
Der String kann im printf- und perror-Stil gesetzt werden.	`set_string(...)`
Alternativ kann ein Standardtext verwendet werden.	`set_string(void)`

Die Subklassen von Exception fügen meist keine weitere Funktionalität hinzu. Sie existieren nur zum Zweck der Unterscheidung in `catch()`-Statements.
Um das “Werfen” und Definieren von Exceptions vereinfachen zu können, gibt es die Template-Funktion `do_throw`, sowie die Makros `NEW_EXCEPTION` und `NEW_EXCEPTION_WITH_TEXT`. Letzteres definiert zusätzlich einen neuen Standardtext für die Subklasse.

[7] In einigen Fällen wurden die Diagramme zugunsten der Lesbarkeit aufgeteilt.

4.2.1.2 Config

In grossen Programmen mit mehreren Subsystemen kann es zum Problem werden, die Konfiguration der einzelnen Teile einfach und konsistent zu halten. Eine mögliche Lösung ist das Verwalten einer zentralen Registratur, in der die Konfigurationsvariablen unter einem bestimmten Namen abgelegt werden. Die Config-Klasse implementiert diese Strategie. Das Interface dieser Klasse wurde statisch gehalten. Die einzelnen Konfigurationswerte sind typenlos (intern werden sie als void-Zeiger dargestellt, der aber auch als Integer interpretiert werden kann). Stellt der Wert einen Zeiger dar, kann der Speicherbereich, auf den er zeigt, am Lebensende der Variable automatisch optional freigegeben werden[8]. Das erweist sich beispielsweise dann als nützlich, wenn Double-Zahlen abgelegt werden sollen.

Klasse: Config (statisch)	
Superklassen: -	
Subklassen: -	
Hält eine Liste von Variablen.	Item (intern)
Synchronisiert den Zugriff auf diese Variablen.	Mutex
Prüft für einen Schlüssel, ob er bekannt ist.	`exists(key)`
Gibt den einem Schlüssel zugeordneten Wert zurück.	`void* get_item(key)`
Schlüssel/Wert-Paare können hinzugefügt und entfernt werden.	`add_item(key, value)` `rem_item(key)`
Ein Wert kann verändert werden. Existiert der Schlüssel noch nicht, wird er hinzugefügt.	`set_item(key,value)`
Kann eine beliebige Funktion auf alle Variablen anwenden.	`foreach(func,arg)`
Kann alle gespeicherten Variablen löschen.	`clear()`
Kann optional den Speicherbereich einer Variable freigeben.	
Gibt eine Liste der gespeicherten Schlüssel zurück.	`get_keys()`

Konfigurationen können eingelesen und abgespeichert werden, indem die globalen Funktionen `read_rcfile()` und `write_rcfile()` aufgerufen werden. Die Syntax der einzelnen Konfigurationsvariablen ist in einer Tabelle abgelegt, in der 4-Tupel der Form (Schlüssel, Lesefunktion, Schreibfunktion, Validierungsfunktion) enthalten sind. Die Lese- und Schreibfunktionen bestimmen die Art, wie eine Variable gelesen und geschrieben wird; mit der Validierungsfunktion kann der eingelesene oder während der Laufzeit geänderte Wert auf seine Gültigkeit überprüft werden.
Dieser Teil des Konfigurationssystems ist offensichtlich nicht objektorientiert, funktioniert jedoch stabil. Sollte das System aber erweitert werden, ist er ein eindeutiger Kandidat für ein Redesign.

[8] Es gilt hier zu beachten, dass der Destruktor eines abgelegten Objekts *nicht* aufgerufen wird, da sein Typ nicht bekannt ist. Für Objekte statischer Grösse ist dies jedoch kein Problem.

4.2.1.3 Time

Die Zeitdarstellung der C-Bibliothek stellt die absolute Zeit mittels zweier 32Bit-Integer dar. Der eine zählt die Anzahl Sekunden seit dem 1.1.1970 00:00, der andere die Mikrosekunden in der aktuellen Sekunde. Hinzu kommt eine alternative Darstellung, die anstatt Mikro- Nanosekunden zählt.
Diese Darstellung macht Berechnungen unnötig komplex, weshalb diese am besten hinter einem klassenorientierten Interface versteckt werden. Die Time-Klasse stellt die Zeit intern als `double` dar und bietet alle üblichen Zeitoperationen und Konvertierungsmethoden, unter anderem auch für die NTP-Timestamps von RTP.

Klasse: Time	
Superklassen: -	
Subklassen: -	
Hat Konstruktoren für mehrere Zeitformate.	`Time(...)`
Default-Konstruktor setzt aktuelle Zeit.	`Time()`
Erfüllt Orthodox Canonical Form.	
Kann nachträglich eine andere Zeit übernehmen.	`set_to(...)`
Kann die aktuelle Zeit übernehmen.	`set()`
Kann als `double` verwendet werden.	`operator double()`
Konvertiert sich in verschiedene Formate.	`get_*`
Unterstützt relationale und arithmetische Operatoren.	`operator X`
Wirft eine Exception bei Fehlbedienung.	Assertion
Wirft eine Exception, wenn Fehler entdeckt werden.	Exception

4.2.1.4 Threads

Unter Linux werden Threads durch die pthread-Bibliothek unterstützt. Leider ist diese eine reine C-Bibliothek, bietet also keine aktiven Objekte oder verwandte Konzepte. Um die Verwendung dieser Bibliothek zu vereinfachen, mussten also einige Wrapper-Klassen geschrieben werden. Die drei Hauptkonzepte Thread, Bedingungsvariable (Condition Variable) und Mutex werden durch die Klassen MThread, Cond und Mutex gekapselt.

MThread

Die MThread-Klasse ist ein Template, da für jede Threads verwendende Klasse eine spezialisierte Thread-Klasse generiert werden muss.

Klasse: MThread (Template)	
Superklassen: -	
Subklassen: -	
Kennt sein designiertes Objekt und dessen Runner-Methode.	
Kann gestartet werden.	`start()`
Ein gestarteter Thread kann gestoppt werden.	`stop()`
Alternativ kann ein Thread ihm “joinen”.	`join()`
Kennt seine pthread-Repräsentation.	`thread_id`
Wirft eine Exception, wenn Fehler entdeckt werden.	ThreadException

Mutex

Um Threads zu synchronisieren, werden hauptsächlich Instanzen von Mutex verwendet.

Klasse: Mutex	
Superklassen: -	
Subklassen: -	
Kann gesperrt und freigegeben werden.	`lock(), unlock()`
Kennt seine pthread-Repräsentation.	`mutex`
Wirft eine Exception, wenn Fehler entdeckt werden.	MutexException

Cond

Soll eine komplexere Synchronisationsstrategie verwendet werden, kommt das Konzept der Bedingungsvariable hinzu, gekapselt durch die Klasse Cond. Es ist wichtig zu bemerken, dass diese Klasse keine Semaphore darstellt, jedoch zusammen mit der Mutex-Klasse als Baustein einer solchen benutzt werden kann.

Klasse: Cond	
Superklassen: -	
Subklassen: -	
Auf die Bedingungserfüllung kann gewartet werden.	`wait()`
Es kann bis zu einem gewissen Zeitpunkt	`timedwait(), Time`
oder für eine gewisse Dauer gewartet werden.	`timedwait_delta()`
Veränderungen der Bedingungs-Parameter können einem	`signal()`
oder allen wartenden Threads gemeldet werden.	`broadcast()`
Kennt seine pthread-Repräsentation.	`cond`
Wirft eine Exception, wenn Fehler entdeckt werden.	CondException

4.2.1.5 Netzwerk

Ähnlich wie die Thread-API ist auch diejenige der Netzwerkfunktionalität eine reine C-Bibliothek. Es gibt verschiedene Klassenbibliotheken, die ein klassenorientiertes Interface zu dieser API implementieren. Leider schien sich keine der

betrachteten Bibliotheken dazu zu eignen, sie mit DiffServ-Funktionalität zu erweitern, weshalb ich mich dazu entschlossen habe, ein eigenes Toolkit zu schreiben. Darin wurden verschiedene Konzepte der POSIX-Netzwerk-API gekapselt. Das dazugehörige Klassendiagramm befindet sich auf Seite 142.

DSType

Die Verwaltung der in DiffServ verwendeten DSCP-Werte ist kryptisch und eignet sich daher nicht dafür, direkt im Code getätigt zu werden. Die DSType-Klasse versteckt die nötigen Bitmanipulationen hinter einem für menschliche Augen freundlicheren Interface. Den verschiedenen DiffServ-Klassen werden in der Enumeration `DSClass` Symbole zugeordnet, nämlich `ASSURED` für die Assured-Forwarding-Klassen, `PREMIUM` für Expedited Forwarding, `NONE` für Best Effort und das spezielle Symbol `IGNORE`. Letzteres bewirkt, dass beim Senden eines Pakets die Klasse des zuletzt geschickten Pakets übernommen wird.

Klasse: DSType	
Superklassen: -	
Subklassen: -	
Kennt seine DSClass und 0-2 dazugehörige Argumente. Diese haben je nach Klasse verschiedene Bedeutung.	`ds_class, arg1, arg2`
Der Default-Konstruktor setzt die Klasse auf IGNORE.	`DSType()`

Gibt seine DSClass und, falls möglich,
die Premium-Service-Datenrate oder
die Assured-Service-Stufe und
-Dropping-Priorität zurück.
	`get_class()`
	`get_ps_rate()`
	`get_as_level()`
	`get_as_dprec()`

Kann sich selbst in ein 16Bit-Wort kodieren und ein solches dekodieren.	`_u16 encode()` `DSType(_u16)`
Wirft eine Exception, wenn Fehler entdeckt werden.	Exception

Folgende Tabelle zeigt die Bedeutung der beiden Argumente:

DSClass	**Argument 1**	**Argument 2**
`IGNORE`	-	-
`NONE`	-	-
`ASSURED`	Level (1–4)	Dropping Priorität (0–3)
`PREMIUM`	Datenrate (1–?)	-

Host

Um eine Internetadresse für ein Zielgerät zu generieren, muss erst sein Name in eine numerische Adresse konvertiert und in Network-Byte-Order umgewandelt werden. Bei UDP- und TCP-Adressen muss man auch noch die Portnummer umwandeln. Diese Aufgaben übernimmt die Host-Klasse:

Klasse: Host	
Superklassen: -	
Subklassen: -	
Hat Konstruktor für “localhost” (optional mit Port).	`Host(port=0)`
Hat Konstruktor für Hostnamen (opt. mit Port).	`Host(name, port=0)`
Hat Konstruktor für numerische Adressen (opt. mit Port).	`Host(address, port=0)`
Hat Konstruktor für C-Adressen (opt. mit Port).	`Host(naddr, port=0)`
Erlaubt Spezialfunktionen Zugriff auf interne Repräsentation.	`addr`
Kann sich klonen.	`clone()`
Gibt den Hostnamen zurück.	`get_name()`
Port kann nachträglich gesetzt werden.	`set_port(port)`
Gibt gesetzten Port (in Host-Byte-Order) zurück.	`get_port()`
Wirft eine Exception, wenn Fehler entdeckt werden.	HostException

Sockets

Ein Prozess greift auf die Netzwerk-Funktionen des Betriebssystems durch sogenannte Sockets zu, die sich verschieden verhalten können, je nachdem in welchem Modus sie sich befinden. So können Sockets unter anderem so konfiguriert werden, dass sie UDP oder TCP benutzen. Da sich aber verschieden konfigurierte Sockets gegen aussen auch verschieden verhalten, bietet es sich an, ein einheitliches Interface zu definieren. Die modus-spezifischen Funktionen können dann in spezialisierten Unterklassen implementiert werden. Die abstrakte Socket-Klasse implementiert die allgemeine Funktionalität von Sockets, ihre Erben (TCPSocket und UDPSocket) die Eigenheiten der jeweiligen Modi.

Klasse: Socket (abstrakt)	
Superklassen: -	
Subklassen: TCPSocket, UDPSocket	
Kennt den Verbindungsstatus und macht ihn zugänglich.	`connected`
Unterstützt nicht-blockierende Transaktionen.	
Kennt die Adressen beider Seiten der Verbindung.	Host
Deklariert Sende- und Empfang-Methoden.	`send(), recv()`
Gibt lokale Adresse zurück.	`get_local()`
Gibt entfernte Adresse zurück.	`get_peer()`
Kann die gesendeten Pakete mit einem DSType markieren.	
Übersetzt Systemfehler in Exceptions.	SocketException

TCPSocket bietet ein Interface, das zwar einfach ist, aber für die meisten Anwendungen genügen dürfte. Einige Spezialfunktionen, wie das Binden auf ein spezifisches Netzwerk-Interface, wurden weggelassen, da sie in den seltensten Fällen benötigt werden.

Klasse: TCPSocket	
Superklassen: Socket	
Subklassen: -	
Unterstützt Client- und Servermodus.	
Hat Konstruktor für Servermodus.	`TCPSocket(localport)`
Hat Konstruktor für Clientmodus.	`TCPSocket(peer)`
Baut eine Verbindung auf (Client).	`connect()`
Wartet auf eine Verbindung (Server).	`accept()`
Terminiert eine bestehende Verbindung (beide).	`shutdown()`
Implementiert Senden und Empfangen.	`send(), recv()`
Übersetzt Systemfehler in Exceptions.	SocketException

UDPSocket nutzt nur einen Teil der Möglichkeiten, die das Betriebssystem bietet. Beispielsweise erwartet es feste Ports auf beiden Seiten und kann das Ziel der Pakete nicht dynamisch ändern. Das verringert die Komplexität der Klasse, aber auch deren Wiederverwendbarkeit.

Klasse: UDPSocket	
Superklassen: Socket	
Subklassen: -	
Legt lokalen Port und Peer fest.	`UDPSocket(localport, peer)`
Implementiert Senden und Empfangen.	`send(), recv()`
Übersetzt Systemfehler in Exceptions.	SocketException

Komplexe Netzwerk-Programme müssen häufig mehrere Sockets gleichzeitig verwalten. Das kann dazu führen, dass der Prozess darauf wartet, dass auf einem Socket Daten ankommen, währenddem auf einem anderen Daten darauf warten, gelesen zu werden. Eine Lösung ist, für jedes Socket einen Thread zu kreieren, der auf Daten wartet, eine andere, die Sockets nicht-blockierend abzufragen. Effizienter ist es jedoch, die `select`-Funktion des Betriebssystems zu benutzen, die eine Liste von Sockets annimmt und zurückgibt, welche dieser Sockets bereit zum Empfangen (oder Senden) sind. Aus diesem Grund gibt es zusätzlich zu den Socket-Klassen eine `select_read`-Funktion, die eine Liste von Socket-Instanzen annimmt und wiederum die Systemfunktion (mit entsprechenden Argumenten) aufruft. Ihre Signatur ist

_u16 select_read(unsigned n, struct timeval *timeout, ...)

n bezeichnet dabei die Anzahl Sockets in der Liste (höchstens 16), und mittels $timeout$ kann die Wartezeit beschränkt werden. Die zu überwachenden Sockets müssen als Socket* übergeben werden. Der Rückgabewert der Funktion ist eine 16Bit-Maske, in der für jedes aktive Socket ein Bit gesetzt ist (für das erste Socket Bit 0, für das zweite Bit 1, etc).

4.2.1.6 Source und Sink

Die Interfaces Source und Sink definieren ein einheitliches Interface für Klassen, die einen Datenstrom erzeugen, verbrauchen oder durch sich durchfliessen lassen. Zu diesem Zweck enthalten sie lediglich eine `read`- und eine `write`-Methode, ergänzt durch die Varianten `read_all` und `write_all`, die sicherstellen, dass alles gelesen/geschrieben wurde, bevor die Methode zurückkehrt.

Klasse: Source (abstrakt)	
Superklassen: -	
Subklassen: -	
Definiert eine Methode zum Lesen von Daten aus dem Strom. Diese kann wahlweise blockierend oder nicht-blockierend sein.	`read()`
Definiert eine Lese-Methode mit garantiert blockierendem Verhalten.	`read_all()`

Klasse: Sink (abstrakt)	
Superklassen: -	
Subklassen: -	
Definiert eine Methode zum Schreiben von Daten in den Strom. Diese kann wahlweise blockierend oder nicht-blockierend sein.	`write()`
Definiert eine Schreib-Methode mit garantiert blockierendem Verhalten.	`write_all()`

4.2.1.7 Container

Bei der Implementierung des Programms haben sich zwei container-artige Konzepte herauskristallisiert, die genügend allgemeine Verwendbarkeit aufweisen, um sie als Hilfsklassen zu implementieren. Daraus sind die Klassen LimitedMap und ProdConBuf entstanden.

LimitedMap

In einigen Fällen ist es notwendig, zusätzlich zu den Schlüssel-Wert-Assoziationen einer Map die “Aktualität” der einzelnen Assoziationen zu kennen. Wenn eine Information ein gewisses Alter überschreitet, wird sie nutzlos. Man könnte dies mit dem menschlichen Kurzzeitgedächtnis vergleichen. Alte Informationen werden vergessen. Die LimitedMap-Templateklasse ist eine solche spezialisierte Map.

Klasse: LimitedMap (Template)	
Superklassen: -	
Subklassen: -	
Templateparameter sind `Key` und `Val`.	
Merkt sich eine begrenzte Menge von (Key,Val)-Paaren.	
Hat eine maximale Anzahl solcher Paare.	
Sucht nach Schlüsseln.	`contains(key)`
Gibt den einem Schlüssel zugeordeten Wert zurück.	`get(key)`
Fügt neue (Key,Val)-Paare hinzu, "vergisst" aber das älteste Paar, wenn die maximale Anzahl erreicht ist.	`put(key,val)`
Wirft eine Exception, wenn Fehler entdeckt werden.	LimitedMapException

ProdConBuf

Das Backend von DSPhone besteht zum grossen Teil aus aktiven Objekten, weshalb das als Producer-Consumer bekannte Synchronisationsproblem des Öfteren auftritt. Es liegt also nahe, diese Funktionalität in einer Hilfsklasse zu kapseln. Ihr Name ist ProdConBuf, was für Producer-Consumer-Buffer steht. Der Zugriff wird für die Backend-Klassen transparent gehalten, indem ProdConBuf die auch zum Backend gehörenden Interfaces Source und Sink erbt.

Klasse: ProdConBuf	
Superklassen: Source, Sink	
Subklassen: -	
Hat einen Puffer von Daten und synchronisiert den Zugriff darauf.	Mutex, Cond
Bietet lesenden und schreibenden Zugriff.	`read(), write()`
Gibt die Anzahl Bytes zurück, die momentan im Puffer sind.	`available()`
`read()` blockiert, bis genug Daten vorhanden sind.	
`write()` blockiert, bis Platz zum Schreiben der Daten vorhanden ist.	

4.2.2 Basissystem und Backend

Das *Basissystem* übernimmt die Initialisierung beim Programmstart, parst gegebenenfalls Kommandozeilenargumente und liest die Konfigurationsdatei. Es kommuniziert mit dem Benutzer und ist für die Signalisierung zwischen den Endpunkten verantwortlich. Nach der Signalisierungsphase eines Anrufs initialisiert es das Backend und übergibt ihm die Kontrolle.
Das *Backend* wiederum kreiert das Audio-Subsystem und erstellt das vom Benutzer konfigurierte Protokoll-Subsystem (meist das RTP-Subsystem), um danach als Schnittstelle zwischen den beiden zu fungieren. Dies macht es solange, bis entweder der Benutzer oder der gegenüberliegende Endpunkt den Anruf beendet.
Es folgt eine genauere Betrachtung der wichtigen Abläufe.

4.2.2.1 Programmstart

Nachdem das Programm aufgerufen wurde, installiert es als erstes eine neue Fehlerbehandlungsroutine für Speicherallokationsfehler, was im übrigen Code das ständige Überprüfen der Rückgabewerte von `new` unnötig macht. Als nächstes werden Standardwerte in die Konfigurationstabelle geschrieben und der Name der zu lesenden Konfigurationsdatei auf "$HOME/.dsphonerc" gesetzt. Dieser kann beim nun folgenden Parsen der Kommandozeilenargumente wieder geändert werden.
Es folgt das Einlesen der Konfigurationsdatei. Für in der Datei nicht erwähnte Variablen wird der oben gesetzte Standardwert übernommen, alle anderen erhalten den neuen Wert. Falls die Datei nicht existiert, gelten verschiedene Verhaltensregeln, abhängig davon, ob diese explizit angegeben wurde. Ist dies der Fall, wird ein Fehler ausgegeben und das Programm beendet. Andernfalls behalten alle Variablen ihre Standardwerte, und der Fehler wird ignoriert. Nachdem nun noch die Art der angezeigten Diagnosemeldungen konfiguriert wurde, startet das Benutzer-Interface.

Von hier weg unterscheidet sich der Ablauf zwischen den GUI- und kommandozeilenorientierten Versionen von DSPhone. Letztere fährt sofort mit dem Aufbau einer Verbindung weiter, während dem Benutzer im anderen Fall verschiedene Optionen geboten werden.

4.2.2.2 Einleiten und Abbruch einer Verbindung

Sobald eine Verbindung aufgebaut werden soll, wird ein Initiator-Objekt im Server- oder Client-Modus instantiiert. Die Benutzer-Interfaces unterscheiden sich dabei nur darin, welche Art von Log-Objekt sie dem Konstruktor übergeben. Dieses kann vom statischen Typ TextLog oder LogWindow sein.
Das Initiator-Objekt öffnet ein TCPSocket für die Signalisierung und startet den objektinternen Thread, den *Starterthread*. Danach kehrt der Hauptthread aus dem Konstruktor zurück und wartet auf weitere Eingaben des Benutzers. Der Starterthread hingegen startet die Client- oder Serverseite des (zeilenorientierten) Signalisierungsprotokolls, das folgendermassen abläuft (siehe auch Abb. 4.13):

Der Anruf beginnt damit, dass der Anrufer eine TCP-Verbindung zum gewünschten Server öffnet. Dieser sendet darauf den Begrüssungsstring "DSPhone" und baut intern das restliche Backend-System, und damit auch die Audio- und Protokoll-Subsysteme, auf. Das fertige Backend generiert und übermittelt einen String, der beschreibt, in welcher Weise das Backend des Anrufers konfiguriert werden soll. Wenn dieser die Konfiguration übernommen und sein Backend entsprechend initialisiert hat, antwortet er mit "confirmed". Kann er die Anforderun-

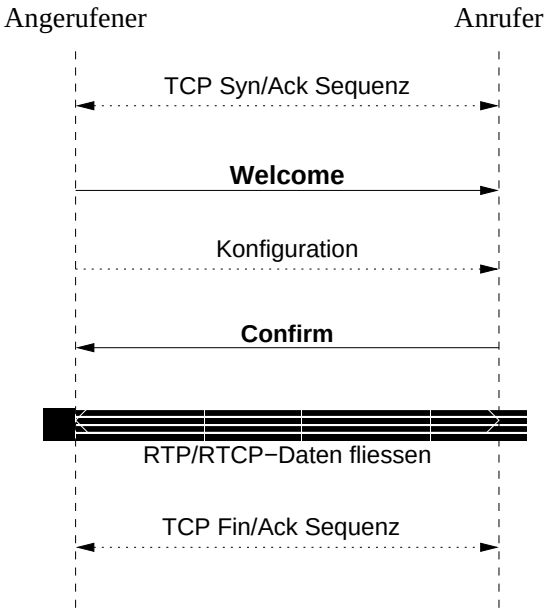


Abbildung 4.13: Anrufaufbau in DSPhone

gen nicht erfüllen, sendet er stattdessen “unable”, worauf beide Seiten die Verbindung schliessen und der Anruf abgebrochen ist.
Damit ist der Aufbau abgeschlossen und die Daten fliessen über das ausgewählte Medientransportprotokoll. Die Beendigung der Verbindung ist abhängig vom Medientransportprotokoll — im Fall von RTP geschieht sie durch ein BYE-Paket einer der Parteien.

Nach Abschluss der Aufbauphase wartet der Starterthread darauf, dass der Medientransport abgeschlossen wird. Wird er geweckt, löscht er das Backend und beendet sich. Das Backend bricht die Verbindung aufgrund zweier verschiedener Ereignisse ab: einerseits kann ihm der Hauptthread nach entsprechenden Benutzereingaben signalisieren, die Verbindung baldmöglichst abzubrechen, andererseits kann die Abbruchsaufforderung auch vom anderen Endpunkt kommen.

Aufbau des Backends

In obiger Beschreibung wurden die Details der Initialisierung des Backends der Übersichtlichkeit wegen weggelassen. Dieser Abschnitt geht nun näher auf diese Details ein. Die folgenden Erläuterung werden in Abb. 4.14 verdeutlicht.

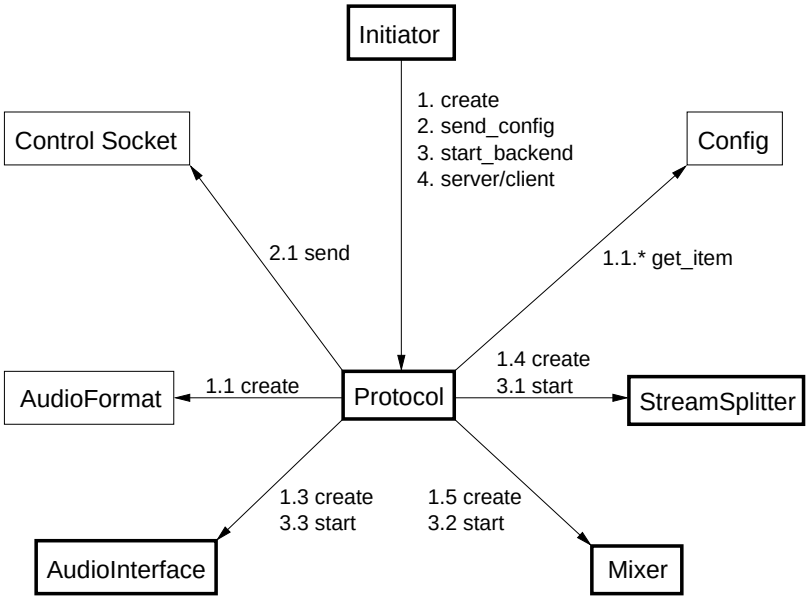


Abbildung 4.14: Aufbau des Backends

Der Aufbau beginnt damit, dass die Factory-Methode des Initiators eine Protocol-Instanz von statischen Typ RTP oder SimpleTCP kreiert. Im Protocol-Konstruktor wird die Konfiguration ausgelesen und darauf nacheinander das Backend-Audioformat, das Audio-Subsystem, der StreamSplitter und der Mixer kreiert. Ist das Backend nicht im Duplex-Modus, fällt bei dieser Aufzählung der StreamSplitter weg. Im Send-Modus fällt zusätzlich der Mixer weg.
Im nächsten Schritt ruft Initiator die `send_config`-Methode von Protocol auf, worauf der entsprechende String erstellt und über das Control Socket gesendet wird.
Wenn das geschehen ist und eine Bestätigung von der anderen Partei empfangen wurde, lässt Initiator die Backend-, Audio- und Protokollthreads starten.

4.2.2.3 Struktur des Backends

Das Backend dient als Schnittstelle zwischen dem Audio- und dem Protokoll-Subsystem und bearbeitet die zwischen beiden Seiten fliessenden Daten. Dazu gehört, je nach Modus, das Konvertieren, Mischen und Umleiten der Audiodaten. Es kann zwischen den Modi *Send*, *Receive*, *Simple Duplex* und *Duplex* gewählt werden. Send und Receive sind dann von Nutzen, wenn die Daten entweder nur gesendet oder nur empfangen werden sollen. Duplex und Simple Duplex dienen zum gleichzeitigen Empfangen und Senden von Daten, wobei die Datenströme in beiden Richtungen bei Simple Duplex unabhängig voneinander fliessen und bei

Duplex gemischt werden, was dazu dient, dem Sprecher durch ein lokales Echo den Eindruck einer “lebendigen” Leitung zu vermitteln.
Die Schnittstellen zum Audio-Subsystem sind die Klassen Mixer und Stream-Splitter. Diejenige zum Protokoll-Subsystem ist etwas komplizierterer Natur: Alle Protokoll-Subsysteme enthalten einen Erben der abstrakten Protocol-Klasse, die aber konzeptuell zum Backend gehört. Die Schnittstelle liegt also zwischen Protocol und ihren Erben. Dieses Design ist etwas problematisch, weil dadurch die Kapselung der Protokoll-Subsysteme durchbrochen wird.
Wie die einzelnen Objekte interagieren, wird in Abb. 4.15 graphisch dargestellt. Die Numerierung der Nachrichten ist hier etwas speziell: Vor jeder Nachricht steht ein Buchstabe, der das aktive Objekt identifiziert, von dessen Thread sie gesendet wird. Alle Threads befinden sich ausserdem in einer Endlosschleife, die erst beim Beenden der Verbindung unterbrochen wird. Abb. 4.2 zeigt ausserdem den Verlauf der Datenströme im Send-, Receive-, Simple-Duplex- und Duplex-Modus.

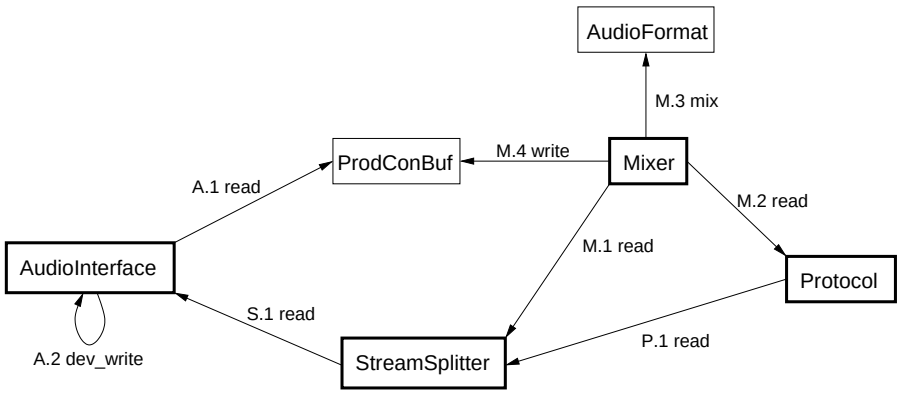


Abbildung 4.15: Abläufe während einer Duplex-Verbindung

Im Folgenden werden nun die Klassen des Backends im Einzelnen vorgestellt. Das dazugehörige Klassendiagramm befindet sich auf Seite 143.

Log

Da es verschiedene Benutzer-Interfaces geben kann, muss die Ausgabe von Statusmeldungen gekapselt werden. Dies geschieht durch die abstrakte Log-Klasse, die den Backend-Klassen ein einheitliches Interface bietet, über das sie ihre Meldungen ausgeben können. Die Enumeration LogPrio ermöglicht es ihnen ausserdem, Meldungen nach Inhalt und Wichtigkeit zu unterscheiden.

Klasse: Log (abstrakt)	
Superklassen: -	
Subklassen: LogWindow, TextLog	
Kann Meldungen verschiedener Prioritätausgeben.	`print(prio,fmt,...)`
Hat eine Maske, die bestimmt, welche Meldungen ausgegeben werden.	
Diese Maske wird der Konfiguration entnommen.	Config
Wirft bei Fehlern Exceptions.	Exception

Initiator

Initiator bildet die Schnittstelle zum Basissystem und ist die zentrale Klasse des Backends.

Klasse: Initiator	
Superklassen: -	
Subklassen: -	
Hat eine Instanz von Protocol.	Protocol
Hat eine Verbindung zum anderen Endpunkt.	TCPSocket
Gibt Statusmeldungen über ein Log-Objekt aus.	Log
Instanzen von Initiator sind aktive Objekte.	MThread
Kann im Server- oder Client-Modus sein.	`server_mode`
Der Modus wird mit dem entsprechenden Konstruktor gewählt.	`Initiator(log,port)` `Initiator(log,host,port)`
Kann eine laufende Verbindung abbrechen.	`end_transmission()`
Führt die Anrufs-Signalisierung durch.	`proto_server()` `proto_client()`
Erstellt restliches Protocol über Factorymethoden, eine für lokale Konfiguration und eine für Konfigurationsstrings vom Server.	
Wirft bei Fehlern Exceptions.	Exception

Protocol

Die Protocol-Klasse steuert auf Anweisung von Initiator den Auf- und Abbau der restlichen Backend-Klassen und des Audio-Subsystems und dient gleichzeitig (per Vererbung) als Schnittstelle zum Protokoll-Subsystem. Ausserdem synchronisiert sie bei stehender Verbindung Benutzer-Interface und Backend. Da Protocol das Backend je nach Modus verschieden initialisiert, werden diese Modi als Enumeration ProtoMode dargestellt.
Protokoll-Subsysteme enthalten eine von Protocol erbende Klasse; SimpleTCP, ein aus einer einzigen Klasse bestehendes Protokoll-Subsystem, dient als einfaches Beispiel dafür. Da zwischen den Erben von Protocol und StreamSplitter und Mixer Daten fliessen, erbt die Klasse ausserdem von Source und Sink.

Klasse: Protocol (abstrakt)	
Superklassen: Source, Sink	
Subklassen: RTP, SimpleTCP	
Kann im Send-, Receive-, Simple Duplex- oder Duplex-Modus sein.	
Erstellt und übermittelt den Konfigurationsstring an die andere Partei.	TCPSocket
Gibt Meldungen über eine Log-Instanz aus.	Log
Hat das vom Backend verwendete Audioformat.	Audioformat
Hat ein Interface zum Audio-Subsytem.	AudioIface
Hat die restlichen Backend-Klassen.	StreamSplitter, Mixer
Synchronisiert Benutzerinterface und Backend.	Mutex, Cond
Dient als Quelle und/oder Ziel der Datenströme im Backend.	
Kann eine stehende Verbindung abbrechen.	
Kann im Server- oder Client-Modus betrieben werden.	
Wirft bei Fehlern Exceptions.	ProtoException

StreamSplitter

Im Duplexmodus müssen die vom Audiointerface kommenden Daten an den Mixer und ans Protokoll-Subsystem weitergeleitet werden, ohne dass dabei Daten "vergessen" gehen. Die StreamSplitter-Klasse implementiert also einen Zwischenspeicher, der die Audiodaten aufnimmt und eventuelle Geschwindigkeitsunterschiede zwischen den beiden ausgleicht.

Klasse: StreamSplitter	
Superklassen: -	
Subklassen: -	
Legt Daten in einem Zwischenspeicher ab.	
Hat einen Thread, der neue Daten liest.	Source, MThread
Dieser Thread kann gestartet und angehalten werden.	
Hält sich eine Liste von Klienten.	Client (intern)
Klienten können hinzugefügt werden.	`add_client()`
Klienten können entfernt werden.	`remove_client()`
Klienten können vom Zwischenspeicher lesen.	`read()`
Merkt sich für jeden Klienten, wieviele Daten er gelesen hat.	
Von allen Klienten gelesene Daten gelten als frei und können mit neuen Daten überschrieben werden.	
Leseoperationen blockieren, bis sie erfüllt werden können.	
Gibt Meldungen über eine Log-Instanz aus.	Log
Wirft bei Fehlern Exceptions.	Exception

Mixer

Bevor die Audiodaten dem Audio-Subsystem übergeben werden, fliessen sie durch den Mixer. Nur im Send-Modus, in dem keine Daten ausgegeben werden,

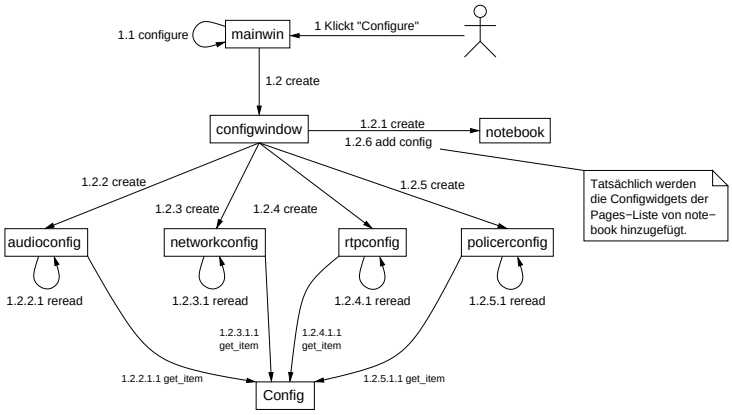
fällt er weg. Im Receive- und Simple-Duplex-Modus ist seine alleinige Aufgabe, Daten vom Protokoll- zum Audio-Subsystem zu verschieben. Der Name der Klasse kommt aber von ihrer Funktion im Duplex-Betrieb, wo die Audiodaten von StreamSplitter und vom Protokoll-Subsystem gemischt werden und erst dann dem Audio-Subsystem zugeführt werden.

Klasse: Mixer	
Superklassen: -	
Subklassen: -	
Kann im Misch- oder Weiterleitmodus sein.	
Kennt Quelle und Ziel der Daten, im Mischmodus ausserdem einen StreamSplitter.	Source, Sink StreamSplitter
Kennt das Format der Audiodaten.	AudioFormat
Weiterleitmodus: Leitet Daten von der Quelle zum Ziel weiter.	
Mischmodus: Liest Daten von Quelle und StreamSplitter, mischt sie und leitet sie zum Ziel weiter. Sind von der Quelle nicht genug Daten erhältlich, werden sie durch Nulldaten ersetzt.	
Hat einen internen Thread, der diese Aufgaben übernimmt.	MThread
Dieser Thread kann gestartet und angehalten werden.	
Gibt Meldungen über eine Log-Instanz aus.	Log
Wirft bei Fehlern Exceptions.	Exception

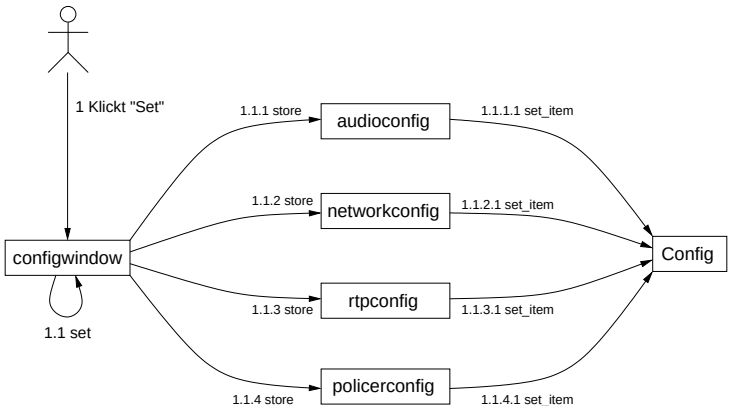
4.2.2.4 GUI

Die graphische Benutzerschnittstelle von DSPhone setzt auf der Klassen-Bibliothek Gtk-- auf, einer C++-Variante der C-Bibliothek Gtk+. Da sie nur einige neue Widgets definiert und der Bibliothek ansonsten keine neue Funktionalität hinzufügt, genügt hier das Klassendiagramm (Abb. 4.25) und die Ablaufdiagramme in Abb. 4.16, um das Design zu verdeutlichen. Eine Ausnahme bildet die LogWindow-Klasse, da sie die Schnittstelle zum Backend darstellt.

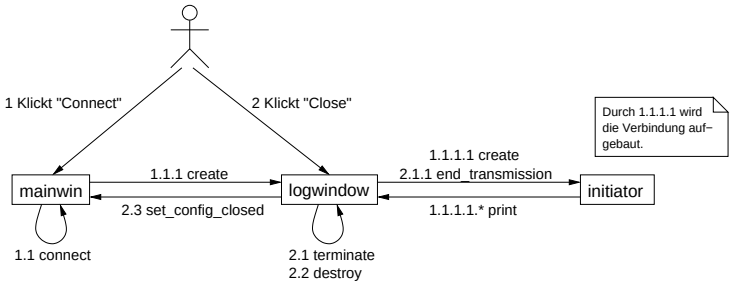
LogWindow realisiert das Interface von Log und hat zudem die Aufgabe, den Verbindungsaufbau einzuleiten. In der folgenden Beschreibung wurden die Details der GUI-Komponenten ausgelassen, um die Übersichtlichkeit zu erhalten.



(a) Konfigurationsfenster: Aufbau



(b) Konfigurationsfenster: Werte setzen



(c) Verbindungsaufbau (Client-Modus, Server-Modus funktioniert analog)

Abbildung 4.16: Abläufe des GUI

Klasse: LogWindow	
Superklassen: Log, Gtk::Window	
Subklassen: -	
Hat ein Textfeld, um Meldungen auszugeben.	Gtk::Text
Akzeptiert Meldungen von anderen Objekten.	`print()`
Synchronisiert den Zugriff.	Mutex, Cond
Gibt die Meldungen im Idle-Zyklus der GUI aus.	
Startet das Backend im Konstruktor.	Initiator
Übergibt sich dem Backend selbst als Log-Instanz.	
Hat einen “Close”-Button, mit dem eine Verbindung abgebrochen und das Fenster geschlossen wird.	
Zerstört das Backend, wenn die Verbindung abgebrochen werden soll.	Initiator
Signalisiert dem MainWin, wenn das Fenster geschlossen wird.	`log_closed()`

4.2.3 Audio-Subsystem

Die Audiofunktionalität von DSPhone lässt sich in zwei Gebiete unterteilen. Das eine beschäftigt sich mit dem Format von Audiodaten und davon abhängigen Operationen, wie etwa dem Konvertieren oder Mischen, das andere mit der Ein- und Ausgabe dieser Daten über Soundkarten oder logische Geräte. Das Klassendiagramm dieses Subsystems befindet sich auf Seite 145.

4.2.3.1 Formate

Obwohl DSPhone zum Zeitpunkt des Schreibens nur zwei Familien von Audioformaten unterstützt, können weitere Formate oder Formatfamilien einfach hinzugefügt werden, da die genaue Formatwahl für die restlichen Teile des Programms transparent ist. Sie greifen auf die benötigten Eigenschaften durch das von AudioFormat definierte Interface zu. Die einzige Ausnahme bilden die verschiedenen Audio-Interfaces, da sie überprüfen müssen, ob das vom Benutzer gewählte Format von der Audiohardware unterstützt wird. Dies ist auch der Grund, weshalb Audio-Formate und Audio-Interfaces zusammen ein Subsystem bilden — gäbe es diese Abhängigkeit nicht, könnten die Formatklassen auch zu den Hilfsklassen zählen.

AudioFormat

Wie bereits angetönt, definiert die Klasse AudioFormat ein allgemeines Interface für die verschiedenen Formatklassen, das alle üblichen Operationen enthält. Ausserdem enthält die Klasse eine Enumeration FormatID, die den statischen Typ der Instanz enthält und so typensichere Casts erlaubt[9]. Die folgenden Ausführun-

[9] Dies war notwendig, da die erste verwendete Version von g++ die neuen Cast-Operatoren des C++-Standards nicht kannte.

gen gehen auf die Eigenheiten der einzelnen Formate nur ein, wenn dies nötig erscheint. Weitere Informationen können in 1.2.6 nachgelesen werden.

Klasse: AudioFormat	
Superklassen: -	
Subklassen: PCMFormat, ULawFormat	
Kann sich klonen.	`clone()`
Gibt die Einheitsgrösse des Formats zurück.	`sample_size()`
Mischt zwei Puffer, deren Inhalt in diesem Format ist.	`mix()`
Konvertiert einen Puffer in ein anderes AudioFormat.	`convert_to()`
Konvertiert Daten von lokaler zu Netzwerkdarstellung und zurück.	`hton()` `ntoh`
Füllt Puffer mit dem formatabhängigen Nullsignal.	`bzero()`
Kodiert und dekodiert sich für die Übertragung.	`encode(), decode()`
Unterstützt Vergleichsoperatoren.	
Der statische Typ der Klasse kann abgefragt werden.	

PCMFormat

Das einfachste und gebräuchlichste Audioformat ist PCM. Allerdings gibt es viele verschiedene Unterformate mit unterschiedlicher Bittiefe, Samplerate und Byteordnung. Die Daten können ausserdem vorzeichenlos oder vorzeichenbehaftet sein und Mono- oder Stereosignale enthalten. Die PCMFormat-Klasse implementiert die gebräuchlisten Kombinationen dieser Parameter.

Klasse: PCMFormat	
Superklassen: AudioFormat	
Subklassen: -	
Kennt die Samplerate.	`rate`
Kennt die Bittiefe (8 oder 16).	`bits`
Weiss, ob Stereodaten vorliegen.	`stereo`
Weiss, ob die Daten vorzeichenbehaftet sind.	`sign`
Implementiert das AudioFormat-Interface.	
Wirft bei Fehlern Exceptions.	Exception, Assertion

ULawFormat

Im PSTN sind die gebräuchlisten Audioformate μ-Law und A-Law, wovon μ-Law das einfachere ist. Theoretisch könnte dieses Format verschiedene Bitraten und -tiefen unterstützen, tatsächlich wird aber meist eine feste Bittiefe von 8 gewählt. Auch die Bitrate hat üblicherweise einen festen Wert von 8000. Da bei der Konvertierung zwischen verschiedenen Raten aber auf die Routinen von PCMFormat zurückgegriffen werden kann, implementiert ULawFormat variable Bitraten, was höhere Flexibilität zur Folge hat.

Klasse: ULawFormat	
Superklassen: AudioFormat	
Subklassen: -	
Kennt die Samplerate.	`rate`
Implementiert das AudioFormat-Interface.	
Wirft bei Fehlern Exceptions.	Exception, Assertion

4.2.3.2 Audio-Interface

Obwohl mit dem Open Sound System (OSS) des Linux-Kernels eine einheitliche Schnittstelle für eine Vielzahl von Soundkarten und ähnlichen Geräten zur Verfügung steht, gibt es eine Reihe von Bibliotheken, die den Applikationen entweder ein leistungsfähigeres Interface bieten oder die dürftige Ausnutzung der Fähigkeiten einiger Soundkarten verbessern. In DSPhone werden zwei solche Bibliotheken verwendet: Die ESD-Bibliothek (Enlightened Sound Daemon), die auf OSS aufsetzt und automatisch eine beliebige Anzahl von Audioströmen konvertieren und mischen kann, und die Advanced Linux Sound Architecture ALSA, die nicht nur eine entsprechende Bibliothek bietet, sondern auch gleich noch die Kernelmodule durch eigene, leistungsfähigere ersetzt.
Aus diesen Gründen und wegen dem Bedarf an logischen Audiogeräten entstand die AudioIface-Klasse (Iface steht kurz für Interface). Sie dient als Schnittstelle zwischen Audio-Subsystem und Backend und implementiert die Funktionalitäten allgemeinen Charakters, teilweise mittels Templatemethoden[10]. Zum Zweck der Anbindung an die Backend-Klassen erbt AudioIface von Source und Sink.

Klasse: AudioIface (abstrakt)	
Superklassen: Source, Sink	
Subklassen: NullIface, LoopBackIface, FileIface, AlsaIface, EsoundIface, OssIface	
Hat einen Ausgangspuffer.	ProdConBuf
In diesen Puffer kann geschrieben werden.	`write()`
Kennt das Format der hereinkommenden Daten.	AudioFormat
Hat abstrakte Methoden für den Zugriff auf das Gerät.	`read(), dev_write()`
Konstruktor nimmt Buffer- und I/O-Einheiten-Grösse.	
Gibt die tatsächliche Grösse der I/O-Einheiten zurück.	`chunk_size()`
Der Pufferinhalt wird von einem internen Thread weitergeleitet.	MThread
Der Thread kann gestartet und gestoppt werden.	`start(), terminate()`
Gibt Meldungen über eine Log-Instanz aus.	Log
Wirft bei Fehlern Exceptions.	AudioException

[10] Damit sind nicht C++-Templates gemeint, sondern das Designpattern dieses Namens.

Bibliotheksanbindungen

Von den geplanten drei Bibliotheksanbindungen wurden schliesslich nur zwei verwirklicht. Die OSS-Anbindung erwies sich als ungeeignet, da nur die wenigsten Kernmodule Duplexbetrieb erlauben. Von ESD gibt es glücklicherweise eine ALSA-Anbindung, wodurch schliesslich zwei Anbindungen realisiert werden konnten. Es folgt ein kurzer Überblick über die beiden Klassen, die Implementierungen folgen den jeweiligen Dokumentationen.

Klasse: EsoundIface	
Superklassen: AudioIface	
Subklassen: -	
Hat ein ESD-Handle.	`esd`
Hat Filedeskriptoren für In- und Output.	`infd, outfd`
Öffnet im Konstruktor eine Verbindung zum ESD-Daemon.	
Kann vom Daemon Daten lesen.	`read()`
Kann Daten an den Daemon weiterleiten.	`dev_write()`
Unterstützt beliebige Chunk-Grössen.	
Wirft bei Fehlern Exceptions.	EsoundException

Klasse: AlsaIface	
Superklassen: AudioIface	
Subklassen: -	
Hat ein Handle für ein ALSA-Gerät.	`handle`
Öffnet im Konstruktor das Default-PCM-Gerät.	
Kann optional nach jeder n-ten gelesenen Einheit Diagnosemeldungen mit `dprintf` ausgeben.	
Versucht die gewünschte Chunk-Grösse zu setzen und ermittelt die danach tatsächlich verwendete.	
Kann vom Gerät Daten lesen.	`read()`
Kann Daten an das Gerät weiterleiten.	`dev_write()`
Wirft bei Fehlern Exceptions.	AlsaException

Logische Geräte

Um wiederholbare Tests realisieren zu können, muss das Programm verschieden logische Geräte enthalten. Für einfache Tests ist das NullIface gedacht, das ein Nullsignal generiert und Daten nicht abspielt, sondern einfach ignoriert. Das FileIface andererseits liest Audiodaten aus einer Datei und schreibt hereinkommende Daten in eine andere. Damit kann dasselbe Signal für beliebig viele Tests verwendet und die Resultate direkt gespeichert werden. Beide, NullIface und FileIface, verzögern Leseoperationen so lange, wie es bei echter Soundhardware der Fall wäre, um das Timing des Programms zu erhalten.

Etwas ausgefallener ist das LoopBackIface, das mit `dev_write()` geschriebene Daten über die `read`-Methode wieder ausgibt. Damit lässt sich ein Echoeffekt simulieren, was nützlich sein kann, um die Verzögerung des Echosignals zu ermitteln.

Klasse: NullIface	
Superklassen: AudioIface	
Subklassen: -	
"Vergisst" hereinkommende Daten.	`dev_write()`
Gibt ein Nullsignal zurück.	`read()`
Das Lesen wird solange blockiert, bis die Datenrate derjenigen des Audioformats entspricht.	Time, Mutex, Cond AudioFormat

Klasse: FileIface	
Superklassen: AudioIface	
Subklassen: -	
Hat zwei geöffnete Files.	`fin, fout`
Gibt den Inhalt von `fin` aus.	`read()`
Speichert hereinkommende Daten in `fout`.	`dev_write()`
Eines der Files kann NIL sein, um eine lese- oder schreib-geschützte Datei zu simulieren.	
Das Lesen wird solange blockiert, bis die Datenrate derjenigen des Audioformats entspricht.	Time, Mutex, Cond AudioFormat
Wirft bei Fehlern Exceptions.	Assertion, FileException

Klasse: LoopBackIface	
Superklassen: AudioIface	
Subklassen: -	
Hat einen Zwischenspeicher.	ProdConBuf
`dev_write()` schreibt in diesen Puffer.	
`read()` liest daraus.	
Beide Operationen können also blockieren.	

4.2.4 SimpleTCP

Die SimpleTCP-Klasse dient dazu, die Schnittstelle zwischen Backend und Protokoll-Subsystem zu testen, und gibt ein einfaches Beispiel für die Implementierung eines solchen Subsystems. Sie überträgt die Audiodaten direkt über die schon geöffnete TCP-Verbindung und definiert zu diesem Zweck ein kleines Protokoll. Beide Parteien senden abwechselnd einen Befehl, gefolgt von Argumenten oder Daten, beginnend mit dem Server. Da das Protokoll möglichst einfach sein soll, kann es nur in einem der Duplex-Modi betrieben werden. Die Syntax und Semantik der Befehle ist in folgender Tabelle aufgelistet.

Befehl	**Syntax**	**Bedeutung**
stop	`s`	Beendet die Verbindung
data	`d<data>`	Überträgt 1024 Datenbytes

Das Design von SimpleTCP ist bewusst sehr einfach gehalten, was aber ihrem Anwendungszweck entspricht. Grösse und Format der Übertragungseinheiten sind fix und es werden keine Statistiken der Übertragungsqualität geführt.

Klasse: SimpleTCP	
Superklassen: Protocol	
Subklassen: -	
Hat das Format der übertragenen Daten.	AudioFormat
Verwendet Blöcke fester Grösse für die Übertragung.	`chunk_size`
Hat Queues für ein- und ausgehende Blöcke.	`incoming, outgoing`
Implementiert die Source- und Sink-Interfaces.	`read(), write()`
Implementiert Server- und Client-Seite des Protokolls.	`server(), client()`
Kann die Übertragung beenden.	`end_transmission()`

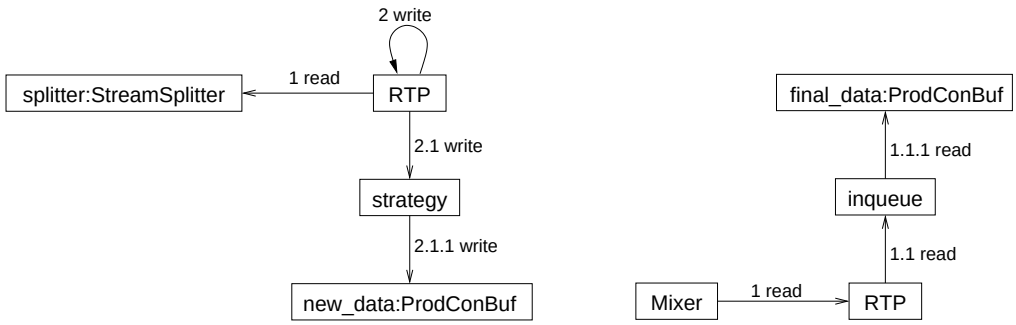
4.2.5 RTP-Subsystem

Das RTP-Subsystem ist wohl der wichtigste Teil von DSPhone, da in ihm die verschiedenen in Kapitel 3 behandelten Ansätze integriert sind. Das beinhaltet auch das Markieren prioritärer Pakete mit der Unterteilung von Frames, entweder in High- und Low-Nibbles oder in Samples mit geradem und ungeradem Index, obwohl hierfür keine Tests durchgeführt wurden.

4.2.5.1 Struktur

Innerhalb des Subsystems nimmt die *RTP*-Klasse eine zentrale Rolle ein. Sie sendet und empfängt Pakete und kommuniziert mit dem Backend, delegiert aber die eigentliche Funktionalität an die Klassen RTPStrategy und RTPInQueue, wobei diese Funktionalität je nach Backend-Modus variieren kann. RTP enthält ausserdem alle drei zum Subsystem gehörenden Threads.
RTPStrategy produziert Pakete nach einer bestimmten Strategie, die auf den aktuellen Übertragungsstatistiken und einem Entscheidungsalgorithmus basiert. Statistikverwaltung und Algorithmus sind wiederum in die Klassen *RTPStats* und *Policer* ausgelagert. Die verschiedenen von Policer beinflussten Parameter sind ausserdem in der Klasse *Policy* gekapselt.
Ankommende Datenpakete gelangen in die *RTPInQueue*, wo die Audiodaten extrahiert und konvertiert werden, bevor sie dem Backend übergeben werden. Dazu gehört auch die Aufgabe, die Daten von aufgeteilten Paketen wieder zusammenzufügen.
Netzwerkpakete sind dafür optimiert, alle relevanten Informationen in kleinstmöglicher Länge zu kodieren. Da dies aber den Zugriff auf die einzelnen Felder erschwert, eignen sich diese Strukturen nur bedingt als Datenstrukturen innerhalb eines Programms. Dieses Problem kann umgangen werden, indem man sie durch Klassen kapselt, die Zugriffsmethoden für alle relevanten Informationen definiert

und gegebenenfalls Felder automatisch mit sinnvollen Daten füllt. Die RTP-Paket-Hierarchie des RTP-Subsystems (siehe Abschnitt 4.2.5.4) übernimmt diese Aufgabe.
Die in Abb. 4.17 und Abb. 4.18 enthaltenen Diagramme und das Klassendiagramm von Seite 146 geben einen guten Überblick über die Zusammenhänge. Einige Details, wie die RTCP-Bandbreitenberechnung oder das Scout-Objekt, sind in den Ablaufdiagrammen nicht enthalten, werden aber in den folgenden Abschnitten erläutert.



(a) Datenlesezyklus (Duplexmodus)

(b) Datenweitergabe ans Backend

Abbildung 4.17: Abläufe im RTP-Subsystem

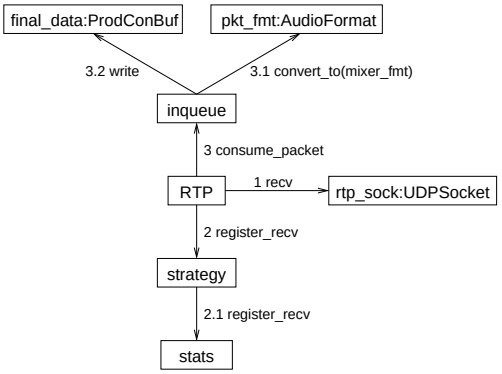
4.2.5.2 Zentrale Klassen

RTP

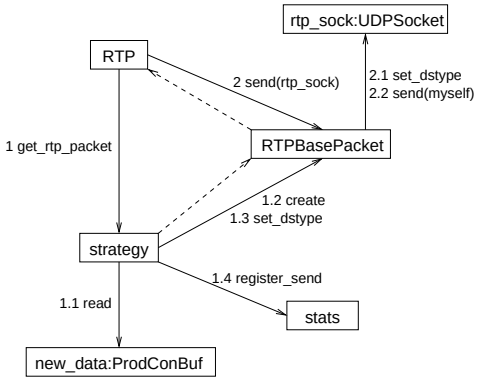
Da die RTP-Klasse den zentralen und für das Backend den einzig sichtbaren Teil des RTP-Subsystems darstellt, erbt sie von Protocol. Sie hat auch zwei Konstruktoren, einen für den Server- und einen für den Client-Modus. Im Gegensatz zu SimpleTCP beschränken sich die Unterschiede zwischen diesen beiden Betriebsarten aber auf die Konstruktoren — der eine greift auf die Konfiguration zurück, um das Subsystem zu kreieren, der andere benutzt dazu den vom Server übertragenen Konfigurationsstring. Dieser hat das Format

```
RTP M 0xSSSSSSSS 0xPPPP 0xQQQQ.
```

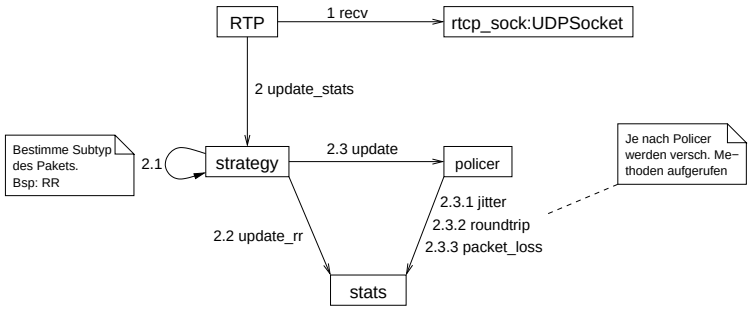
M bezeichnet den Modus, den der Client annehmen soll(’D’ für Duplex, ’d’ für Simple Duplex, ’r’ für Receive und ’s’ für Send), S die SSRC des Servers in acht Hexadezimal-Ziffern, P den RTP-Port (4 Ziffern) und Q den RTCP-Port (4 Ziffern). Da die RTP-Sitzung nur aus zwei Parteien besteht, erhält der Client



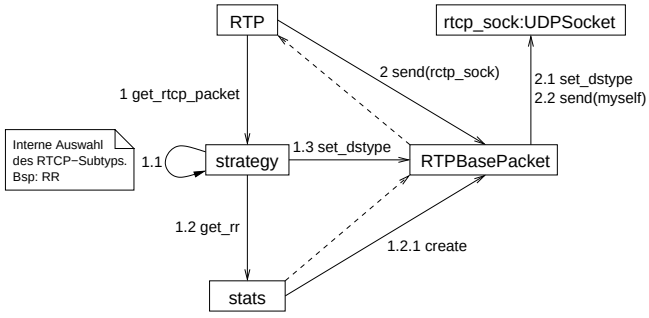
(a) Empfang eines RTP-Pakets



(b) Erstellen und Senden eines RTP-Pakets



(c) Empfang eines RTCP-Pakets



(d) Erstellen und Senden eines RTCP-Pakets

Abbildung 4.18: Abläufe im RTP-Subsystem (forts.)

seinen SSRC-Identifikator durch das bitweise Inverse des Server-Identifikators. Nach der Initialisierung verläuft die Kommunikation zwischen den Subsystemen beider Endpunkte symmetrisch. Dementsprechend sind die Methoden `server` und `client` nur Umleitungen auf die `run`-Methode. Die Threads des Subsystems, `fetcher receiver` und `sender` sind alle in dieser Klasse enthalten. Ihre Aufgaben werden weiter unten betrachtet.

Klasse: RTP	
Superklassen: Protocol	
Subklassen: -	
Kennt SSRC und CNAME seines Endsystems.	
Hat Sockets für RTP- und RTCP-Pakete.	UDPSocket
Hat `inqueue` und `strategy`.	RTPStrategy, RTPInqueue
Enthält "Datenholer"-Thread.	MThread
Enthält Empfänger-Thread.	MThread
Enthält Sender-Thread.	MThread
Threads verhalten sich je nach Backend-Modus.	
Implementiert das Protocol-Interface.	
Hat Servermodus-Konstruktor.	Config
Hat Clientmodus Konstruktor (nimmt Config-String).	
`server()` und `client()` sind symmetrisch.	`run()`
Delegiert `read()` an `inqueue`.	RTPInQueue
Delegiert `write()` an `strategy`.	RTPStrategy
Leitet ankommende Pakete an `inqueue` weiter.	Socket, RTPInQueue
Holt Pakete von `strategy` und sendet sie.	RTPStrategy, Socket
Wirft bei Fehlern Exceptions.	Assertion, SocketException

RTPInQueue

RTP-Pakete, die empfangen werden, müssen auf ihre Aktualität geprüft, gegebenenfalls mit früher erhaltenen Paketen vereinigt und ihr Inhalt ins interne Audioformat konvertiert werden. Ausserdem müssen die so behandelten Daten zwischengespeichert und dem Backend zugänglich gemacht werden. Diese Aufgaben übernimmt die Klasse RTPInQueue. Um die Datenweitergabe einheitlich zu halten erbt sie von Source.
Wenn RTP der `consume_packet`-Methode von RTPInQueue ein Paket übergibt, muss diese erst bestimmen, wie das Paket zu behandeln ist. Abb. 4.19 zeigt den entsprechenden Entscheidungsbaum, die Bedeutung der dazugehörigen Aktionen können der Tabelle 4.3 entnommen werden.

Dazu muss bemerkt werden, dass mit `proc_pkt` ein allfälliges Teilpaket gemeint ist, dessen Gegenstück noch nicht empfangen wurde. In Fällen, wo das Gegenstück nicht in nützlicher Frist ankommt, kann der ursprüngliche Paketinhalt aus dem Paket teilweise rekonstruiert werden.
Die restlichen Eigenschaften der RTPInQueue-Klasse können der CRC-Karte entnommen werden.

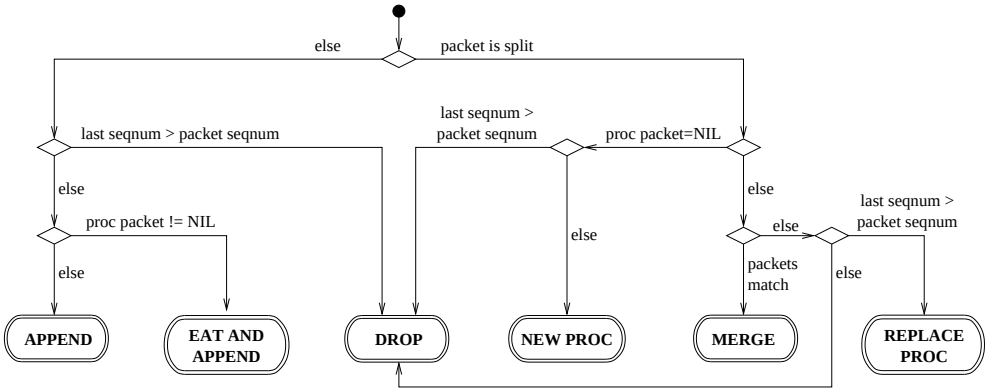


Abbildung 4.19: Entscheidungsbaum von `consume_packet`

APPEND	Konvertiere den Paketinhalt und hänge ihn an die Ausgangsqueue.
EAT AND APPEND	Wie APPEND, verwerte aber das vorher empfangene Teilpaket.
DROP	Verwirf das empfangene Paket.
MERGE	Das neue Paket ist das Gegenstück zu `proc_pkt`, vereinige sie.
NEW PROC	Benutze das neue Paket als `proc_pkt`.
REPLACE PROC	Wie NEW PROC, verwerte aber das vorher empfangene Teilpaket.

Tabelle 4.3: Aktionen in `consume_packet`

Klasse: RTPInQueue	
Superklassen: Source	
Subklassen: -	
Kennt das Backend-Audioformat.	AudioFormat
Verwertet oder verwirft ankommende Pakete.	RTPPacket
Vereinigt, wenn möglich, aufgeteilte Pakete.	
Konvertiert die ankommenden Daten ins Backend-Format.	
Schreibt die Resultate in einen Zwischenspeicher.	ProdConBuf
Leitet die Resultate auf Anfrage weiter.	
Wirft bei Fehlern Exceptions.	Assertion, RTPPacketException

RTPStrategy

Grundsätzlich soll RTPStrategy die vom Backend kommenden Daten zwischenspeichern und auf Anfrage RTP- und RTCP-Pakete nach einer vorher gewählten Strategie produzieren. Die erste Aufgabe ist einfach gelöst: `write`-Aufrufe werden einfach an eine ProdConBuf-Instanz weitergeleitet.
Um hingegen eine Sendestrategie implementieren zu können, benötigt RTPStrategy entsprechende Informationen über den Charakter der Verbindung. Es liegt also nahe, die Auswertung empfangener RTCP-Pakete dieser Klasse zu überlassen. Hinzu kommt die Forderung nach einem dynamisch auswechselbaren Strategiealgorithmus sowie nach korrekter Auswahl und Bandbreitenkontrolle der RTCP-Pakete. Ausserdem müssen die eigenen Sende- und Empfangsstatistiken gespeichert und in Form von RTCP-Paketen wiedergegeben werden können.
Bei der Berechnung der RTCP-Bandbreiten kann ein Sonderfall eintreten: Normalerweise genügt es, für die Berechnung die gesendeten RTP-Pakete als Referenz zu nehmen. Ist das Backend aber im Receive-Modus, werden keine Pakete gesendet, weshalb in diesem Fall die empfangenen RTP-Pakete die Referenz darstellen. All dies führt zu folgender Struktur:
Die Klasse lagert die Auswertung und Speicherung der Statistiken sowie die Generierung der dazugehörigen RTCP-Pakete in die Klasse *RTPStats* aus. Die neuen Paketen zukommenden Regeln wie Grösse und DiffServ-Typ sind in der Klasse *Policy* abgelegt. Den Algorithmus, der diese Regeln bestimmt, kapselt das Interface *Policer*, was die Wahl des Algorithmus dynamisch wählbar macht. Einige Policer benutzen ein sogenanntes *Scout*-Objekt, das eigene RTCP-Pakete erzeugen kann und entspechend bei der Auswahl der RTCP-Pakete berückichtigt werden muss. Schliesslich muss die Bandbreite der gesendeten RTCP-Pakete kontrolliert werden, was durch Instanzen der *BWShare*-Klasse erreicht wird, die wahlweise von den gesendeten oder den empfangenen Paketen in Kenntnis gesetzt werden. Jeder RTCP-Typ erhält so einen gewissen Anteil an der Gesamtbandbreite der Sitzung. Eine graphische Darstellung dieser Struktur ist im Klassendiagramm von Seite 146 enthalten.

Klasse: RTPStrategy	
Superklassen: Sink	
Subklassen: -	
Neue Daten können übergeben werden.	`write()`
Hat einen Zwischenspeicher für Daten.	ProdConBuf
Kennt das Backend-Audioformat und die Profil-Formate.	AudioFormat
Registriert Empfang und Senden von Paketen.	RTPBasePacket
Delegiert Statistikverwaltung und RTCP-Verarbeitung.	RTPStats
Generiert RTP- und RTCP-Pakete.	`get_rtp_packet()` `get_rtcp_packet()`
Dazu führt sie eine Sequenznummer und eine künstliche Sampleuhr.	Time
Delegiert RTCP-Generierung.	RTPStats, Scout
Hat Senderegeln.	Policy
Hat Strategiealgorithmus.	Policer
Kontrolliert Anzahl und Sendezeit der RTCP-Pakete.	BWShare
Kann diese Kontrolle von den gesendeten oder den empfangenen Paketen abhängig machen.	
Kennt den SSRC-Identifikator und den CNAME.	
Wirft bei Fehlern Exceptions.	Assertion

Da die Bandbreitenkontrolle für RTCP-Pakete zwar einfach ist, aber durch die zahlreichen RTCP-Typen die Lesbarkeit des Codes vermindert, kapselt die BWShare-Klasse diese Funktionalität. Der Algorithmus funktioniert für einen RTCP-Subtyp T folgendermassen:

Sei $psize$ die Paketgrösse von T,
$share$ der Anteil von T an der Gesamtbandbreite.
Initialisiere $bytecount$ auf 0

Für jedes gesendete Paket {
    bytecount = bytecount + Grösse des gesendeten Pakets
    Falls $bytecount \cdot share \geq psize$ {
        Sende T-Paket
        $bytecount = -psize$ // Ergibt mit obiger Addition $bytecount = 0$
    }
}

Die Klasse hat folgende Verantwortlichkeiten:

Klasse: BWShare	
Superklassen: -	
Subklassen: -	

Kennt einen Bandbreitenanteil und eine Paketgrösse.
Die Paketgrösse kann nach der Konstruktion verändert werden.
Führt einen internen Bytezähler.
Die Grössen gesendeter Pakete können darauf addiert werden.
Gibt zurück, ob ein Paket gesendet werden darf.
Der Bytezähler kann zurückgesetzt werden.
	
	`set_pkt_size()`
	
	`fill()`
	`ready_to_send()`
	`flush()`

RTPStats

Die RTP-Klasse speichert die Empfangs- und Sendestatistiken sowie die Daten, die in den vom anderen Endpunkt gesendeten RTCP-Paketen enthalten sind, und implementiert verschiedene Berechnungen auf diesen. Mit Ausnahme des APP-Subtyps verarbeitet und generiert sie ausserdem alle RTCP-Pakete.
Wo möglich sind die Werte in Gruppen eingeteilt, dargestellt durch `structs`. Diese sind `sr`, `rr`, `sr` und `sdes` für die per RTCP empfangenen Informationen, sowie `rcv` und `snd` für lokale Empfangs- und Sendestatistiken. Unabhängig von diesen Gruppen sind nur die beiden SSRC-Identifikatoren der Sitzung gespeichert. Intern hat die Klasse eine Reihe von Hilfsmethoden und -variablen, unter anderem eine LimitedMap[11] gesendeter SR-Pakete, die für die Roundtrip-Berechnung benötigt wird.

Klasse: RTPStats	
Superklassen: -	
Subklassen: -	
Speichert Ankunftszeit und Inhalte des letzten SR-Pakets und des letzten RR-Pakets. Speichert die aktuellen SDES-Werte der Gegenseite. Führt über die gesendeten und empfangenen Pakete Buch. Bietet Zugang zu diesen Statistiken.	RRPacket, Time SRPacket SDESPacket
Generiert RR-, SR-, sowie SDES-Pakete, letztere in Minimal- und Maximal-Ausführung.	RRPacket, SRPacket SDESPacket

Berechnet auf Anfrage auf Sekunden normalisierte, aktuelle
Werte für Jitter, Packet Loss und Roundtrip.
Roundtrip wird direkt und gefiltert angeboten, um unstabiles
Verhalten bei kurzfristigen Spitzen vermeiden zu können.
	`jitter()`
	`loss_ratio()`
	`roundtrip()`
	`last_roundtrip()`

Merkt sich Sendezeit und ID der letzten n SR-Pakete, um damit die Roundtripzeit zu berechnen.	LimitedMap
Kennt die Sampleclock-Frequenz (für Roundt.-Berechnungen).	

[11] siehe Abschnitt 4.2.1.7

Threads

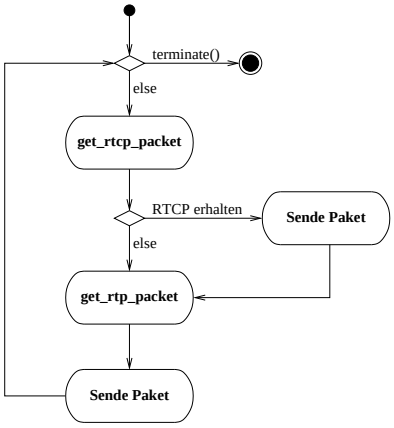
Im RTP-Subsystem gibt es drei Threads, die alle zur RTP-Klasse gehören.
Fetcher wird in allen Modi ausser Receive ausgeführt. Seine Aufgabe ist einfach: Er liest Daten vom Backend und übergibt sie der `write`-Methode von RTP. Diese wiederum ruft die `write`-Methode von RTPStrategy und damit diejenige von `new_data` (der ProdConBuf in RTPStrategy) auf. Im Sendemodus werden die Daten direkt vom AudioIface gelesen, ansonsten wird der StreamSplitter benutzt.
Die Aufgabe von *Receiver* ist es, auf Pakete zu warten, die auf `rtp_sock` oder `rtcp_sock` ankommen. Darauf wird überprüft, ob das Paket ein RTP-Paket ist, um es danach RTPStrategy oder RTPInQueue zu übergeben. Im Sendemodus werden nur Pakete auf `rtcp_sock` empfangen.
Nach dem Start des Subsystems wartet der Thread des Initiators darauf, dass Receiver terminiert. Dies deshalb, weil eventuelle BYE-Pakete der anderen Partei hier empfangen werden.
Sender fordert von RTPStrategy abwechselnd ein RTCP- und ein RTP-Paket an und versendet es über das entsprechende Socket. Ist kein RTCP-Paket fällig, gibt RTPStrategy einen NULL-Zeiger zurück, worauf Sender direkt mit dem nächsten RTP-Paket weitermacht. Im Receivemodus fordert der Thread nur RTCP-Pakete an. Abb. 4.20 zeigt diese Abläufe.

4.2.5.3 Policer und Scouts

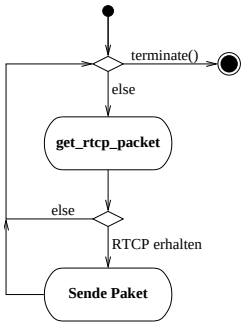
Um die verschiedenen Strategien ohne erneutes Kompilieren des Programms austauschbar zu machen, definiert die Klasse Policer ein einfaches Interface, das die Unterschiede zwischen ihnen kapselt, und das nur eine einzige Methode enthält, `udpate()`. Hinzu kommt eine Factory-Funktion, die die Konfigurationsvariable "policer" ausliest und eine Instanz des entsprechenden Policers zurückgibt.
Einem Policer werden Referenzen auf Instanzen von RTPStats und Policy übergeben, ausserdem eine Referenz auf den Scout-Zeiger in RTPStrategy. Dies rührt daher, dass einige Policer den aktiven Scout auswechseln.

Klasse: Policer	
Superklassen: -	
Subklassen: DefaultPolicer, StaticPolicer, SwitchingPolicer, ServicePolicer	
Kennt Statistiken und Policy.	RTPStats, Policy
Hat eine Referenz auf den Scout-Zeiger in RTPStrategy und kann ihn umsetzen.	Scout RTPStrategy
Kennt den eigenen SSRC-Identifikator (benötigt, um neue Scouts generieren zu können).	
Aktualisiert aufgrund von Statistik und Scout die Policy.	`update()`

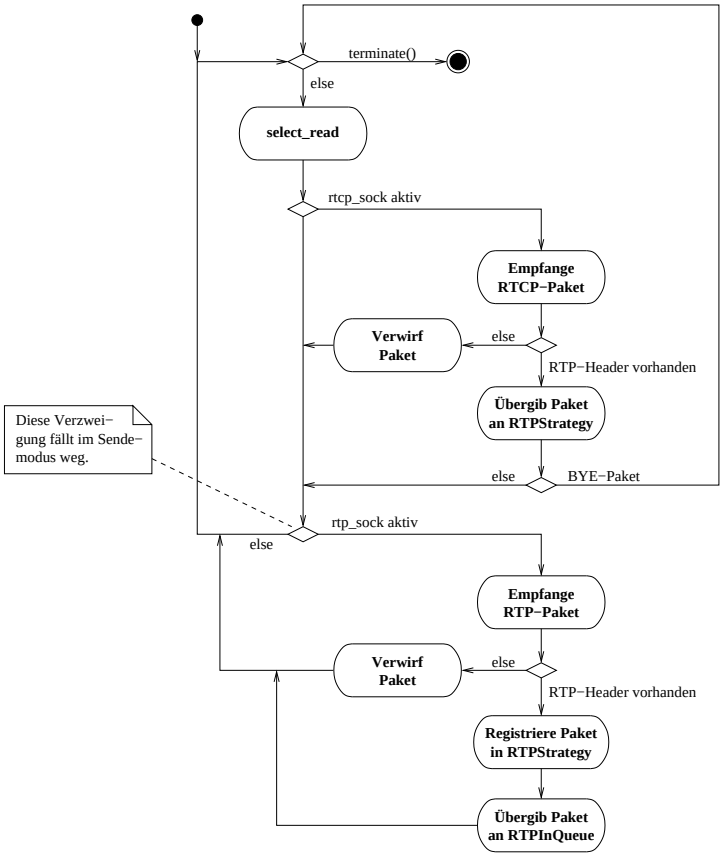
Bevor nun die einzelnen Policer behandelt werden, ist eine Einführung der beiden Scouts angebracht.



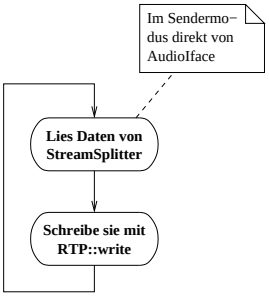
(a) Sender



(b) Sender (Receive-Modus)



(c) Receiver



(d) Fetcher

Abbildung 4.20: Abläufe der RTP-Threads

Scouts

Scouts sind Objekte, die RTCP-APP-Pakete generieren, um die Situation in einer anderen Serviceklasse “auszuspähen”, ohne die Bandbreite merklich zu belasten. Wie Policer ist Scout ein abstraktes Interface, das die dahinterliegende Implementierung auswechselbar und für RTPStrategy transparent macht. Es enthält einerseits Methoden, die RTCP-Pakete generieren und verwerten und solche, die dem Policer die Resultate einer Messreihe mitteilen. Die CRC-Karte beschreibt die Aufgaben, die ein Scout erfüllen muss.

Klasse: Scout	
Superklassen: -	
Subklassen: NoopScout, ServiceScout	
Generiert normale RTCP-Pakete.	`get_probe(false)`
Generiert dringende (Antwort-) Pakete.	`get_probe(true)`
Wenn die Messreihe beendet ist, werden keine Pakete mehr generiert.	
Empfängt Pakete der Gegenseite und antwortet gegebenenfalls.	`receive(packet)`
Zeigt an, wann eine Messreihe beendet ist.	`bool done()`

Gibt die Resultate einer beendeten Messreihe zurück.	`double jitter()`
	`double mean_delay()`
	`double loss_ratio()`

Kennt die SSRC, um Pakete generieren zu können.	
Gibt den geprüften DiffServ-Typ zurück.	`probed_type()`

Dieses Interface wird von zwei Klassen implementiert, NoopScout und ServiceScout. Wie der Name schon sagt, tut der NoopScout nichts — genauer: er verhält sich immer, als hätte er eine Messreihe beendet[12]. Er eignet sich somit für Situationen, in denen ein Scout nicht sinnvoll ist, beispielsweise bei einem statischen Policer. ServiceScout hingegen sendet eine gewisse Anzahl von RTCP-Ping-Paketen, die vom Scout der gegenüberliegenden Partei beantwortet werden und deren Verhalten eine grobe Schätzung von Jitter, Verlustrate und Verzögerung ermöglicht.

[12] Pakete der anderen Partei müssen trotzdem beantwortet werden.

Klasse: ServiceScout	
Superklassen: Scout	
Subklassen: -	
Implementiert das Scout-Interface.	
Kennt den zu testenden DiffServ-Typ.	DSType
Hat ein maximales Zeitfenster für Antworten.	Time
Generiert eine konstante Anzahl Probe-Pakete.	Ping
Diese haben Sequenznummern 0–n.	
Führt eine Liste der gesendeten und beantworteten Pakete.	
Berechnet nach Abschluss der Messreihe die Schätzwerte.	

Wenn ServiceScout eine Messreihe beendet hat, muss die Instanz ersetzt werden, um eine neue starten zu können.

Policer-Implementierungen

DSPhone beinhaltet zum Zeitpunkt des Schreibens vier Policer.
Der einfachste davon ist der *DefaultPolicer*, der die Policy auf fest vorgegebene Werte setzt. Der einzige konfigurierbare Parameter ist das Sendeintervall der RTP-Pakete.

Klasse: DefaultPolicer	
Superklassen: Policer	
Subklassen: -	
Implementiert das Policer-Interface.	
Setzt die Policy-Variablen.	Policy
Setzt μ-Law-Format ohne Paketaufteilung.	
Übernimmt die Backend-Audiorate und setzt die Payloadgrösse auf das Äquivalent des konfigurierten Paketintervalls.	Config
Setzt den Default-DiffServ-Typ.	DSType
Übernimmt den von RTPStrategy gesetzten NoopScout.	

Auch von statischer Natur, aber etwas flexibler ist *StaticPolicer*. Ein entsprechender Eintrag in der Konfigurationsdatei legt die DiffServ-Klasse, mit der die Pakete gesendet werden, fest.

Klasse: StaticPolicer	
Superklassen: Policer	
Subklassen: -	
Implementiert das Policer-Interface.	
Setzt die Policy-Variablen.	Policy
Setzt μ-Law-Format ohne Paketaufteilung.	
Übernimmt die Backend-Audiorate und setzt die Payloadgrösse auf das Äquivalent des konfigurierten Paketintervalls.	Config
Setzt den in der Konfigurationsvariablen “static_dstype” abgelegten DiffServ-Typ.	DSType
Übernimmt den von RTPStrategy gesetzten NoopScout.	

Beide Policer verwenden keinen Scout, setzen also den Scout-Zeiger auf eine Instanz von NoopScout. Anders *SwitchingPolicer* und *ServicePolicer*: sobald sie sich in einer höheren DiffServ-Klasse befinden, setzen sie einen ServiceScout für die jeweils tiefere Klasse ein, um wieder auf diese zurückfallen zu können, falls sich die Situation in dieser Klasse verbessert. Die folgenden Ausführungen geben nur einen groben Überblick über die Implementierung der beiden Klassen. Details zu den dazugehörigen Entscheidungsalgorithmen sind in Abschnitt 4.1.1.3 aufgeführt.

Der SwitchingPolicer setzt den Ansatz der Adaption der Serviceklasse in einer einfachen Form um, indem er zwischen zwei wählbaren Klassen wechselt. Die erwartungsgemäss schlechtere wird “tiefe Klasse” genannt, die andere “hohe Klasse”. Die drei Parameter Jitter, Verzögerung und Paketverlustrate werden durch Evaluationsfunktionen bewertet und ergeben in ihrer Summe eine Gesamtbewertung der Situation. Steigt diese über den `treshold`-Wert und ist gerade die tiefe Klasse aktiv, wechselt SwitchingPolicer zur hohen DiffServklasse und setzt einen ServicePolicer ein, um die tiefe Klasse weiter zu beobachten. Ist die Situation in der hohen Klasse gut und die mit derselben Methode ermittelte Bewertung der Scoutresultate unter dem `treshold`-Wert, wird wieder zur tiefen Klasse gewechselt. Um die Parameter bei einem Klassenwechsel einpendeln zu lassen, wird nur nach jeder n-ten Aktualisierung der Statistiken ein Wechsel in Betracht gezogen.

Klasse: SwitchingPolicer	
Superklassen: Policer	
Subklassen: -	
Kennt eine hohe und eine tiefe DiffServ-Klasse. Weiss, welche gerade benutzt wird.	DSType
Benutzt einen NoopScout, wenn die tiefe Klasse aktiv ist.	NoopScout
Überwacht die tiefe Klasse, wenn die hohe aktiv ist.	ServiceScout
Hat drei Evaluationsfunktionen, für Jitter, Verzögerung und Paketverlust.	
Kennt ein Evaluationsintervall und entscheidet immer dann über einen Klassenwechsel, wenn eines vergangen ist.	
Hat einen Schwellenwert, mit dem die Situation in “gut” und “schlecht” eingeteilt werden kann.	
Wechselt von tief zu hoch, wenn die Bewertung schlecht wird.	
Wechselt von hoch zu tief, wenn die Bewertung der hohen Klasse und der Scoutwerte gut ist.	

ServicePolicer bewertet die einzelnen Parameter in ähnlicher Weise wie SwitchingPolicer, betrachtet aber nicht die Summe dieser Bewertungen, sondern vergleicht sie mit den Schwellenwerten `t_j`, `t_d` und `t_pl`, für Jitter, Verzögerung (Delay) und Paketverlustrate (Packet Loss ratio). Je nachdem, welche der Bewertungen zu hoch ist, wird eine andere Serviceklasse gewählt. Auch hier wird, um

die Parameter bei einem Klassenwechsel einpendeln zu lassen, nur nach jeder n-ten Aktualisierung der Statistiken ein Wechsel in Betracht gezogen.

Klasse: ServicePolicer	
Superklassen: Policer	
Subklassen: -	
Weiss, welche DiffServklasse gerade benutzt wird.	DSType
Benutzt einen NoopScout, wenn die Best Effort aktiv ist.	NoopScout
Überwacht die jeweils vorherige Klasse nach einer Steigerung.	ServiceScout
Überwacht die um eins tiefere Klasse nach einer Senkung.	
Ausnahme: Bei BE wird der Scout deaktiviert.	NoopScout
Hat drei Evaluationsfunktionen, für Jitter, Verzögerung und Paketverlust, sowie drei Schwellenwerte für diese Parameter.	
Kennt ein Evaluationsintervall und entscheidet immer dann über einen Klassenwechsel, wenn eines vergangen ist.	
Wechselt je nachdem, welche Bewertung den Schwellenwert überschritten hat.	
Kehrt zurück, wenn die Bewertung der jetzigen und der per Scout überwachten Klasse gut ist.	

Die in diesem Abschnitt besprochenen Zusammenhänge finden sich auch in den Klassendiagrammen von Seite 146 und Seite 147.

Evaluationsfunktionen

Die in Abschnitt 4.1.1.3 aufgeführten Evaluationsfunktionen werden in DSPhone in Klassen gekapselt. Diese erben alle vom abstrakten Interface *EvalFunction*, das nur den Operator () überlädt. Die Implementierungen sind *EvalJitter*, *EvalDelay* und *EvalPL* (für Packet Loss) und halten sich an die vorgegebene mathematische Definition. Die Berechnungsroutine von EvalJitter wurde allerdings etwas optimiert.
Da die Verantwortlichkeiten dieser Klassen einfach ist, und die mathematischen Funktionen schon definiert wurden, dürften die genauen Formeln zusammen mit den CRC-Karten genügen, um das Design darzustellen.

Klasse: EvalFunction	
Superklassen: -	
Subklassen: EvalJitter, EvalDelay, EvalPL	
Definiert eine Familie von Funktionsobjekten, indem sie den Operator () überlädt.	`double operator()(double)`

EvalJitter implementiert die Funktion

$$e_j(x) = \begin{cases} 1 - \frac{1}{1+\left(\frac{x \cdot tp}{s}\right)^4}, & x > 0 \\ 0, & sonst \end{cases}$$

wobei s der gewählte Wendepunkt und tp der Wendepunkt der Funktion im Falle $\frac{tp}{s} = 1$ ist.

Klasse: EvalJitter	
Superklassen: EvalFunction	
Subklassen: -	
Implementiert EvalFunction. Hat einen Skalierungswert s. Dieser kann nachträglich verändert werden.	`set_scale(s)`

EvalDelay implementiert

$$e_d(x) = \begin{cases} 1 - e^{-\frac{x-s}{s}}, & x > s \\ \frac{c(x-s)}{s}, & sonst \end{cases}$$

mit Nullpunkt s und $e_d(0) = c$.

Klasse: EvalDelay	
Superklassen: EvalFunction	
Subklassen: -	
Implementiert EvalFunction. Hat einen Skalierungswert s und einen Kompensationswert c. Der Skalierungswert kann nachträglich verändert werden.	`set_scale(s)`

Schliesslich implementiert EvalPL die Funktion

$$e_{pl}(x) = \begin{cases} m \cdot x, & x \geq 0 \\ 0, & sonst \end{cases}$$

mit Steigung m.

Klasse: EvalPL	
Superklassen: EvalFunction	
Subklassen: -	
Implementiert EvalFunction. Hat eine Steigung m.	

Diese Hierarchie ist auch Teil des Klassendiagramms von Seite 147.

4.2.5.4 RTP-Paket-Hierarchie

Um die Gemeinsamkeiten der verschiedenen Paketformate auszunutzen, und die Komplexität der Paketgenerierung und -verarbeitung hinter einer einfach zu benutzenden Schnittstelle zu verstecken, sind die verschiedenen Pakete und ihre Komponenten in einer Klassenhierarchie abgebildet.
Die Klasse *RTPBasePacket* steht an der Wurzel des Vererbungsbaumes. Sie enthält die rohen Daten eines Pakets, die von den erbenden Klassen strukturiert werden, indem sie Zeiger in diesem Datenblock definieren, über die einzelne Felder extrahiert und eingefügt werden können. Blöcke mit fester Unterteilung, wie zum Beispiel Header, werden durch C-Strukturen interpretiert, die auf die jeweilige Anfangsadresse des Blocks “gecastet” werden. Für Felder, die mehr als ein Byte einnehmen, ist je eine Zugriffsmethode definiert, die eine eventuell nötige Byteordnungs-Umrechnung vornimmt. Für kleinere Felder ist dies nicht notwendig. Abb. 4.21 demonstriert dieses Konzept anhand eines RTP-Pakets.

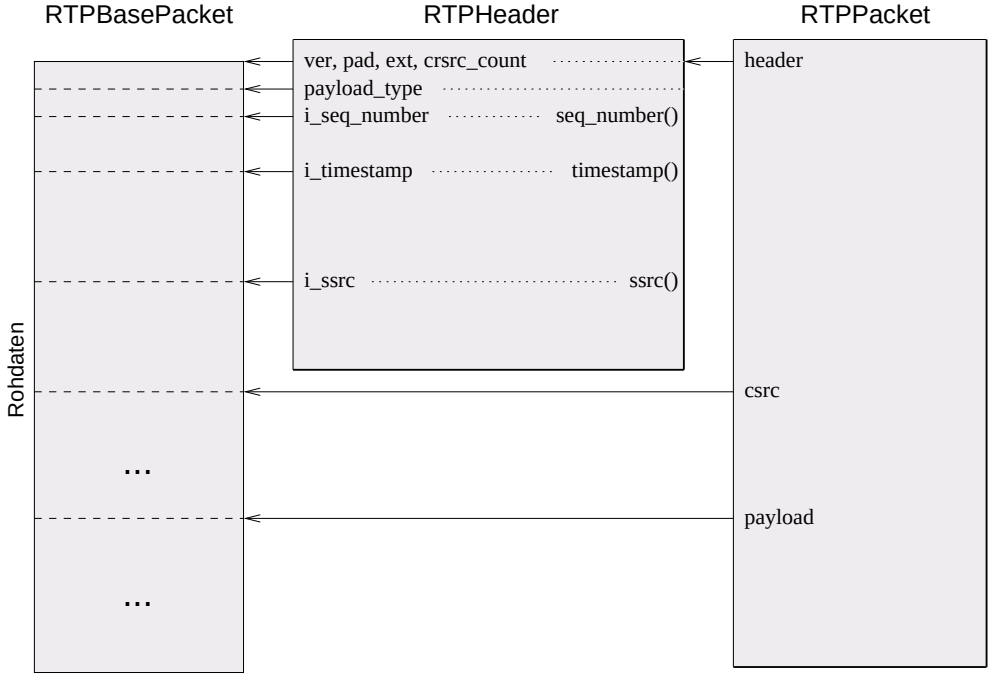


Abbildung 4.21: Zugriff auf die Rohdaten eines RTP-Pakets

Alle von RTPBasePacket erbenden Klassen verhalten sich im Hinblick auf Exceptions identisch: Bei programminternen Fehlern wird eine Assertion, bei Fehlern in einem empfangenen Paket hingegen eine RTPPacketException geworfen. Die Zusammenhänge zwischen den Klassen dieser Hierarchie können dem Klassendiagramm auf Seite 148 entnommen werden, die einzelnen RTP-Pakete sind

in 1.2.5 definiert. Im Folgenden werden die Klassen, in RTP-, RTCP- und APP-Pakete gegliedert, kurz einzeln betrachtet.

RTPBasePacket

Wie oben schon erwähnt, enthält die RTPBasePacket-Klasse den Speicherbereich, in dem sich die Rohdaten des Pakets befinden. Zusätzlich kann sich das Paket mit der richtigen DiffServ-Klasse selber versenden, bietet eine Reihe von Methoden zur Typenbestimmung von empfangenen Paketen und kann entweder einen bereits reservierten Speicherbereich übernehmen oder einen eigenen reservieren.

Klasse: RTPBasePacket	
Superklassen: -	
Subklassen: RTPPacket, RRPacket, SRPacket, SDESPacket, BYEPacket, APPPacket	
Enthält die Rohdaten eines Pakets.	`raw_data`
Kennt deren Länge.	`raw_len`
Kann diese Daten über ein Socket versenden.	`send(s)`, UPDSocket
Kennt den DiffServ-Typ, mit dem das Paket gesendet werden soll und setzt diesen vor dem Senden.	DSType Socket
Dieser Typ kann verändert werden.	`set_dstype(t)`
Hat eine Reihe von Methoden, die prüfen, welcher Pakettyp in `raw_data` gespeichert ist.	`is_*_packet()`
`raw_data` kann übernommen werden.	`RTPBasePacket(data,len)`
`raw_data` kann alloziert werden.	`RTPBasePacket(len)`
Gibt den Speicher nur frei, wenn er im Konstruktor alloziert wurde.	

RTP-Klassen

RTP-Pakete haben eine relativ statische Struktur und kennen keine Subtypen, weshalb dieser Ast des Vererbungsbaumes nur zwei Klassen enthält, RTPHeader und RTPPacket. RTPHeader deckt den statischen Teil des Pakets ab und ist in RTP-Packet enthalten.

Klasse: RTPHeader	
Superklassen: -	
Subklassen: -	
Deckt die Felder Version, Padding, Extension und CSRC-Count mit einem Bitfield ab.	
Deckt das Payload-Type-Feld ab.	
Bietet byteordnungsunabhängigen Zugriff auf Sequenznummer, Timestamp und SSRC.	
Hat eine Methode zum automatischen Einfüllen der Felder.	

RTPPacket enthält die RTPHeader und deckt zusätzlich die variabel grossen Teile des Pakets ab. Ausserdem bietet sie einige profilspezifische Methoden.

Klasse: RTPPacket	
Superklassen: RTPBasePacket	
Subklassen: -	

Deckt den Header des Pakets mit einem RTPHeader ab.
Hat Zeiger auf die CSRC-Liste und den Payload-Teil des Pakets.
Kennt ausserdem die Länge des Payload.
Hat einen Konstruktor für neue Pakete, der automatisch die Headerfelder ausfüllt.
Hat einen Konstruktor für gegebene Speicherbereiche.
Hat einen Konstruktor für neu empfangene Pakete.
Gibt zurück, ob und wie das Paket aufgeteilt ist und ob es den ersten oder zweiten Teil der Daten enthält.
	RTPHeader
	
	`RTPPacket(len,type, fields)`
	`RTPPacket(data,len)`
	`RTPPacket(npkt)`

RTCP-Klassen

Durch die Vielfalt verschiedener RTCP-Subtypen bedingt, gestaltet sich dieser Teil der Klassenhierarchie bedeutend komplizierter als der RTP-Teil. Auch hier kann der statische Header, den alle RTCP-Pakete enthalten, mit einer eigenen Klasse abgedeckt werden. Diese hat den Namen RTCPHeader.

Klasse: RTCPHeader	
Superklassen: -	
Subklassen: -	
Deckt die Felder Version, Padding und Chunk-Count mit einem Bitfield ab. Deckt das Payload-Type-Feld ab. Bietet byteordnungsunabhängigen Zugriff auf das Length-Feld. Hat eine Methode zum automatischen Einfüllen der Felder.	

RR-/SR-Pakete: RR- und SR-Pakete enthalten beide einen oder mehrere RR-Blöcke. In SR-Paketen kommt ein SR-Block dazu. Beide werden mit einer eigenen Klasse abgedeckt.

Klasse: RRBlock	
Superklassen: -	
Subklassen: -	
Bietet byteordnungsunabhängigen Zugriff auf alle im RRBlock enthaltenen Felder. Hat eine Methode zum automatischen Einfüllen der Felder.	

Klasse: SRBlock	
Superklassen: -	
Subklassen: -	
Bietet byteordnungsunabhängigen Zugriff auf alle im SRBlock enthaltenen Felder. Hat eine Methode zum automatischen Einfüllen der Felder.	

Durch diese beiden Klassen wird ein Grossteil der in RR- und SR-Paketen transportierten Information bereits abgedeckt. RRPacket und SRPacket delegieren folglich den grössten Teil ihrer Funktionalität an sie.

Klasse: RRPacket	
Superklassen: RTPBasePacket	
Subklassen: -	
Deckt den Header des Pakets mit einem RTCPHeader ab.	RTCPHeader
Hat einen Zeiger auf den ersten RRBlock.	RRBlock
Bietet byteordnungsunabhängigen Zugriff auf SSRC.	
Hat einen Konstruktor für neue Pakete, der automatisch die Headerfelder ausfüllt.	`RRPacket(blocks,ssrc)`
Hat einen Konstruktor für neu empfangene Pakete.	`RRPacket(npkt)`

Klasse: SRPacket	
Superklassen: RTPBasePacket	
Subklassen: -	
Deckt den Header des Pakets mit einem RTCPHeader ab.	RTCPHeader
Hat einen Zeiger auf den SR-Block.	SRBlock
Hat einen Zeiger auf den ersten RR-Block.	RRBlock
Bietet byteordnungsunabhängigen Zugriff auf SSRC.	
Hat einen Konstruktor für neue Pakete, der automatisch die Headerfelder ausfüllt.	`SRPacket(blocks,ssrc, fields)`
Hat einen Konstruktor für neu empfangene Pakete.	`SRPacket(npkt)`

SDES-Paket: Der komplexeste RTCP-Subtyp ist sicherlich SDES. Items variabler Länge können, in einer variablen Anzahl von Chunks verpackt, in einem Paket enthalten sein. Entsprechend hat SDESPacket zwei Hilfsklassen, SDESItem und SDESChunk. SDES-Pakete haben im Unterschied zu anderen RTCP-Paketen keine fest vorgegebene oder einfach zu berechnende Länge, weshalb die Hilfsklassen eigenen Speicher reservieren müssen. Die Speicherbereiche der Items können dann in den Chunks zu Blöcken und diese wiederum im Paket zusammengefasst werden.

In der umgekehrten Richtung ist der Speicher natürlich schon reserviert, was den Items und Chunks mitgeteilt werden muss, damit sie nicht versuchen, ihren Teil des Speicherblocks freizugeben.

Klasse: SDESItem	
Superklassen: -	
Subklassen: -	
Hat einen Speicherbereich mit den Rohdaten. Weiss, ob dieser freigegeben werden muss.	`data`
Bietet Zugriff auf Typ, Länge und Information.	`type(),info_length()` `info()`
Gibt die Länge der Daten zurück. Erfüllt Orthodox Canonical Form.	`length()`
Kann neu oder aus einem Puffer konstruiert werden.	`SDESItem(type,len,data)` `SDESItem(buffer)`

Auf der nächsthöheren Ebene befindet sich die SDESChunk-Klasse. Sie übernimmt die Aufgabe, beim Paketaufbau die einzelnen Items in einem Speicherblock zusammenzufassen. Umgekehrt parst sie die Chunks von empfangenen Paketen und erstellt eine Liste der darin enthaltenen Items.

Klasse: SDESChunk	
Superklassen: -	
Subklassen: -	
Hat einen Speicherbereich mit den Rohdaten und kennt dessen Länge.	`data` `length`
Weiss, ob dieser freigegeben werden muss.	
Hat eine Liste von Items.	SDESItem
Kombiniert diese und schreibt den resultierenden Block nach `data`.	
Bietet byteordnungsunabhängigen Zugriff auf SSRC.	
Kann aus einer Liste von Items erstellt werden.	`SDESChunk(ssrc,items)`
Erstellt die Item-Liste aus Rohdaten.	`SDESChunk(buffer)`
Hat einen Copykonstruktor.	

Mit Hilfe dieser Klassen gestaltet sich das Design von SDESPacket einfach.

Klasse: SDESPacket	
Superklassen: RTPBasePacket	
Subklassen: -	
Deckt den Header des Pakets mit einem RTCPHeader ab.	RTCPHeader
Hat eine Liste von Chunks.	SDESChunk
Hängt die Chunks zusammen und schreibt sie nach `raw_data`.	
Hat einen Konstruktor, der eine Liste von Chunks nimmt.	
Hat einen Konstruktor für neu empfangene Pakete.	`SDESPacket(npkt)`

BYE-Paket: Das einfachste RTCP-Paket ist das BYE-Paket, was sich auch auf die entsprechende Klasse auswirkt. Nebst einem SSRC-Identifikator enthält es nur einen String im Pascal-Stil (d.h. die Länge wird im ersten Byte gespeichert).

Klasse: BYEPacket	
Superklassen: RTPBasePacket	
Subklassen: -	
Deckt den Header des Pakets mit einem RTCPHeader ab.	RTCPHeader
Hat Zeiger auf den `reason`-String und die dazugehörige Länge.	
Kann aus einem C-String konstruiert werden.	`BYEPacket(ssrc,reason)`
Hat einen Konstruktor für neu empfangene Pakete.	`BYEPacket(npkt)`

APP-Klassen

Eine besondere Rolle übernehmen die applikationsspezifischen APP-Pakete. Bis auf einen die Applikation identifizierenden, 4 Byte langen String kann der Inhalt frei definiert werden. Das Feld `chunk_count` im RTCP-Header wird bei APP-Paketen als Subtyp-Identifikator interpretiert. Die in DSPhone verwendeten APP-Subtypen sind in Abschnitt 4.1.1.3 definiert.
Die Klasse APPPacket hat in der APP-Hierarchie eine ähnliche Aufgabe, wie sie RTPBasePacket für die gesamte Hierarchie erfüllt. Sie definiert ein Grundmuster für den Paketaufbau und überlässt die Details ihren Erben. Ausserdem enthält sie eine Reihe von Methoden, mit denen der Subtyp eines empfangenen APP-Pakets bestimmt werden kann.

Klasse: APPPacket	
Superklassen: RTPBasePacket	
Subklassen: Ping	
Deckt den Header des Pakets mit einem RTCPHeader ab.	RTCPHeader
Bietet byteordnungsunabhängigen Zugriff auf SSRC.	
Bietet Zugriff auf den Applikations-ID-String.	
Reserviert im Konstruktor Platz für die Daten des Subtyps und setzt `ssrc` und `app_name`.	`BYEPacket(ssrc, subtype,data_len`
Hat einen Konstruktor für neu empfangene Pakete.	`BYEPacket(npkt)`
Kann feststellen, um welchen Subtypen es sich handelt.	

Das in DSPhone benutzte RTP-Profil definiert zwei APP-Subtypen, Ping und Ping-Reply, deren Aufbau identisch ist. Es ist daher ein logischer Schritt, beide Subtypen in einer Klasse zusammenzufassen, die gefragt werden kann, ob sie ein Ping- oder ein Ping-Reply-Paket repräsentiert. Diese Klasse hat den nicht ganz korrekten Namen Ping.

Klasse: Ping	
Superklassen: APPPacket	
Subklassen: -	
Hat Zeiger auf die Ping-Felder und bietet byteordnungsunabhängigen Zugriff auf sie.	
Kodiert den verwendeten DiffServ-Typ.	DSType
Speichert NTP-Timestamps.	Time
Gibt zurück, welcher Subtyp repräsentiert wird.	`is_request(), is_reply()`

Hat Konstruktor für neue Ping-Pakete, für auf ein Ping antwortende Ping-Replies und für neu empfangene Pakete.	`Ping(ssrc,id,dstype)`
	`Ping(ssrc,ping)`
	`Ping(npkt:APPPacket)`

4.3 Bemerkungen zu Design und Implementation

4.3.1 Stand der Implementierung

Der grösste Teil des im obigen Abschnitt vorgestellten Designs ist vollständig implementiert. Die Ausnahme bildet das graphische Benutzerinterface, das zwar weitgehend implementiert ist, aber Lücken aufweist. Während der Testphase wurden ausserdem an verschiedenen Stellen im Programm Veränderungen vorgenommen, so dass Details der tatsächlichen Implementierung vom Design abweichen können. So existiert die direkte Assoziation zwischen Socket und DSType (siehe Abb. 4.23) im Code nicht, sondern geht über die RTPBasePacket-Klasse. Andere Unterschiede sind abweichende Methodenparameter und einige im Design nicht vermerkte lokale Funktionen.
Einige Teile des Programms sind noch instabil und ungenügend getestet. Die folgenden Abschnitte behandeln einzelne Teile von DSPhone und sind weiter in *Stand*, *Bekannte Fehler* und *Mögliche Verbesserungen* unterteilt.

4.3.1.1 Basissystem

Stand

Das Textmodus-Interface ist vollständig implementiert und hat sich während der Tests als stabil erwiesen. Auch die Kompilierungs- und Laufzeitkonfiguration läuft befriedigend.
Die graphische Benutzerschnittstelle ist zu grossen Teilen realisiert. Allerdings sind nicht alle Subsysteme vollständig graphisch konfigurierbar.

Bekannte Probleme

Durch Inkompatibilitäten von Gtk-- mit der pthread-Bibliothek kommt es gelegentlich zu Hängern und Programmabstürzen. Ausserdem ist die Speicherverwaltung der GUI-Klassen nicht ganz fehlerfrei, was aber teilweise nachweislich auf die Grafikbibliothek zurückzuführen ist.

Mögliche Verbesserungen

Ein flexibleres Design könnte die Abhängigkeit von der Grafikbibliothek vermindern und damit deren Instabilitäten besser isolieren. Das könnte eventuell das Abspalten des Interface in einen eigenen Prozess bedingen.

4.3.1.2 Backend

Stand

Alle Backend-Klassen sind vollständig implementiert und getestet, allerdings nicht ganz fehlerbereinigt.

Bekannte Probleme

In den Duplex-Modi kommt es nach einer gewissen Zeit zu Deadlocks, was wahrscheinlich auf einen Synchronisationsfehler zwischen StreamSplitter und Mixer oder Protocol zurückzuführen ist. Durch die etwas unklare Trennung zwischen Backend und Protokoll-Subsystem kann dieser Fehler auch nicht mit Sicherheit im Backend isoliert werden.

Ein beim Design nicht berücksichtigter Faktor war, dass Audiogeräte nicht beliebige Chunkgrössen unterstützen. Die tatsächliche Chunkgrösse muss dadurch auf umständliche Weise den Backendklassen mitgeteilt werden, um unregelmässigen Datenfluss zu vermeiden, der durch unterschiedliche Grössen entsteht.

Mögliche Verbesserungen

Um die Struktur des Programms zu vereinfachen und die Fehlersuche zu erleichtern, sollten geeignetere Schnittstellen zu den angrenzenden Subsystemen gesucht werden. Problematisch ist vor allem die unklare Abgrenzung zwischen Backend und Protokoll-Subsystem, da diese durch das Erben von Protocol durchbrochen werden kann.
Die Datenstrom-Orientierung des Backends führt schnell zu Timingproblemen, die mit einem mehr auf Datenblöcken basierenden Design besser gelöst werden

könnten. Solche Blöcke könnten zusätzlich einen Verweis auf ihr Format enthalten, wodurch sich auch in dieser Hinsicht eine Flexibilisierung ergäbe.

4.3.1.3 Audio-Subsystem

Stand

Alle Interface-Klassen ausser OSS sind implementiert. Wirklich getestet wurden aber nur FileIface und NullIface, da sie in den Experimenten Verwendung fanden. AlsaIface und EsoundIface haben sich als lauffähig erwiesen, können aber mangels eingehender Tests nicht als stabil bezeichnet werden. LoopBackIface kam nie zum Einsatz und ist ungetestet.
Die beiden Audioformat-Klassen sind implementiert und hinreichend getestet. Die Konvertierungsroutinen scheinen fehlerfrei zu laufen.

Bekannte Probleme

Wie auch im Backend war das Design nicht dafür vorgesehen, dass Audiogeräte gewisse Chunkgrössen erzwingen können, weshalb das Interface nicht in der geplanten Art verwirklicht werden konnte.

Mögliche Verbesserungen

Das Design dieses Subsystems hat sich als stabil und genügend flexibel erwiesen. Im Bereich der Formatkonvertierung könnten allerdings überflüssige Abhängigkeiten entfernt werden: Jedes Format muss sich momentan in jedes andere konvertieren können, was sie voneinander abhängig macht. Durch eine Factoryklasse, die die jeweils benötigten Konvertierungsalgorithmen in Form von Funktionsobjekten produziert, könnte das Hinzufügen von neuen Algorithmen vereinfacht werden. Allerdings müsste dann zu Laufzeit für jede mögliche Konvertierung überprüft werden, ob sie unterstützt wird.

4.3.1.4 RTP-Subsystem

Stand

Alle Klassen im RTP-Subsystem wurden implementiert und sind getestet. Die Routinen zur Paketaufteilung und -vereinigung haben allerdings nur wenige Tests absolviert. Leider konnten nicht alle Fehler gefunden werden, was sich aber auf die Experimente nicht auswirken kann, da diese Fehler erst nach Beendigung der Verbindung auftreten.

Bekannte Probleme

Bei der Deallokation des Subsystems, im Besonderen der Threads, kommt es gelegentlich zu Abstürzen. Dies rührt wahrscheinlich daher, dass die Threads nicht sauber beendet werden, entweder aufgrund eines Fehlers in der pthread-Bibliothek oder durch inkorrekte Benutzung der API.

Die unsaubere Trennung zwischen Backend und diesem Subsystem wirkt sich auch hier aus, vor allem in Form einer unklaren Verantwortlichkeitsverteilung zwischen Protocol und RTP. Beispielsweise sollte es nicht Aufgabe von RTP sein, zwischen StreamSplitter und AudioIface zu wählen, wenn Daten vom Backend gelesen werden. Dieses Problem wird “Tight Coupling” genannt.

Das interne Design des Subsystems ist auch nicht ganz ausgereift. Durch das Einführen von Scout-Objekten wurde die Trennung zwischen Policer und RTP-Strategy durchbrochen, was auch die Synchronisation zwischen Sender- und Receiver-Thread erschwert. Auch der Umweg, den ankommende RTP-Pakete über die RTPStrategy-Klasse nehmen müssen, ist unschön.

Da die fehlerhafte Jitter-Berechnung in der ServiceScout-Klasse erst lange nach den Tests erkannt und korrigiert wurde, blieb keine Zeit, die korrigierte Version wiederum zu testen. Damit der Code als stabil gelten kann, sollte das noch nachgeholt werden.

Mögliche Verbesserungen

Wie oben erwähnt, sollte die Schnittstelle zwischen Backend und RTP-Subsystem klarer definiert sein. Vor allem die Eigenheiten der Backend-Modi sollten nur im Backend selber Auswirkungen haben.
Die internen Designfehler könnten durch folgende Änderungen verbessert werden: Das RTPStats-Objekt sollte nicht Teil von RTPStrategy sein, sondern allen Klassen des Subsystems zur Verfügung stehen. Somit könnten die Zuständigkeiten zwischen RTPInQueue und RTPStrategy in ankommende und zu sendende Pakete unterteilt werden. Das Scout-Problem kann gelöst werden, indem das Erstellen von Probe-Paketen den Verantwortlichkeiten von Policer hinzugefügt wird. Somit wird für RTPStrategy transparent, welcher Scout gerade aktiv ist.

4.3.1.5 Hilfsklassen

Stand

Alle im entsprechenden Abschnitt aufgeführten Klassen wurden implementiert und getestet. Socket erfüllt aber die vom Design geforderten Aufgaben nicht ganz, da sie zwar ein Interface zum Ändern der DiffServ-Parameter bietet, aber keine DSType-Objekte akzeptiert.

Bekannte Probleme

Das Interface von Socket genügt den Designanforderungen nicht und schiebt die Aufgabe, die in einem DSType abgelegten Informationen zu setzen, auf RTPBasePacket ab. Der Grund, warum dies bisher nicht geändert wurde ist, dass die Socket-Klassen in ihrer jetztigen Form sehr stabil funktionieren, eine Eigenschaft, die durch Veränderungen leicht zunichte gemacht wird.

Beim Setzen von DiffServ-Klassen durch das Socket tritt ein weiteres Problem auf: Der Linuxkernel verbietet Benutzern ohne Netzwerkadministrator-Status, von der ursprünglichen IPv4-Spezifikation abweichende TOS-Werte zu setzen. DSPhone muss also mit entsprechenden Rechten gestartet werden. Alternativ dazu kann der `setsockopt`-Code im Kernel entsprechend verändert werden, was aber alles andere als elegant wäre.

Weiter ist es nicht ganz sicher, ob die Implementierung der MThread-Klasse die pthread-API korrekt benutzt, da beim Terminieren von Threads des Öfteren seltsames Verhalten auftritt.

Mögliche Verbesserungen

Das erste der erwähnten Probleme ist relativ einfach zu beheben, was aber aus den oben erwähnten Gründen nicht geschehen ist. Da UDPSocket für die allgemeine Verwendung jedoch zu spezialisiert ist — sie unterstützt nur ein Zielsystem — sollten die Socket-Klassen unabhängig davon redesignt werden.
Weiter ist die Art, wie Konfigurationswerte in Config abgelegt werden, nicht typensicher und kapselt die Implementierungsdetails zuwenig ab. Die flache Struktur der Variablen könnte ausserdem durch eine Baumstruktur ersetzt werden.

4.3.1.6 Gesamtprogramm

Das gesamte Programm enthält ca. 10000 Zeilen Code, reine Kommentar- und Leerzeilen nicht mitgerechnet. Die Quelldateien sind ausserdem alle im selben Verzeichnis abgelegt, zusammen mit einigen Hilfsprogrammen, weshalb sich durch eine durchdachtere Verzeichnisstruktur das Verwalten der Quellen sicherlich vereinfachen würde. Konfiguration, Kompilieren und Installation des Programms werden zwar durch die verwendeten Autoconf-Skripten sehr vereinfacht, sind aber nicht vollständig automatisiert.

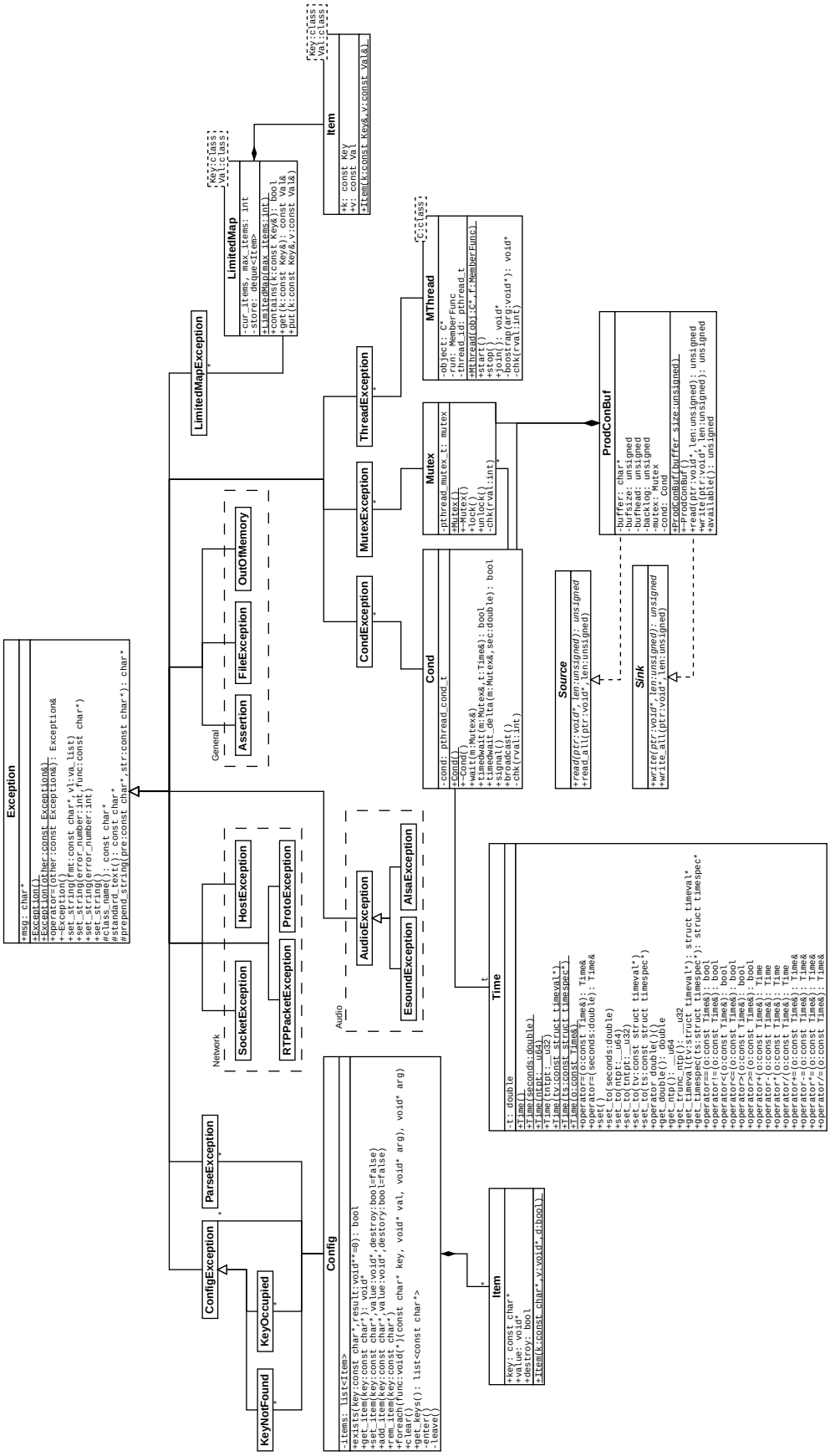


Abbildung 4.22: Diagramm der Hilfsklassen

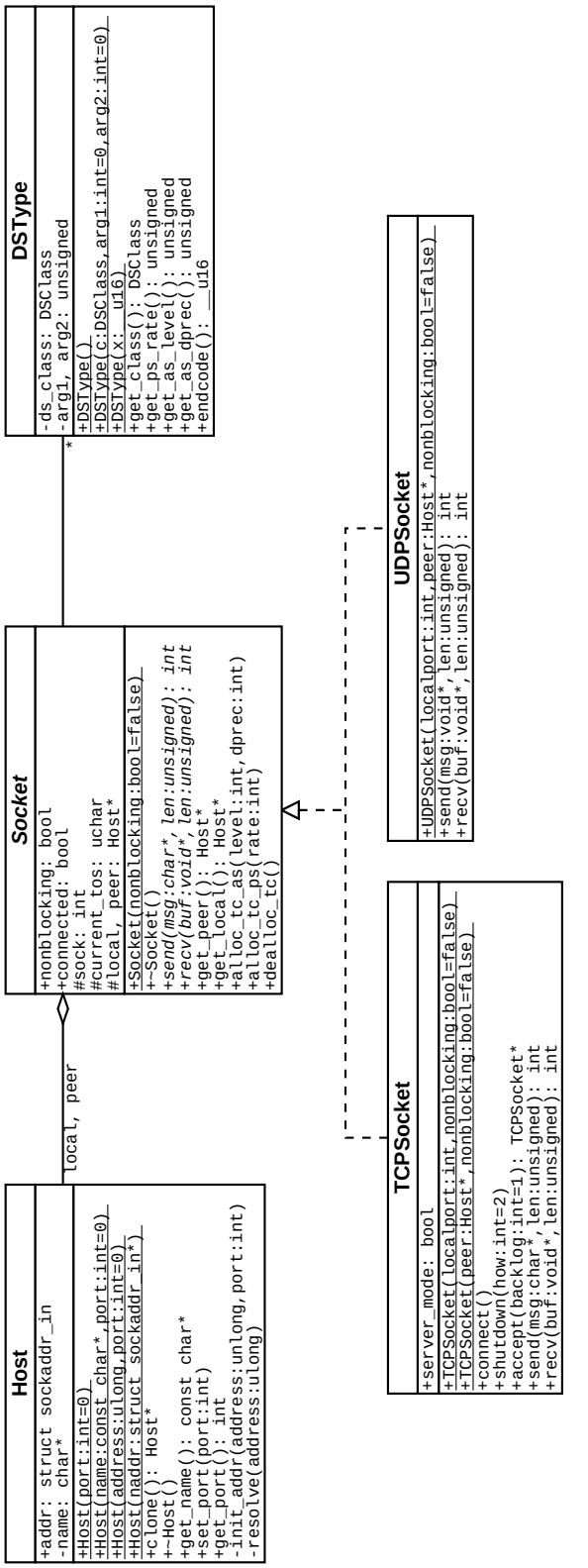


Abbildung 4.23: Diagramm der Netzwerk-Klassen

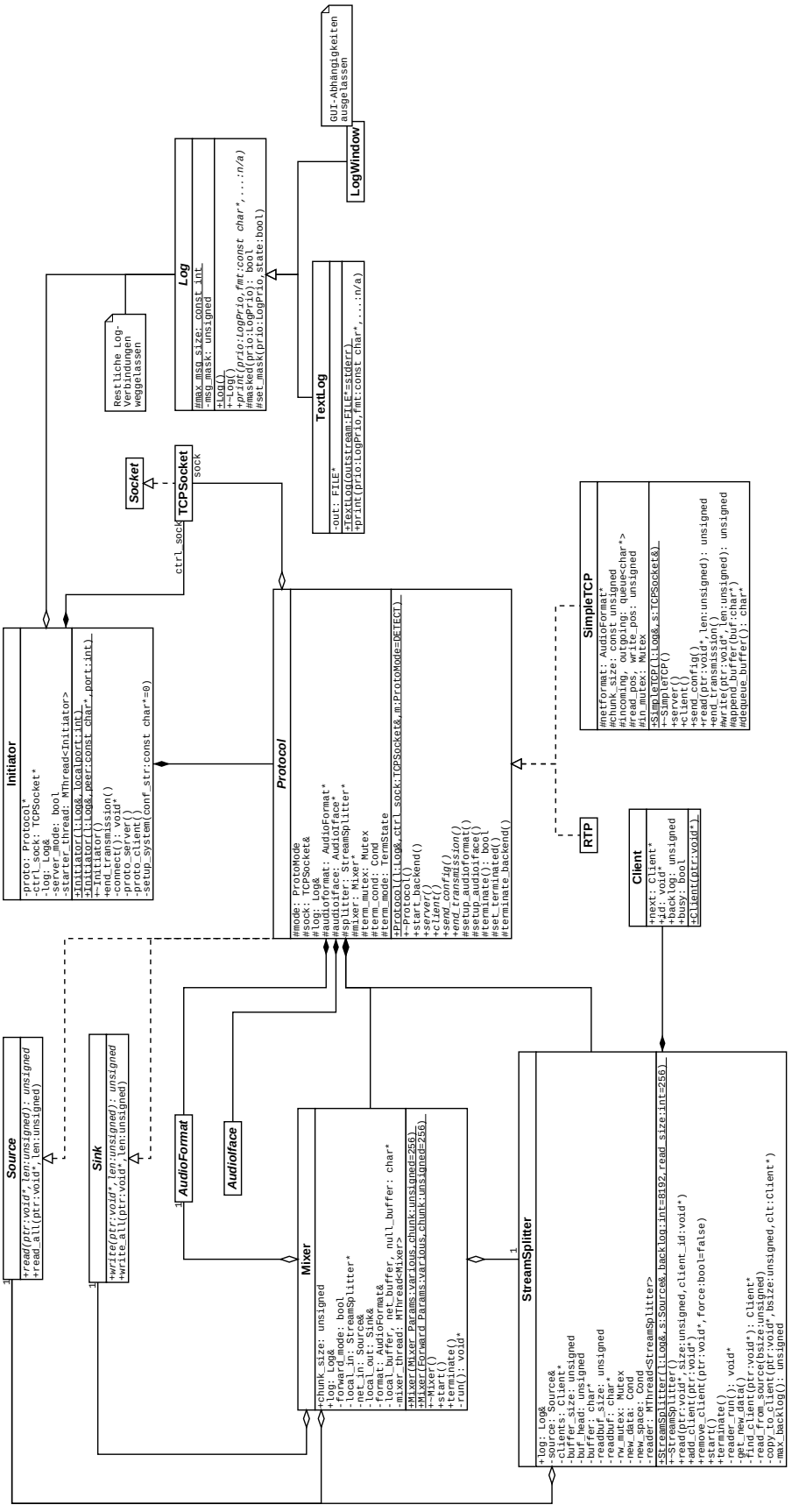


Abbildung 4.24: Klassendiagramm des Backends

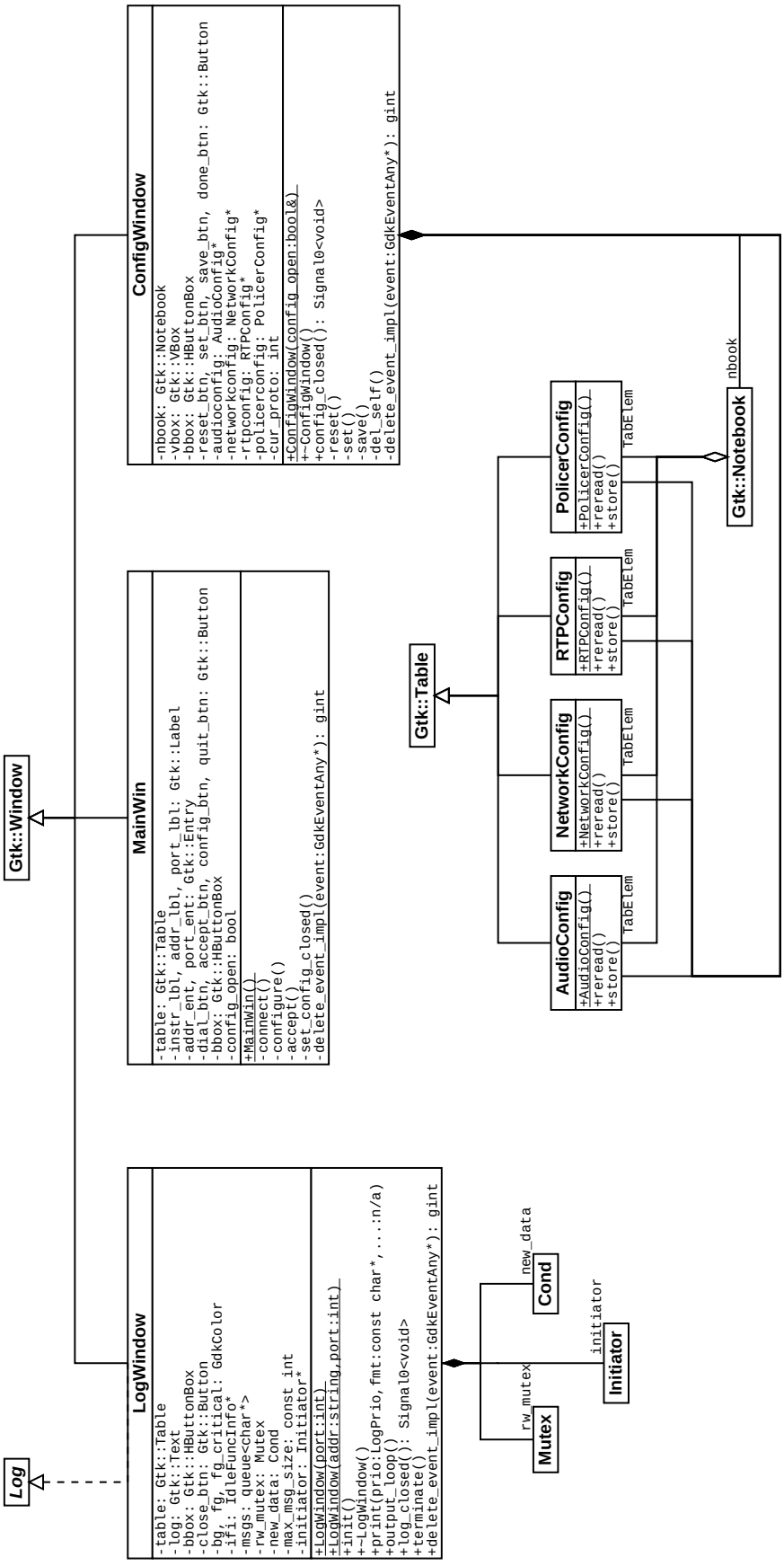


Abbildung 4.25: Klassendiagramm der graphischen Benutzerschnittstelle

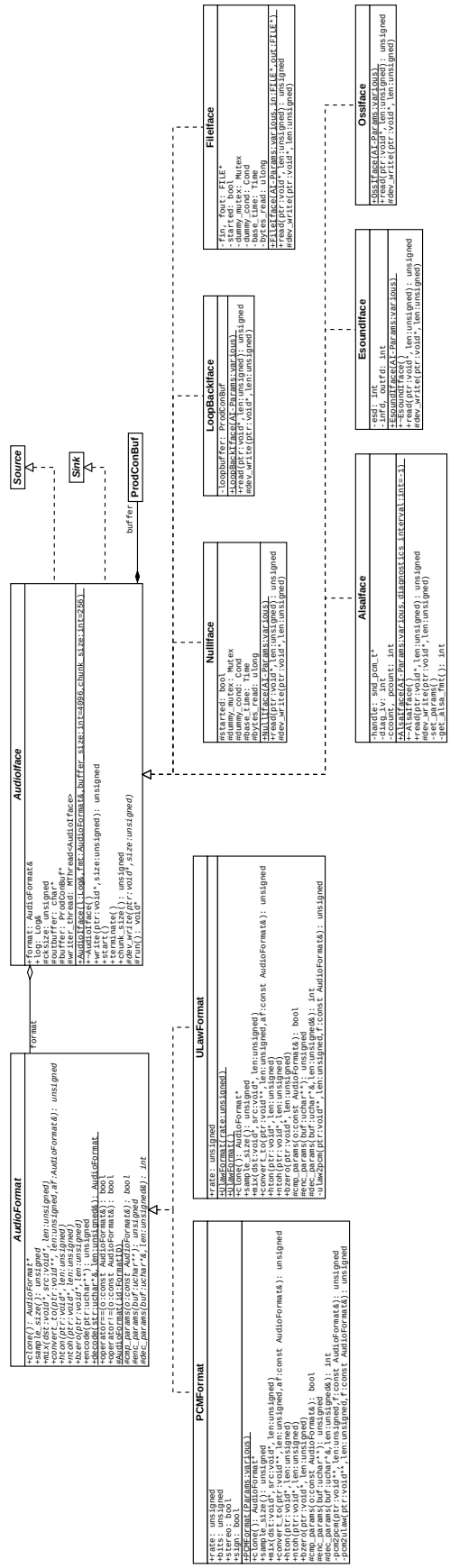


Abbildung 4.26: Klassendiagramm des Audio-Subsystems

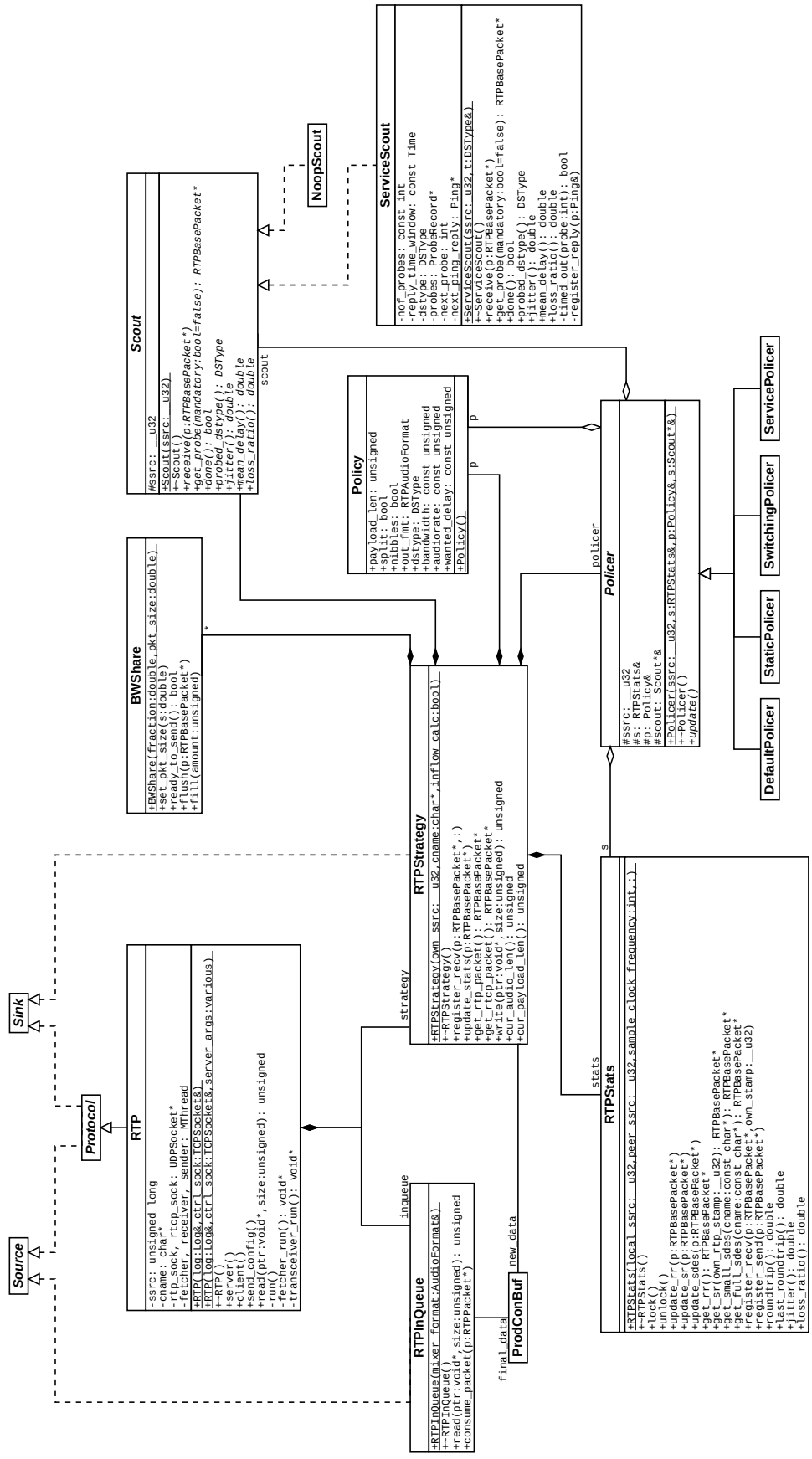


Abbildung 4.27: Klassendiagramm des RTP-Subsystems

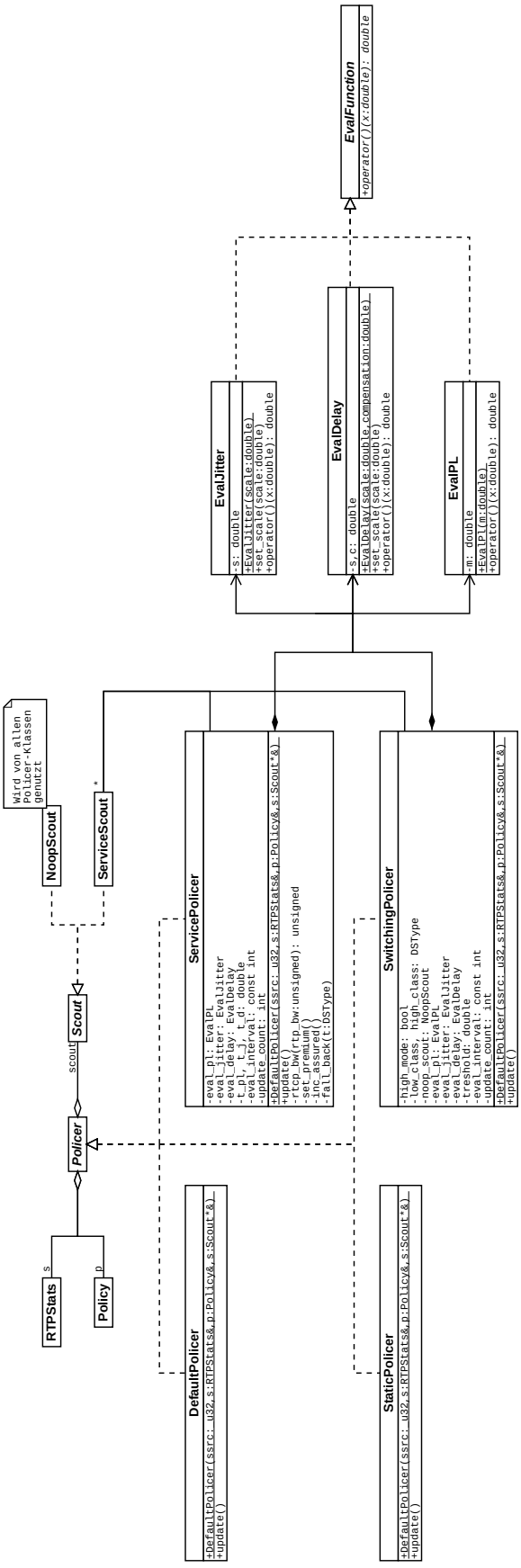


Abbildung 4.28: Policer- und EvalFunction-Klassen

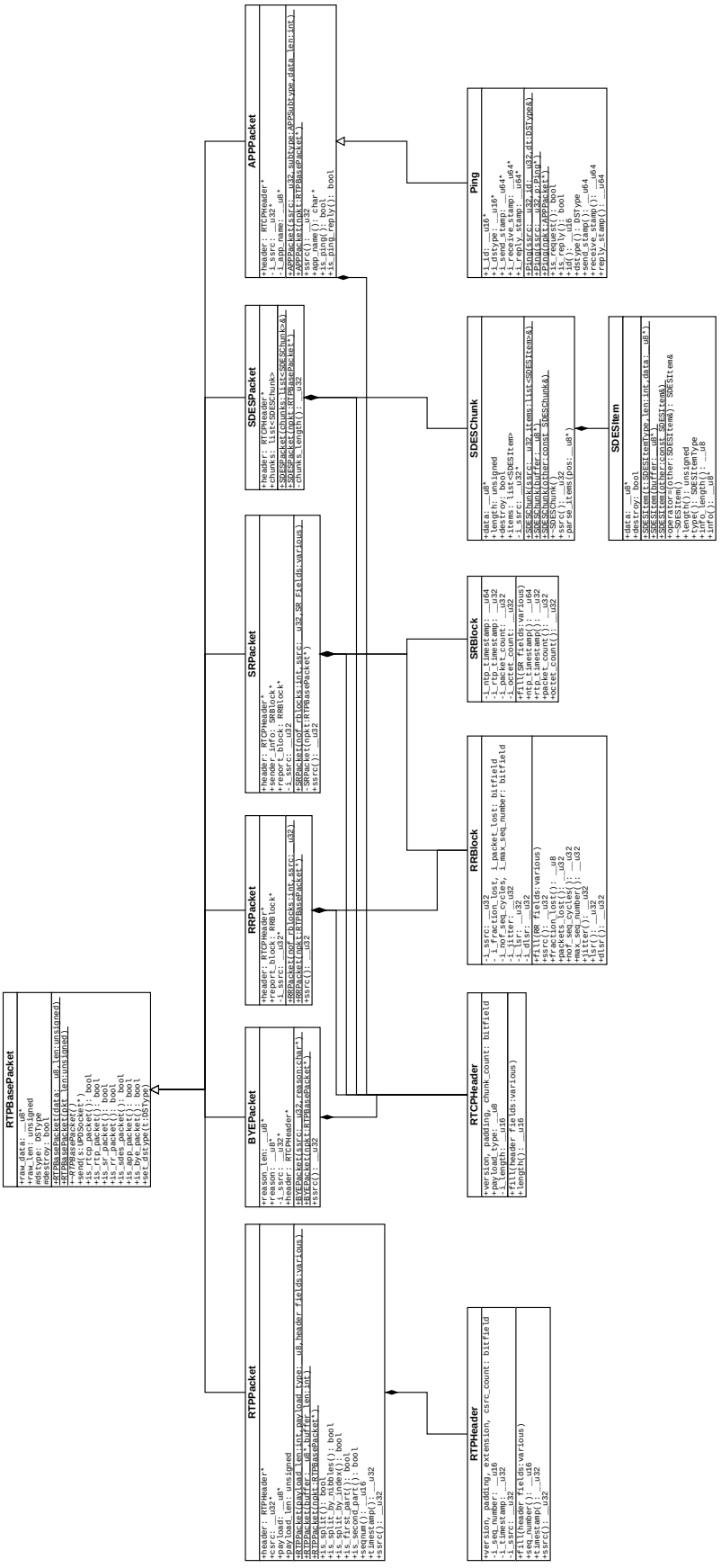


Abbildung 4.29: Die RTP-Paket-Klassen

Kapitel 5

Leistungsbewertung

Um das Verhalten von DSPhone zu überprüfen, wurden Tests für jeden der implementierten Policer durchgeführt. Ausserdem wurde mit weiteren Tests sichergestellt, dass sich UDPgen den Spezifikationen entsprechend verhält.
Die bei den Tests berücksichtigten Werte sind die vom Empfänger per RR-Paket gemeldeten Variablen Verlustrate und Jitter, der in Situationsbewertungen verwendete Verlustraten-Schätzer und die nach dem Verfahren von Seite 25 berechnete Umlaufzeit (Roundtrip).
In den Tests von Switching- und ServicePolicer kommen die Resultate der Bewertungsfunktionen, der Sondenpaket-Messreihen und wiederum deren Bewertungen hinzu.

5.1 Aufbau

DSPhone wurde in einem IP-Netzwerk mit folgendem Aufbau getestet: Die beiden Endpunkte sind über drei DiffServ-Router miteinander verbunden, die von einem externen System, das mit allen drei Routern einzeln verbunden ist, zusätzlich mit UDP-Paketen belastet werden. Abb. 5.1 zeigt diese Topologie. Wie daraus ersichtlich ist, kann der Störsender (Challenger) über drei verschiedene Routen Pakete an das Ziel (Enterprise) senden, was durch zwei Alias-Adressen und entsprechende Routingtabellen in Challenger bewerkstelligt wird. Alle Systeme in diesem Netz sind mit 100 Mbit/s Ethernet im Duplexbetrieb untereinander verbunden.

Auf allen Systemen läuft der mit DiffServ-Unterstützung erweiterte Linux-Kern 2.2.17, allerdings mit verschiedenen DiffServ-Konfigurationen: Atlantis und Endeavour, als zweiter und dritter Hop, weisen die typische Konfiguration eines inneren DS-Knotens auf (siehe Abb. 5.2). Discovery, als erster Hop, ist wenn nötig

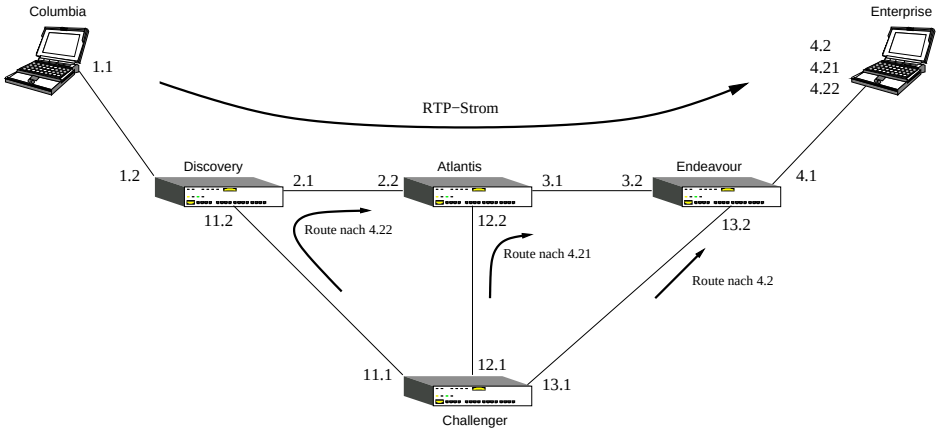


Abbildung 5.1: Topologie des Testnetzes

als Ingress-Knoten konfiguriert (Abb. 5.3). In allen anderen Fällen hat auch dieser Router die Konfiguration eines inneren Knotens. Auf Quelle, Ziel und Störsender hingegen sind gar keine DiffServ-Module geladen.

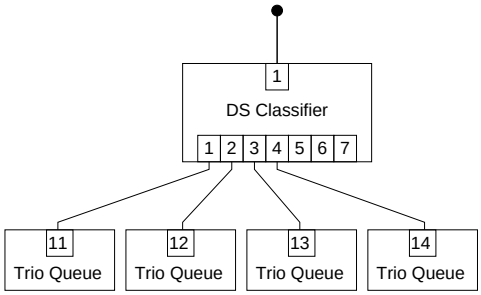


Abbildung 5.2: DiffServ-Konfiguration der inneren Knoten

Während der Tests wurde ein unidirektionaler Datenfluss mit Audiodaten im μ-Law-Format bei 8000Hz Abtastrate und mit einem Sendeintervall von 30ms von Columbia zu Enterprise geschickt, der von Challenger aus mit von UDPgen erzeugten UDP-Paketen auf allen drei Zwischen-Hops gestört wurde.

Die für die folgenden Tests verwendeten Konfigurationsdateien und -skripte sind im Anhang A zu finden.

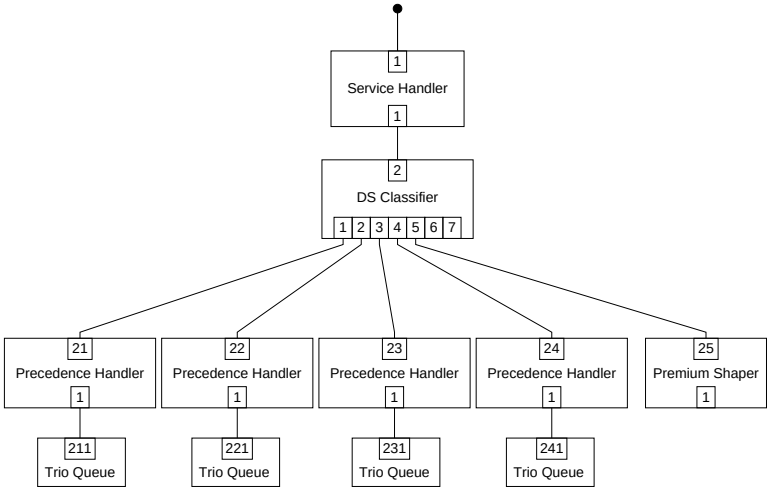


Abbildung 5.3: DiffServ-Konfiguration des Ingress-Knotens

5.2 UDPgen

Da UDPgen in allen folgenden Tests als Störverkehr-Generator zum Einsatz kam, musste zuerst das Verhalten dieses Programms verifiziert werden.

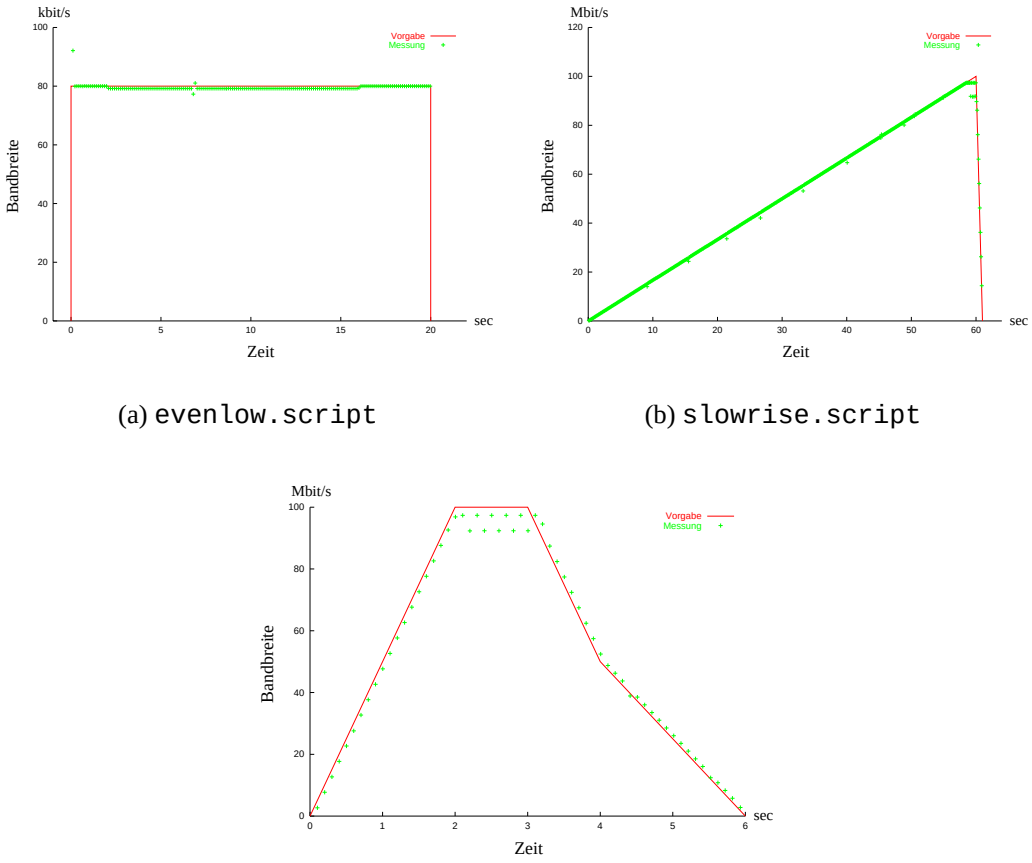
5.2.1 Aufbau

Zum Testen von UDPgen wurden anhand dreier Skripte UDP-Pakete an ein benachbartes System gesendet, wo UDPrcv die ankommende Bandbreite aufzeichnete. Durch einen Vergleich der Vorgaben mit den tatsächlich von UDPrcv registrierten Bandbreiten kann, mit Einschränkungen, auf die Genauigkeit von UDPgen geschlossen werden. Die drei Skripte erzeugen einen gleichmässigen Strom mit kleiner Bandbreite, eine kurze Spitze und eine innert 60 Sekunden anschwellende Spitze. Letztere eignet sich dazu, Ereignisse zu analysieren, die bei bestimmten Bandbreiten auftreten; beispielsweise den Zeitpunkt, wenn ein Router seine Leistungsgrenze erreicht.

5.2.2 Resultate

Wie Abb. 5.4 zeigt, stimmt die empfangene Bandbreite im Allgemeinen sehr gut mit dem vorgegebenen Verlauf überein. Das Verhalten an der physikalischen Bandbreitengrenze ist allerdings etwas instabil, was aber angesichts der sonst guten Resultate mit grosser Wahrscheinlichkeit auf das Betriebssystem oder die Hardware zurückzuführen ist. Bei genauerem Hinschauen fallen einige kleinere Unregelmässigkeiten auf, deren Ursache leider nicht klar ist.

Wenn man das Verhalten als Ganzes betrachtet erscheint der Einsatz von UDPgen bei den folgenden Tests aber als gerechtfertigt.



(a) `evenlow.script` (b) `slowrise.script`

(c) `shortpeak.script`

Abbildung 5.4: Vergleich der vorgegebenen Verläufe mit den Messungen

5.3 DefaultPolicer

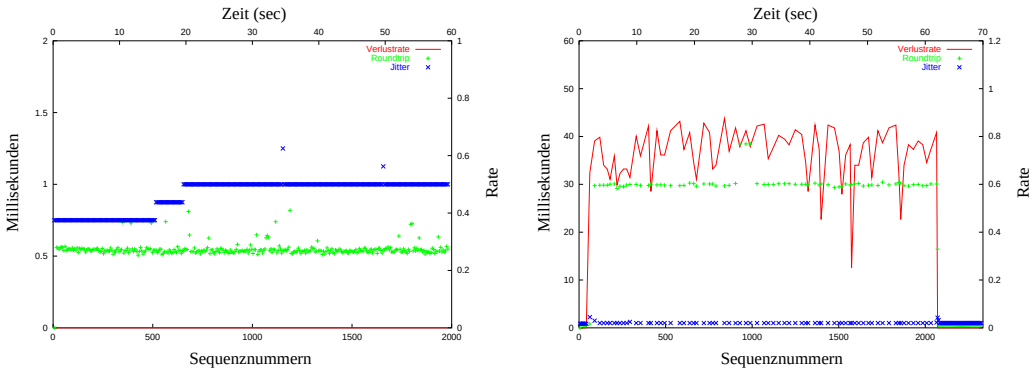
Um die Resultate der folgenden Tests korrekt interpretieren zu können, musste das Verhalten von DSPhone ohne DiffServ-Benutzung, d.h. mit dem DefaultPolicer, ermittelt werden, um Referenzwerte zu erhalten. Dies geschah durch drei verschiedene Versuche, bei denen die Verbindung auf unterschiedliche Weise belastet wurde.

5.3.1 Aufbau

Der Bandbreitenverlauf des Störverkehrs, den Challenger sendete, war für alle drei Router identisch. Im ersten Versuch wurden gar keine Störpakete gesendet, im zweiten war das Netz 60 Sekunden lang vollständig mit UDP-Paketen überflutet. Im dritten Versuch sendeten drei UDPgen-Prozesse mit langsam anschwellender Bandbreite, die während 60 Sekunden von 0 auf 100 Mbit/s stieg, um dann innerhalb einer Sekunde wieder auf 0 zurückzufallen.

5.3.2 Resultate

Im Fall einer störungsfreien Verbindung (siehe Abb. 5.5(a)) ist die Übertragung sehr gut: Kein einziges Paket ging verloren und die Roundtrip-Zeit bewegt sich im Bereich einer halben Millisekunde. Interessanterweise sind die Verzögerungsschwankungen bis zu viermal so gross wie die Verzögerung, ca. die halbe Roundtrip-Zeit, selbst. Offenbar entstehen diese Schwankungen also nicht im Netzwerk, sondern in den Endpunkten, etwa indem ankommende Pakete von der Anwendung nicht in regelmässigen Abständen akzeptiert werden. Andererseits kann es auch sein, dass die geringe Genauigkeit der Jitter-Berechnung in Bereichen extrem kleiner Verzögerung zu Ungenauigkeiten führt (Bei der hier verwendeten Abtastrate von 8000 Hz ist die Auflösung nur 1/8 Millisekunden).



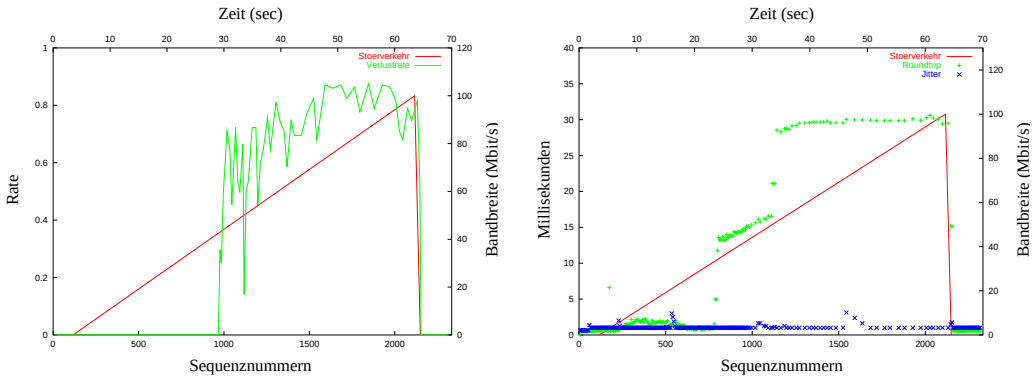
(a) kein Störverkehr

(b) maximaler Störverkehr

Abbildung 5.5: DefaultPolicer bei minimalem und maximalem Störverkehr

Im umgekehrten Fall (siehe Abb. 5.5(b)), einem vollständig überlasteten Netz, fällt vor allem der unregelmässige Verlauf der Paketverlustrate auf. Durchschnittlich gingen etwa 65%, im Minimum etwa 23% und im Maximum etwa 84%

der Pakete verloren. Hier scheint ein Zusammenhang zu den oben beschriebenen Schwankungen bei UDPgen, die auch im Bereich des physikalischen Maximums auftraten, zu bestehen. Entweder lösten die Schwankungen von UDPgen jene in diesem Versuch aus, oder es unterliegen beide Programme, DSPhone und UDPgen, demselben Phänomen.
Die Roundtrip-Zeit stieg im Vergleich zum unbelasteten Netz stark, von einer halben auf 30 Millisekunden. Rein rechnerisch war eine Einweg-Verzögerung von 24.8 msec zu erwarten (UDP-Pakete mit 1400 Bytes Nutzlast, Queuelängen von 100 Paketen und zwei ausgelastete Router), was eine Differenz von etwas mehr als 5 msec ergibt. Diese erklärt sich aus dem an der Leistungsgrenze arbeitenden ersten Router und der Übertragungszeit der zurückgesendeten RR-Pakete. Die Verzögerungsschwankungen verdoppelten sich auf 2 msec.



(a) Entwicklung der Verlustrate

(b) Entwicklung von Verzögerung und Jitter

Abbildung 5.6: DefaultPolicer bei langsam anschwellendem Störverkehr

Der dritte Test (stetig ansteigender Störverkehr) ergab die in Abb. 5.6 dargestellten Resultate. Im allgemeinen stimmen diese mit den zu erwartenden Resulaten überein, Verlustrate und Roundtrip-Zeit verhalten sich aber auffallend “sprunghaft”. Die Roundtrip-Zeit stieg in zwei grossen Flanken auf etwa 30 msec und blieb dann konstant, die Verlustrate sprang zwischen diesen Flanken auf etwa 60%, um dann allmählich auf 80% zu steigen.
Vergleicht man die Verläufe mit den Auslastungsverläufen der einzelnen Router (siehe Abb. 5.7), erkennt man sofort einige Übereinstimmungen. Die beiden Flanken der Roundtrip-Zeit fallen genau mit den Zeitpunkten zusammen, an denen die Bandbreite der ankommenden Pakete die Ausgangsbandbreite von Atlantis bzw. Endeavour übersteigt, sich also die Queues zu füllen beginnen. Wenn sich die

Queue von Endeavour gefüllt hat, beginnt er damit, Pakete zu verwerfen. Dadurch erklärt sich der steile Anstieg der Verlustrate zu diesem Zeitpunkt.
Offensichtlich steigt die Verzögerung an, bevor der Router beginnt, Pakete zu verwerfen. Somit könnte bei SwitchingPolicer und ServicePolicer (siehe unten) der Paketverlust fast vollständig eliminiert werden, indem nicht nur die Verzögerung selbst, sondern auch deren Veränderung in die Entscheidung zu einem Klassenwechsel einbezogen würde.

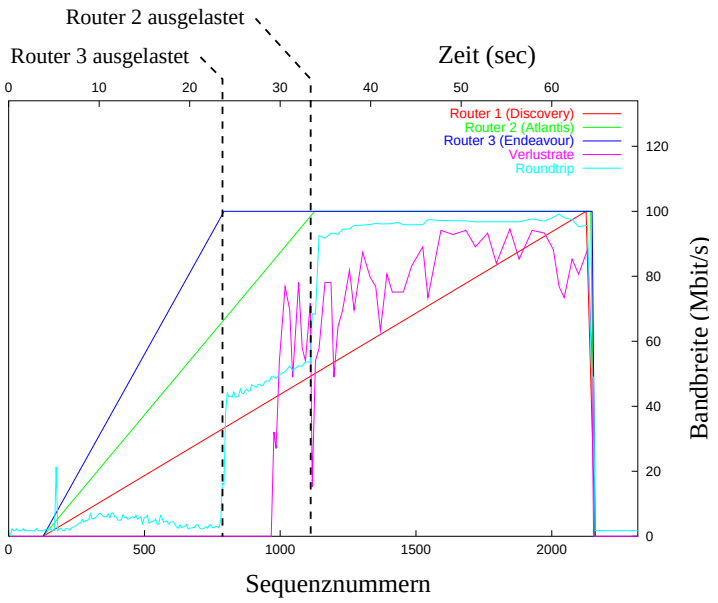


Abbildung 5.7: Verlustrate und Roundtrip im Vergleich zum Router-Durchsatz

5.4 StaticPolicer

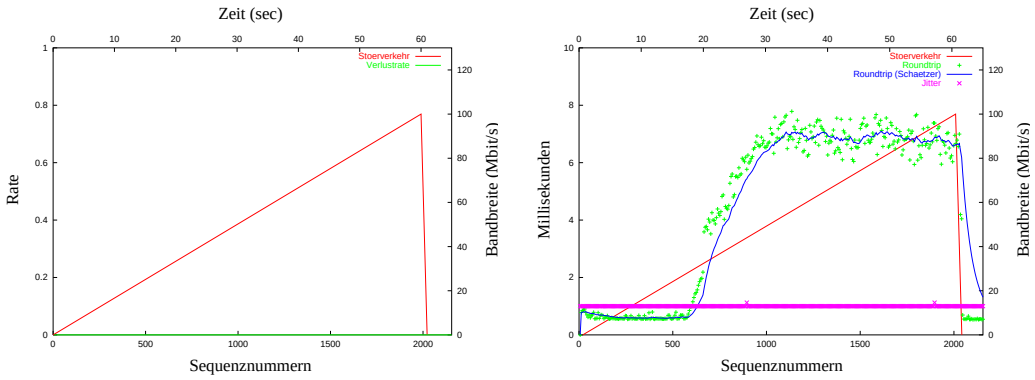
Nachdem in den vorherigen beiden Abschnitten das Verhalten von DSPhone und Testumgebung analysiert worden ist, folgt mit den Tests von StaticPolicer der erste Einsatz der DiffServ-Funktionalität von DSPhone. Das Ziel ist, zu ermitteln, wie gut ein Per-Hop-Behaviour höherer Ordnung gegen einen Datenstau in tieferen Serviceklassen schützt.

5.4.1 Aufbau

Wie im dritten Test von DefaultPolicer wurden die drei Router mit identischer, langsam ansteigender Bandbreite gestört. Der RTP-Strom wurde aber mit Assured bzw. Expedited Forwarding gesendet, der Störverkehr mit Best Effort.

5.4.2 Resulate

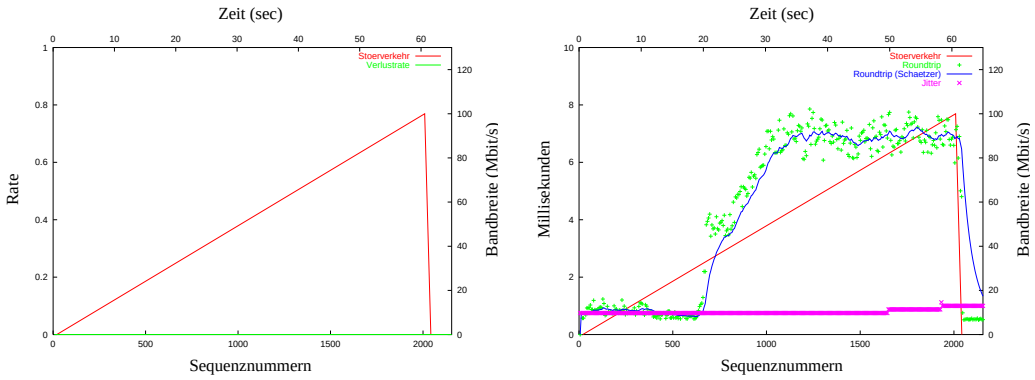
Wie aufgrund früherer Messungen anzunehmen war ([Stattb]), ging weder bei Assured noch bei Expedited Forwarding ein einziges Paket verloren (siehe Abb. 5.8(a) und Abb. 5.9(a)). Auch der Jitter blieb stabil auf tiefem Niveau. Erstaunlich ist hingegen der markante Anstieg der Roundtrip-Zeit, der etwa dann einsetzt, wenn Endeavour (der dritte Router) seine Auslastungsgrenze erreicht und der, was speziell erstaunt, beide Serviceklassen in gleicher Weise betrifft, obwohl sie sehr verschiedene PHBs aufweisen (siehe Abb. 5.8(b) und Abb. 5.9(b)). Möglicherweise liegt hier ein Timing-Problem auf dem Zielsystem vor. Bei einer grossen Anzahl ankommender Pakete können die aktiven UDPrcv-Prozesse den Zeitpunkt verzögern, an dem DSPhone ein SR-Paket in Empfang nehmen kann, was sich direkt auf die Roundtrip-Berechnung (siehe 1.2.5.3) auswirkt.



(a) Verlustrate

(b) Verzögerung und Jitter

Abbildung 5.8: StaticPolicer mit Assured Forwarding 1 bei langsam anschwellendem Störverkehr



(a) Verlustrate

(b) Verzögerung und Jitter

Abbildung 5.9: StaticPolicer mit Expedited Forwarding bei langsam anschwellendem Störverkehr

5.5 SwitchingPolicer

SwitchingPolicer ist der erste von zwei Policern, die den Ansatz aus 3.3 verwirklichen. In diesem Test wird ermittelt, wann von der tiefen zur hohen Serviceklasse und wann zurück gewechselt wird und ob diese Wechsel zu den korrekten Zeiten geschehen.

5.5.1 Aufbau

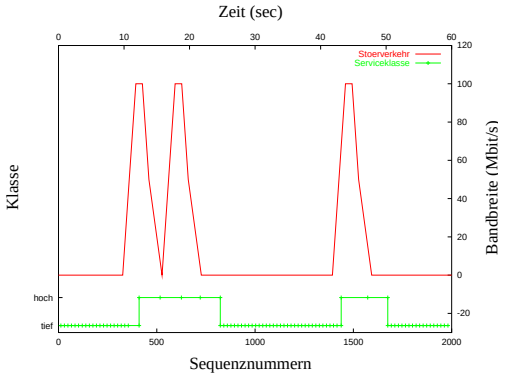
Wiederum war der von Challenger gesendete Störverkehr an allen drei Angriffspunkten identisch. Der UDPgen-Skript bestand aus drei Wiederholungen von `shortpeak.script` (siehe Abb. 5.4), die ersten beiden kurz hintereinander und nach längerer Pause gefolgt von der dritten. Als tiefe Serviceklasse fungierte Best Effort, als hohe Premium Service.

5.5.2 Resulate

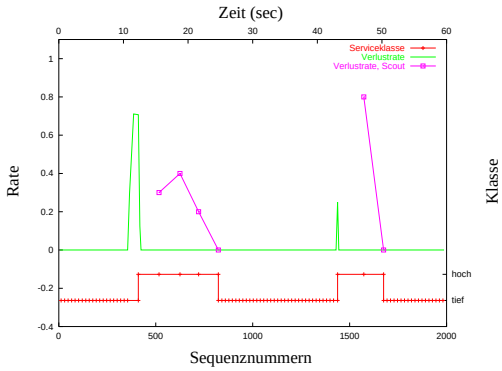
Abb. 5.10(a) zeigt das zentrale Resultat des Tests: SwitchingPolicer wechselte die Serviceklasse korrekt. Die Punkte im Graphen bezeichnen die Zeitpunkte, an denen ein Klassenwechsel erwägt wurde. Dabei fällt auf, dass solche Erwägungen seltener sind, wenn die hohe Serviceklasse benutzt wird. Das kommt daher, dass in der tiefen Klasse die Entscheidung zum Klassenwechsel rein aufgrund der regelmässig eintreffenden RR-Pakete gefällt wird, während in der hohen Serviceklasse jedesmal die Beendigung der Sondenpaket-Messreihe abgewartet werden

muss. Dieser Effekt ist aber durchaus erwünscht, da ein verspäteter Wechsel zur hohen Klasse starke Qualitätseinbussen mit sich bringt, während ein verspätetes Zurückfallen keine negativen Auswirkungen auf die Übertragungsqualität hat. Tatsächlich übersprang SwitchingPolicer die kurzfristige Erholung zwischen den ersten beiden Spitzen.
Wie Abb. 5.10(b) entnommen werden kann, entspricht der Verlauf der Paketverlustrate den Erwartungen. Bei einer Auslastungsspitze stieg sie an, bis SwitchingPolicer zur hohen Klasse wechselte, worauf sie auf 0 zurückfiel. Sobald die Scout-Resultate wieder besser wurden, fiel das Programm zur tiefen Klasse zurück, was aber dank der vorhergehenden Messreihen keine Qualitätseinbussen mit sich brachte.
Auch die in Abb. 5.10(c) dargestellten Werte zeigen ein Verhalten, wie es nach den vorherigen Tests vorauszusehen war: Die Verzögerung stieg zu Beginn der Verkehrsspitzen sprunghaft an und stabilisierte sich auf einem etwas erhöhten Niveau, sobald der Serviceklassenwechsel erfolgt war. Der Jitter verdoppelte sich kurzfristig, um dann auch wieder auf ein stabiles, aber leicht erhöhtes Niveau zu sinken.
Die Scout-Ergebnisse sind da schon interessanter. Vergleich man diejenigen zu Beginn des Experiments mit denen kurz vor dem Ende, fällt der unterschiedliche Jitter-Verlauf auf. Bei einer einzelnen Spitze ist er fast konstant tief; folgen hingegen zwei Spitzen direkt hintereinander, wächst er stark an. Dieser Unterschied entsteht hauptsächlich dadurch, dass das Programm den Jitter der Sondenpakete mit einem Median berechnet, was die Schätzung gegen Ausreisser unempfindlicher macht und der Jitterdefinition in [RFC 2598] entspricht. Dadurch hat die nur kurzfristige unruhige Phase während der dritten Verkehrsspitze praktisch keine Auswirkungen, währenddem sich die länger andauernde Unruhephase während den ersten beiden klar abzeichnet. Messungen mit dem für Ausreisser empfindlicheren arithmetischen Mittelwert zeigten auch bei einem einzelnen Peak erhöhte Jitterwerte.

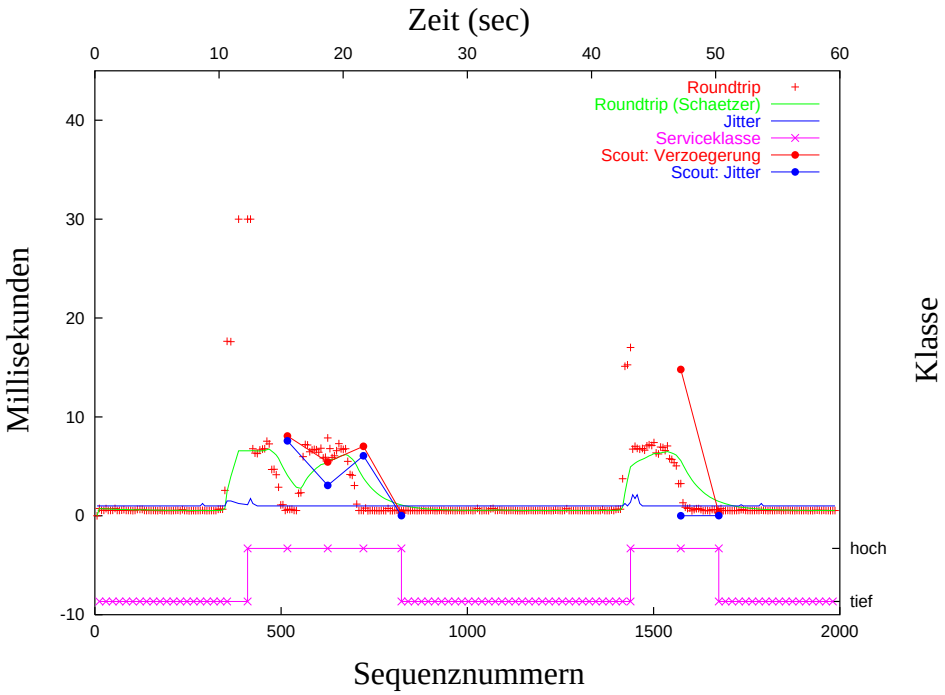
Mit Abb. 5.11 kann der Entscheidungsverlauf von SwitchingPolicer nachvollzogen werden. Es ist deutlich zu erkennen, wie das Programm auf eine schlechter werdende Bewertungen der Situation mit einem Klassenwechsel reagierte, darauf die Resultate der Scout-Messungen beobachtete und zurückfiel, sobald diese auf eine Erholung der Situation schliessen liessen. Offensichtlich hatten nur die Paketverlustraten der RTP-Pakete und der Sondenpakete Einfluss auf die Entscheidungen zum Klassenwechsel. Die Bewertungen der übrigen Parameter waren zu klein, um Einfluss auszuüben.



(a) Wahl der Serviceklasse



(b) Verlustrate



(c) Verzögerung und Jitter

Abbildung 5.10: Resultate des SwitchingPolicer-Tests

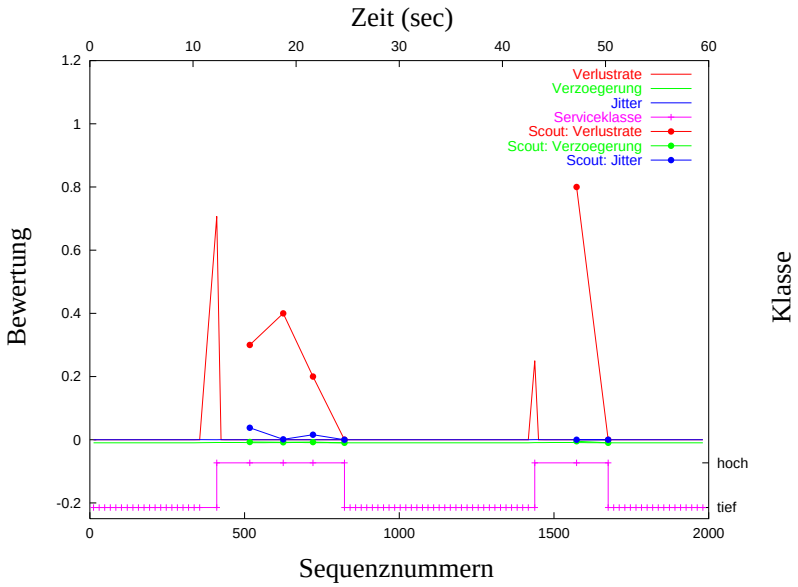


Abbildung 5.11: Bewertungen im SwitchingPolicer-Test

5.6 ServicePolicer

Der Test von ServicePolicer verfolgt ähnliche Ziele wie derjenige von Switching-Policer. Die hauptsächliche Fragestellung ist, wann Klassenwechsel erfolgen und ob diese Wechsel dann geschehen, wenn es sinnvoll ist.

5.6.1 Aufbau

Die vergleichsweise geringe Zahl von Knoten im Testnetz und die Gleichartigkeit ihrer Verbindungen verunmöglicht einen vollständigen, alle Serviceklassen umfassenden Test. Die wirkungsvolle Störung einer Serviceklasse belegt jeweils mindestens eine von Challenger ausgehende Leitung. Aus diesem Grund wurde der Test von ServicePolicer mit folgendem Aufbau durchgeführt:
Die Routen Challenger-Endeavour-Enterprise und Challenger-Atlantis-Endeavour-Enterprise wurden, mit 3 Sekunden Verzögerung, 45 Sekunden lang mit Best-Effort-UDP-Verkehr geflutet. Zusätzlich wurde 30 Sekunden nach Beginn des Tests eine Assured-Service-1-Spitze über Challenger-Discovery-Atlantis-Endeavour-Enterprise gesendet. Dadurch sollte der Policer zum Wechsel von Best Effort zu Assured Service 1, und kurzfristig Assured Service 2, gezwungen werden.
UDPgen selbst ist nicht DiffServ-fähig. Deshalb musste Discovery die Pakete

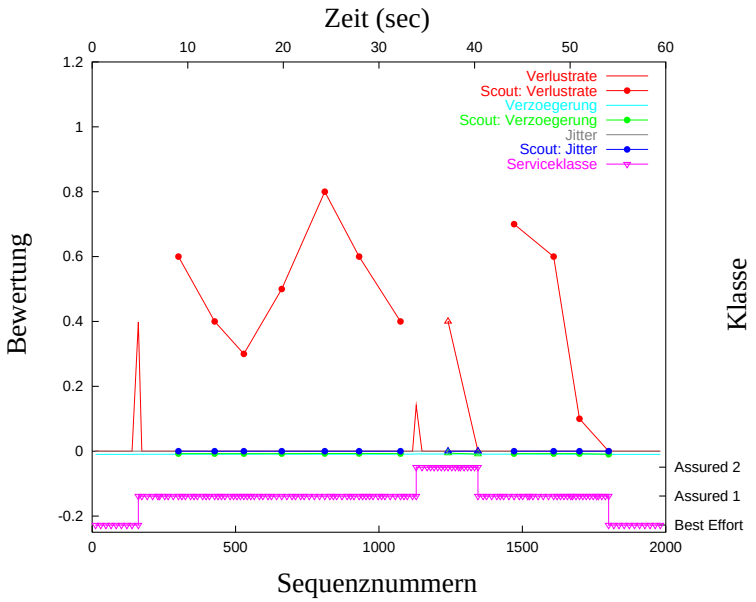


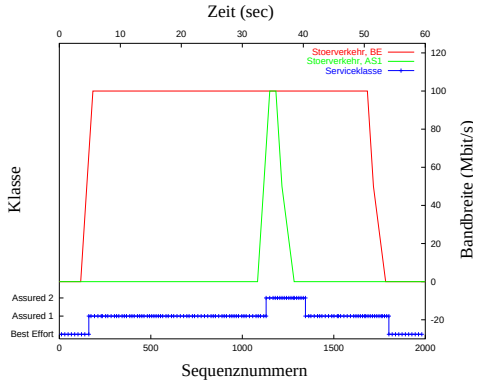
Abbildung 5.12: Bewertungen im ServicePolicer-Test

des erwähnten Assured-Service-1-Störverkehrs mit dem entsprechenden DS-Codepunkt markieren. Die dazu nötige Flussbeschreibungstabelle ist im Anhang A abgedruckt.

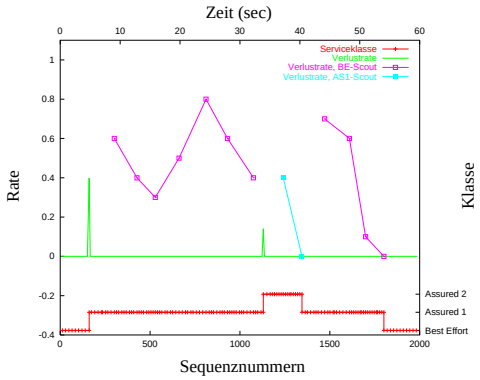
5.6.2 Resulate

Die Klassenwechsel erfolgten korrekt und zu den gewünschten Zeitpunkten, wie Abb. 5.13(a) zeigt. Wie Abb. 5.13(b) und Abb. 5.13(c) entnommen werden kann, waren auch die Verläufe der Verlustrate und der Roundtrip-Zeit, sowie die Messresultate der Scout-Objekte zufriedenstellend. Da in diesem Test der Bandbreitenverlauf meist flach verlief, waren die Jitter-Messungen des Scouts tief, sogar tiefer als die RTP-Messungen auf höheren Serviceklassen. Offenbar liefern die Sondenpakete, durch ihre geringe Anzahl oder die unterschiedliche Berechnungsmethode bedingt, tendenziell tiefere Werte als der in RTP verwendete Schätzer. Die schon in Abschnitt 5.4 festgestellte erhöhte Verzögerung beim Fluten einer anderen Serviceklasse wirkte sich hier doppelt aus. Es bliebe zu überprüfen, ob sich diese zweite Erhöhung auch auf Expedited Forwarding auswirkt.

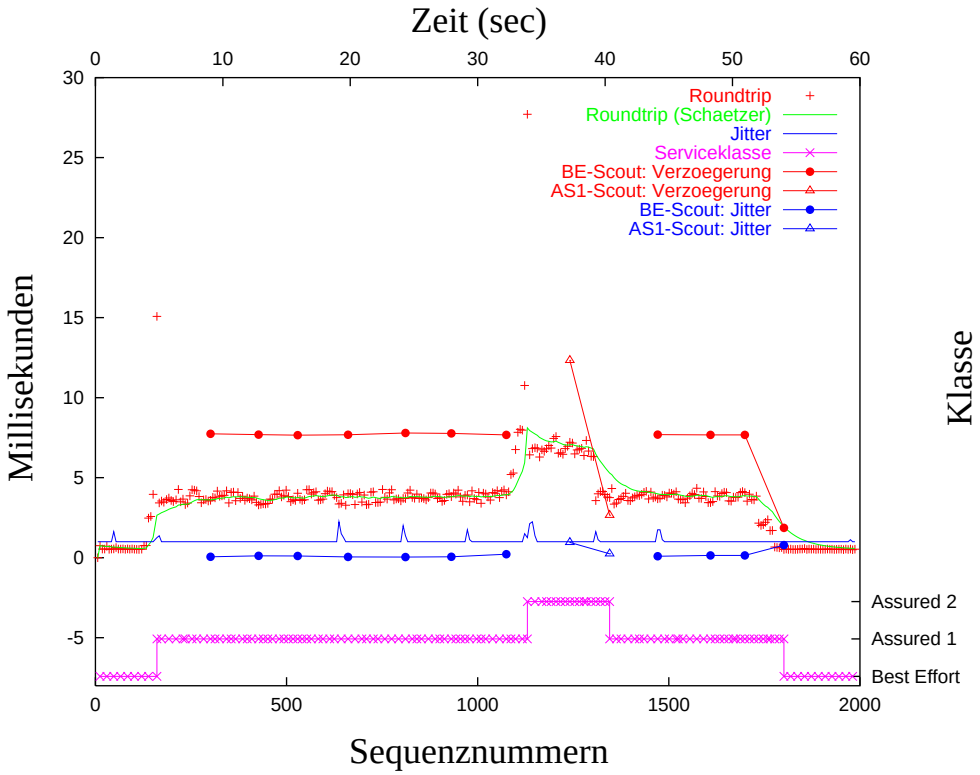
Abb. 5.12 zeigt den Verlauf der Situationsbewertungen. Wiederum war die Verlustrate der dominierende Faktor. Die Bewertungen aller anderen Faktoren war nahezu Null, die der Verzögerung war sogar negativ.
Die Grafik zeigt ausserdem sehr gut, wie der ServicePolicer auf die Bewertungen



(a) Wahl der Serviceklasse



(b) Verlustrate



(c) Verzögerung und Jitter

Abbildung 5.13: Resultate des ServicePolicer-Tests

der Verlustraten der RTP-Pakete und der Sondenpakete reagierte. Sobald erstere stieg, wechselte er zur nächsthöheren Klasse und fiel zurück, wenn die vom Scout ermittelten Werte wieder sanken.

Kapitel 6

Zusammenfassung und Ausblick

Viele der in Kapitel 2 diskutierten Probleme der IP-Telefonie können mit Differentiated Services behoben oder zumindest stark vermindert werden. Mit dem einfachen Ansatz, wie er in Abschnitt 3.2 beschrieben ist, können Resultate erzielt werden, die mit denen von Integrated Services, speziell RSVP, vergleichbar sind. Gleichzeitig löst er einige der Probleme, die der Einsatz von RSVP mit sich bringt, namentlich das Skalierungsproblem, die durch Endpunkt-orientierte Reservierung bedingte schlechte zentrale Verwaltbarkeit und die fehlende Unterstützung von Systemen ohne spezielle QoS-Fähigkeiten.
Der Ansatz des Adaptierens der Serviceklasse eines Flusses (Abschnitt 3.3) ermöglicht DiffServ-Anwendungen, sich dynamisch an die Gegebenheiten in einem Netzwerk anzupassen und so ein gegebenes Service Level Agreement optimal auszunutzen. Dadurch können die Kunden eines Providers die Möglichkeiten, die ihnen ein Vertrag bietet, optimal ausnutzen und somit Kosten sparen.
Im Abschnitt 3.4 wurde das Markieren prioritärer Pakete zusammen mit zwei verwandten Kodierungsschemata vorgestellt. Während der reale Einsatz dieser beiden Schemata wegen ihrer fehlenden Bandbreiteneffizienz nur in Spezialfällen vorteilhaft erscheint, könnte der allgemeine Ansatz in nicht verzögerungsempfindlichen Anwendungen Verwendung finden. Als Beispiel wären Streaming-Anwendungen wie Internet-Radio zu nennen.
Die während der Arbeit entstandene Software namens DSPhone implementiert die ersten zwei der in Kapitel 3 vorgeschlagenen Ansätze vollständig und den dritten teilweise. Einige Fehler des Programms konnten nicht bereinigt werden, der für die Tests notwendige Teil läuft jedoch stabil.
Um DSPhone und die vorgeschlagenen Ansätze zu überprüfen, wurde das Verhalten von DSPhone unter verschiedenen Umständen in einem DiffServ-Testnetzwerk gemessen. Dabei wurden beide implementierten Ansätze analysiert und mit dem Verhalten ohne DiffServ-Einsatz verglichen, was zeigte, dass beide Ansätze realistisch einsetzbar sind.

Viele Themen wurden in dieser Arbeit nur kurz angeschnitten. Bei einigen davon würde es sich lohnen, sie zu vertiefen. Eines davon ist die in Abschnitt 3.2 nur sehr allgemein behandelte verteilte Bandbreitenreservierung, meist Bandwidth Brokering genannt. Es wäre durchaus interessant, sich zu überlegen, wie man ein solches System in einem VoIP-Netz und den dazugehörigen Signalisierungsprotokollen integrieren könnte.
Der Ansatz des Adaptierens der Serviceklasse kann durch das Optimieren der Bewertungsmechanismen und Sendeparameter der Sondenpakete, oder durch die Intergration herkömmlicher Methoden (Kodierungsadaption etc.) weiter verfeinert werden. Wenn die Annahme der festen Qualitätsrelationen zwischen den Serviceklassen aufgegeben wird, ergibt sich eine Reihe weiterer Fragen: Welche Klassen sollen beobachtet werden und mit welcher Bandbreite? Wie hält man die Reaktionszeit des Algorithmus möglichst klein? Bestehen eventuell Korrelationen zwischen Serviceklassen, die man ausnutzen könnte, um die Sondierungsbandbreite zu vermindern?

DSPhone ist dafür ausgelegt, erweiterbar zu sein, das Programm kann daher weiterhin als Experimentiergrundlage dienen. Allerdings wäre es in einem solchen Fall angebracht, einige der in Abschnitt 4.3 vorgeschlagenen Designänderungen durchzuführen. Mit weniger Aufwand verbunden wäre der Einsatz als realistisches Element in einem Testnetzwerk, sei es als Messwerkzeug oder als Verkehrsgenerator.
UDPgen ist ein flexibles und nützliches Hilfsmittel, eigentlich ein Nebenprodukt der Arbeit, und eignet sich als Generator von Störverkehr in vielen denkbaren Testnetzwerken. Zusammen mit UDPrcv, das die ankommende Bandbreite aufzeichnet, ist es auch für die Simulation eines Nutzdatenstroms verwendbar.

Anhang A

Konfigurationsdateien und Skripte

A.1 DiffServ-Konfiguration der Router

In allen Tests ausser demjenigen von ServicePolicer wurden die Router mit folgendem Shell-Skript konfiguriert.

```
#!/bin/sh

tc=‘which tc‘
devnum=$1
dev="eth${devnum}"
table=$devnum

if [ "${devnum}" == "" ]; then
echo Specify network device number
exit 2
fi

# base system
$tc qdisc add dev $dev root handle 1: dsclsfr

# Assured Service part
$tc qdisc add dev $dev parent 1:1 handle 11: trio limit 100\
    low_begin 0.9 low_end 1.0 medium_begin 0.5 medium_end 0.9\
    high_begin 0.1 high_end 0.5
$tc qdisc add dev $dev parent 1:2 handle 12: trio limit 100\
    low_begin 0.9 low_end 1.0 medium_begin 0.5 medium_end 0.9\
    high_begin 0.1 high_end 0.5
$tc qdisc add dev $dev parent 1:3 handle 13: trio limit 100\
    low_begin 0.9 low_end 1.0 medium_begin 0.5 medium_end 0.9\
    high_begin 0.1 high_end 0.5
$tc qdisc add dev $dev parent 1:4 handle 14: trio limit 100\
    low_begin 0.9 low_end 1.0 medium_begin 0.5 medium_end 0.9\
    high_begin 0.1 high_end 0.5
```

Für den ServicePolicer-Test wurde Discovery mit folgendem Skript als Ingress-Knoten konfiguriert, während die anderen Router innere Knoten blieben.

```
#!/bin/sh

tc=‘which tc‘
devnum=$1
dev="eth${devnum}"
table=$devnum

if [ "${devnum}" == "" ]; then
    echo Specify network device number
    exit 2
fi

# base system
$tc qdisc add dev $dev root handle 1: serv_handler table_id $table
$tc qdisc add dev $dev parent 1:1 handle 2: dsclsfr

# Assured Service part
$tc qdisc add dev $dev parent 2:1 handle 21: prec_handler\
    table_id $table dscp as1
$tc qdisc add dev $dev parent 2:2 handle 22: prec_handler\
    table_id $table dscp as2
$tc qdisc add dev $dev parent 2:3 handle 23: prec_handler\
    table_id $table dscp as3
$tc qdisc add dev $dev parent 2:4 handle 24: prec_handler\
    table_id $table dscp as4
$tc qdisc add dev $dev parent 21:1 handle 211: trio limit 100\
    low_begin 0.9 low_end 1.0 medium_begin 0.5 medium_end 0.9\
    high_begin 0.1 high_end 0.5
$tc qdisc add dev $dev parent 22:1 handle 221: trio limit 100\
    low_begin 0.9 low_end 1.0 medium_begin 0.5 medium_end 0.9\
    high_begin 0.1 high_end 0.5
$tc qdisc add dev $dev parent 23:1 handle 231: trio limit 100\
    low_begin 0.9 low_end 1.0 medium_begin 0.5 medium_end 0.9\
    high_begin 0.1 high_end 0.5
$tc qdisc add dev $dev parent 24:1 handle 241: trio limit 100\
    low_begin 0.9 low_end 1.0 medium_begin 0.5 medium_end 0.9\
    high_begin 0.1 high_end 0.5

# Premium Service part
$tc qdisc add dev $dev parent 2:5 handle 25: premium_shaper\
    table_id $table classes 1
```

Die dazugehörige Flussbeschreibungstabelle war diese:

```
# DS table for discovery eth1
#
# challenger to enterprise’s subnet, 100 Mbit/s as1 ingress
4   10.1.11.1 255.255.255.255 *  10.1.4.0  255.255.255.0 * UDP * as1\
  0        0
4   10.1.11.1 255.255.255.255 *  10.1.4.0  255.255.255.0 * UDP as1 *\
  12500000 125000
# columbia’s subnet to anywhere, 1 Mbit/s for any service
4   10.1.1.0  255.255.255.0   *  0.0.0.0   0.0.0.0       * UDP as1 *\
  1000000  20000
4   10.1.1.0  255.255.255.0   *  0.0.0.0   0.0.0.0       * UDP as2 *\
  1000000  20000
4   10.1.1.0  255.255.255.0   *  0.0.0.0   0.0.0.0       * UDP as3 *\
  1000000  20000
4   10.1.1.0  255.255.255.0   *  0.0.0.0   0.0.0.0       * UDP as4 *\
1000000  20000
4   10.1.1.0  255.255.255.0   *  0.0.0.0   0.0.0.0       * UDP ps  *\
  1000000  20000
```

A.2 DSPhone-Konfigurationsdateien

Die Konfigurationdatei des Empfängers war in allen Tests die folgende:

```
# DSPhone Receiver ResourCe script

debug_context = 0x0401

audio_dev = "file"
audio_outfile = "/var/tmp/result.raw"
audio_bits = 16
audio_rate = 8000

sdes_name = "Rasputin Receiver"
sdes_email = "receiver@sink"
sdes_phone = ""
sdes_loc = "Sink"
sdes_tool = "DSPhone"
sdes_note = ""

debug_use_redirection = false
```

Es folgen die Konfigurationsskripte der Senderseite, in der Reihenfolge der Tests. Wo möglich wurden identische Teile ausgelassen (markiert durch `[...]`)

DefaultPolicer

```
# DSPhone Sender ResourCe script

debug_context = 0x2401
debug_use_redirection = false

proto_mode = "send"

# Audio block
audio_dev = "file"
audio_infile = "/var/tmp/oops.raw"
audio_bits = 16
audio_rate = 8000
packetization_interval=30

# Policer block
policer = "default"

# SDES block
sdes_name = "Sergej Sender"
sdes_email = "sender@source"
sdes_phone = ""
sdes_loc = "Source"
sdes_tool = "DSPhone"
sdes_note = ""
```

StaticPolicer - Assured Service 1

```
[...]

# Policer block
policer = "static"
static_dstype = "assured 1"

[...]
```

StaticPolicer - Premium Service

```
[...]

# Policer block
policer = "static"
static_dstype = "premium"

[...]
```

SwitchingPolicer

```
[...]

# Policer block
policer = "switching"
low_type = "none"
high_type = "premium"
switching_treshold = 0.2

[...]
```

ServicePolicer

```
[...]

# Policer block
policer = "service"
evaluation_interval = 3
packet_loss_treshold = 0.1
jitter_treshold = 0.5
delay_treshold = 0.4

[...]
```

Literaturverzeichnis

[RFC 1889] RTP: A Transport Protocol for Real-Time Applications,
Schulzrinne, Casner, Frederick, Jacobson,
IETF AVT Working Group 1996

[RFC 1890] RTP Profile for Audio and Video Conference with Minimal Control,
H. Schulzrinne,
IETF AVT Working Group 1996

[RFC 2326] Real Time Streaming Protocol (RTSP),
Schulzrinne, Rao, Lanphier,
IETF 1998

[RFC 2508] Compressing IP/UDP/RTP Headers for Low-Speed Serial Links,
Casner, Jacobson,
IETF 1999

[G.711] Pulse code modulation (PCM) of voice frequencies,
© ITU 1988

[G.726] Recommendation G.726 (ADPCM),
© ITU 1990

[1016] Analog to Digital Conversion of Radio Voice by 4,800 bit/second Code Excited Linear Prediction (CELP),
Federal Standard 1016, US Department of Defense

[Proc94] Fast CELP Algorithm and Implementation for Speech Compression,
Langi, Grieder, Kinsner,
Proc. 1994 Digital Communications Conference

[H.323] Packet-based multimedia communication systems,
© ITU 1998

[H.245] Control protocol for multimedia communication,
© ITU 1998

[H.255.0] Call Signalling protocols and media stream packetization for packet-based multimedia communication systems,
© ITU 1998

[RFC 2543] SIP: Session Initiation Protocol,
Handley, Schulzrinne, Schooler, Rosenberg,
IETF 1999

[RFC 2327] SDP: Session Description Protocol,
Handley, Jacobson,
IETF 1998

[P.861] Objective quality measurement of telephone-band (30-3400 Hz) speech codecs,
© ITU 1998

[RFC 2205] Resource ReSerVation Protocol (RSVP),
Braden, Zhang, Berson, Herzog, Jamin,
IETF 1997

[RFC 2210] The Use of RSVP with IETF Integrated Services,
Wroclawski,
IETF 1997

[RSVPmap] Integrated Service Mappings for Differentiated Services Networks,
Wroclawski, Charny,
IETF 2001

[RFC 2475] An Architecture for Differentiated Services,
Blake, Black, Carlson, Davies, Wang, Weiss,
IETF 1998

[RFC 2474] Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers,
Nichols, Blake, Baker, Black,
IETF 1998

[RFC 2597] Assured Forwarding PHB Group,
Heinanen, Baker, Weiss, Wroclawski,
IETF 1999

[RFC 2598] An Expedited Forwarding PHB,
Jacobson, Nichols, Poduri,
IETF 1999

[RED] Random Early Detection gateways for Congestion Avoidance,
Floyd, Jacobson,
IEEE/ACM Transactions on Networking, Volume 1, Number 4, August 1993, pp. 397-413

[LinuxDS] A Linux Implementation of a Differentiated Services Router,
Braun, Scheidegger, Einsiedler, Stattenberger, Jonas, Stüttgen,
Networks and Services for Information Society (Interworking'2000)

[Stattb] Performance Evaluation of a Linux DiffServ Implementation,
Stattenberger, Braun, Brunner, Stüttgen,
2001, liegt Elsevier Science vor

[RepDS] Implementation and Configuration of a Linux Differentiated Services Router,
Stattenberger, Braun,
Technical Report, IAM-00-010, November 2000

[AtmDS] Implementation of Differentiated Services over ATM,
Braun, Dasen, Scheidegger, Jonas, Stüttgen,
Conference on High Performance Switching & Routing (Joint IEEE ATM Workshop 2000 and 3rd International Conference on ATM), 2000 Heidelberg, Germany

[DousB] IP Telephony,
Bill Douskalis,
Hewlett-Packard 2000

[Alsa] Advanced Linux Sound Architecture,
www.alsa-project.org

[Esd] Enlightened Sound Daemon,
www.tux.org/˜ricdude/EsounD.html

[Oss] Open Sound System (OSS/Free),
im Linux-Kernel enthalten

[Gtkmm] Gtk--,
gtkmm.sourceforge.net

[Gtk] GTK+: The Gimp ToolKit,
www.gtk.org