Bewertung von adaptiven Ausspielalgorithmen für paketvermittelte Audiodaten

Studienarbeit

von Christian Sieckmeyer
Matrikel-Nr. 113034

19. Oktober 1995

Prof. Adam Wolisz
Betreuer: Dr. Henning Schulzrinne (GMD Fokus, Berlin)

Technische Universität Berlin
Fachbereich 12 - Elektrotechnik
Institut für Fernmeldetechnik

Inhaltsverzeichnis

Einführung

Die Leistungsfähigkeit moderner Workstations und die fortschreitende Vernetzung von Rechenanlagen ermöglichen interaktive Echtzeitdienste. Vor allem die Entwicklung in der Netzwerktechnik erlaubt es, mehr Dienste als Dateitransfer, elektronische Post oder virtuelle Terminals anzubieten. Durch die höheren Übertragungsraten, die heutige und vor allem zukünftige Netzwerke bieten, entstehen neue Anwendungsgebiete. Besonderes Interesse besteht an multimedialen Kommunikationsmöglichkeiten.

In dieser Arbeit beschäftige ich mich mit dem Thema der interaktiven Audiokommunikation. Es soll untersucht werden, welche Möglichkeiten derzeit genutzte Netzwerke, wie das *Internet*, und Endsysteme bieten, um Sprache übertragen zu können. Es werden die hierfür erforderlichen Protokolle und Verarbeitungstechniken erläutert.

Schwerpunkt der Arbeit ist die Simulation eines Empfängers von Audiodaten, um die Fähigkeiten von adaptiven Algorithmen zu testen, die die Eigenschaften eines paketvermittelnden Systems kompensieren sollen. Der Empfang realer Übertragungen wird aufgezeichnet, um die Eigenschaften des Netzwerkes innerhalb einer Simulation reproduzieren zu können. Dies erlaubt eine genaue Betrachtung und Beurteilung von Techniken des Empfangs von interaktiven Audiodaten.

Das erste Kapitel beschreibt die Eigenschaften von interaktiven Kommunikationsbeziehungen und die daraus resultierenden Anforderungen an ein Netzwerk. Im zweiten Kapitel werden paketvermittelnde Systeme vorgestellt und deren Möglichkeiten für die interaktive Kommunikation bewertet. Es werden für diese Kommunikationsform hinderliche Eigenschaften des Netzwerkes erläutert. Anschließend werden die Neuerungen im Bereich der Transportprotokolle für Echtzeitanwendungen zusammengefaßt. Am Ende des Kapitels erläutere ich die Möglichkeit der Kompensation der Netzwerkeigenschaften durch die verzögerte Ausspielung des Audiosignals. Das nächste Kapitel stellt adaptive Algorithmen vor, die der Bestimmung einer geeigneten Ausspielverzögerung dienen. Anschließend wird die Simulationsumgebung beschrieben, mit der die Leistungsfähigkeiten der Algorithmen untersucht werden. Im letzten Kapitel werden die Ergebnisse der Simulationen dargestellt und erläutert.

Kapitel 1

Interaktive Audioübertragung

Unter einer interaktiven Kommunikation versteht man den aktiven und reaktiven Austausch von Information. Auf eine Sendung reagiert der Empfänger seinerseits mit einer Sendung. Verschiedene Kommunikationsbeziehungen, die unter diese Kategorie fallen, sind denkbar:

- Telefonie (Konversation zwischen zwei Teilnehmern)
- Konferenz (Konversation zwischen mehreren Teilnehmern)
- Informationsdienst (z.B. Radio)

Für die interaktive Kommunikation ist es wichtig, daß die Daten möglichst schnell vom Sender zum Empfänger gelangen. Ein Gespräch zwischen zwei Menschen zeichnet sich u.a. dadurch aus, daß ein Informationsaustausch stattfindet, der aus Rede und Gegenrede besteht. Ein Sprecher erwartet von der Gegenseite eine Reaktion. Zu lange Zeitabstände zwischen Versand und Empfang des Gesprochenen würden die Teilnehmer verwirren oder ein Gespräch sogar unmöglich machen. Eine Situation, in der dies sehr deutlich wird, ist, wenn der Zuhörende den Redner unterbrechen möchte. Mitten im Redefluß wirft der Zuhörende einen Kommentar ein. Aufgrund der Verzögerungen in beiden Richtungen kommt dieser Kommentar einen Moment später beim Redner an, unter Umständen zu einem völlig unpassenden Zeitpunkt.

In einer weiteren Situation machen sich lange Verzögerungen ebenfalls negativ bemerkbar. Ein Gesprächsteilnehmer stellt eine Frage und muß anschließend sehr lange auf die Antwort warten. Der Fragende nimmt nun an, daß er nicht verstanden wurde und wiederholt sich unnötig.

Menschen reagieren also sehr sensibel auf Verzögerungen des akustischen Signals ([Par93], Kap. 8.5). Mit der Frage, welche Verzögerungen für die sprachliche Kommunikation akzeptabel sind, beschäftigte man sich schon in den sechziger Jahren. Auch die Einführung von transatlantischen Telefonverbindungen, in denen lange Signallaufzeiten durch die großen Distanzen, wie bei Satellitenverbindungen, auftreten, war Auslöser für Forschungen auf diesem Gebiet.

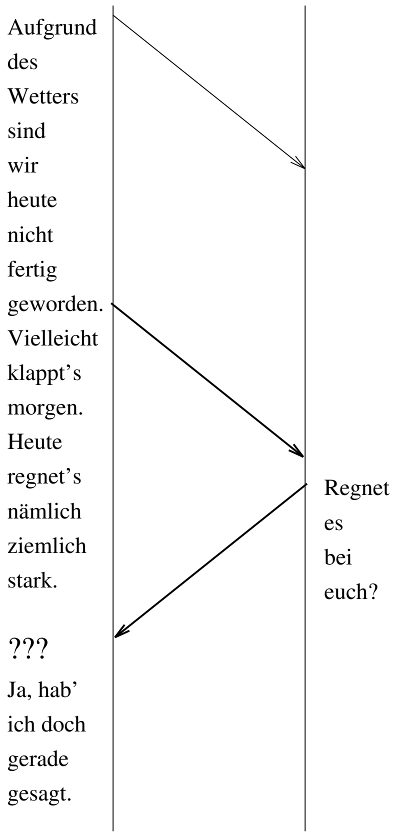

Abbildung 1.1: Zeitablaufdiagramm eines Dialoges

Die Empfehlung G.144 der ITU liefert Anhaltspunkte über die Auswirkungen von Verzögerungen auf die zwischenmenschliche Kommunikation. Ein Auszug ist in der Tabelle 1.1 dargestellt.

Bei der Übertragung mittels eines Computernetzwerkes spielt nicht nur die Entfernung eine Rolle. Das nächste Kapitel wird zeigen, daß weitere Ursachen für Verzögerungen verantwortlich sind.

Wenn für den Datenaustausch maximale Grenzen der Verzögerung vorgeschrieben sind, handelt es sich um eine *Echtzeit*-Kommunikation [Sch95b]. Die Anforderung an das Netzwerk für die interaktive Nutzung lautet z.B., daß das akustische Signal nach seiner Versendung spätestens in 200 ms beim Empfänger hörbar werden muß. Die Zeitdifferenz sollte kaum oder gar nicht wahrnehmbar sein.

Bei der Übertragung von Sprache muß kontinuierlich Information vom Sender zum Empfänger gelangen. Im Abschnitt 2.2 wird deutlich, daß der Sender mit einer bestimmten Frequenz Informationseinheiten erzeugt. Mit dieser Frequenz müssen die Pakete auch beim Empfänger für die Weiterverarbeitung zur Verfügung stehen. Man spricht bei dieser Form von Daten von *Continuous Media*.

Einweg-Verzögerung	Qualität
0 bis 150 ms	akzeptabel
150 bis 400 ms	störend
über 400 ms	nicht akzeptabel

Tabelle 1.1: ITU G.114-Empfehlung zu Einweg-Verzögerungen

Zusammenfassend soll festgehalten werden, daß die interaktive Audio-Übertragung zwei spezielle Dienste voraussetzt:

- Echtzeit-Dienst
- Continuous Media-Dienst

Bei Verwendung von Computernetzwerken, die diese Dienste nur unzureichend erbringen, fallen auch dem Empfänger Aufgaben zu, die eine erfolgreiche Übertragung ermöglichen. Dies ist Thema des nächsten Kapitels, das sich mit paketvermittelnden Netzwerken auf der Basis der TCP/IP-Protokollfamilie beschäftigt.

Kapitel 2

Paketvermittlung

2.1 Eigenschaften der Paketvermittlung

Die Audiodaten sollen durch ein paketvermittelndes Netzwerk übertragen werden. Die Paketvermittlung bietet mehr Flexibilität als die Leitungsvermittlung, was für den Aufbau verschiedenster Kommunikationsbeziehungen entscheidend ist.

Man stelle sich z.B. einen *Radiosender* vor, der seine Informationen über ein Netzwerk verbreitet. Üblicherweise sollte die Anzahl der Zuhörer dabei nicht beschränkt sein. Würde so ein Sender auf der Basis der Leitungsvermittlung arbeiten, müßte jeder Zuhörer eine explizite Verbindung zu der Radiostation aufbauen. Die beim Sender zur Verfügung stehenden Leitungen wären sicher schnell erschöpft. Auch *Konferenzsysteme* lassen sich einfacher auf paketvermittelnden Netzwerken realisieren. Technisch betrachtet, ähnelt eine Konferenzbeziehung der Beziehung bei einer Radiosendung; der Unterschied ist, daß die Rolle des Senders bei einer Konferenz ständig wechselt. Ein Gespräch zwischen zwei Personen ist durch Paketvermittlung kostengünstiger zu realisieren, da eine Leitung für mehrere Kommunikationsbeziehungen verwendet werden kann. Während eines Gespräches kommt es zu Pausen, in denen keine Daten übertragen werden müssen. Innerhalb dieser Pausen kann die Leitung für eine andere Kommunikationsbeziehung genutzt werden.

Es gibt zwei Varianten der Paketvermittlung innerhalb der TCP/IP-Protokollfamilie [Com91]. Die ungesicherte Datagrammübertragung, wie sie UDP bietet, ist die geeignete Form für die Übertragung von Echtzeitdaten. Protokolle, die eine gesicherte Übertragung gestatten (TCP), hätten zu lange Verzögerungen zur Folge. Fehlererkennungs- und Behebungsmechanismen, die über Bestätigungen liefen, würden den Ablauf der Kommunikation zu sehr bremsen. Der Nachteil der ungesicherten Übertragung ist vertretbar, da geringe Paketverluste zwar die Qualität des empfangenen Signals verschlechtern, aber eine akustisch verständliche Ausspielung beim Empfänger möglich bleibt.

Die Paketvermittlung birgt aber auch Schwächen, die sich gerade auf die Übertragung von Echtzeitdaten nachteilig auswirken. Ein paketvermittelndes Netzwerk überträgt einen Datenstrom blockweise. Jeder Block wird als ein Paket vom Sender zum Empfänger geschickt.

Liegen zwischen Sender und Empfänger mehrere Netzknoten, ist der Weg, den die Pakete passieren, nicht vorhersagbar. Im Laufe einer Übertragung, die aus mehreren Paketen besteht, kann sich der Weg auch ändern. Die Wahl des Weges geschieht durch Wegewahlalgorithmen innerhalb des Netzwerkes. Innerhalb der Netzknoten unterliegen die Pakete, aufgrund von Warteschlangen, stochastischen Prozessen.

Der Weg des Paketes und die Zeit, die es braucht, um durch das Netzwerk zu gelangen, sind unbestimmt. Das hat zur Folge, daß Pakete in falscher Reihenfolge beim Empfänger eintreffen können und die Netzwerkverzögerung nicht konstant ist. Zudem können überlastete Netzwerke Paketverluste hervorrufen. Auch kann es passieren, daß duplizierte Pakete beim Empfänger eintreffen. Durch Fehlfunktionen innerhalb des Netzwerkes entstehen aus einem Paket, das der Sender verschickt hat, zwei Pakete, die der Empfänger empfängt. Dieses Verhalten war beim Empfang auf einem Rechner in der GMD-Fokus (Berlin, Deutschland) von *multicast*-Übertragungen [Dee89] des *World Radio Network*[1] zu beobachten, wobei die sendende Station in den USA lag; fast jedes Paket kam doppelt beim Empfänger an.

Ein Empfänger von verbindungslos paketvermittelten Daten muß also mit folgenden Situationen rechnen:

- variable Netzwerkverzögerung
- Duplizieren von Paketen
- Paketverlust
- falsche Paketfolge

Es kommt auf die Form der Daten und auf die Art der Anwendung an, wie und ob überhaupt ein Empfänger auf die Eigenschaften zu reagieren hat. Die nächsten Abschnitte verdeutlichen, daß für die interaktive Audioübertragung jeder der oben genannten Punkte zu berücksichtigen ist.

[1]URL: `http://town.hall.org/radio/live.html`

2.2 Audiocodierung

Um die analogen Signale übertragen zu können, müssen diese in eine digitale Form überführt werden. Es gibt verschiedene Codierungsverfahren, die hierbei zur Anwendung kommen können. Als Beispiel sei hier die Pulse-Code-Modulation (PCM) [Nol92] genannt. Sie vollzieht sich in drei Schritten:

1. Abtastung
2. Analog/Digital-Umsetzung
3. Codierung

Eine PCM-Variante ist die Codierung nach der μ- oder A-Quantisierungkennlinie [ITU92]. Das analoge Signal wird mit der Frequenz $f_s = 8\,\mathrm{kHz}$ abgetastet. Ein Analog/Digital-Umsetzer bildet aus den Abtastwerten einen 12-Bit Quantisierungswert. Anschließend wird mit Hilfe der Quantisierungskennlinie die digitale Information in einem Oktett codiert.

Für eine Stereo-Übertragung werden zwei analoge Signale voneinander getrennt abgetastet und codiert. Codierungsverfahren sehen hierfür die Trennung in Kanäle vor. Die Quantisierungswerte der verschiedenen Kanäle werden in einem Binärmuster codiert.

Codierungsverfahren lassen sich demnach durch folgende drei Größen charakterisieren:

f_s: Abtastfrequenz

l: Anzahl der Bits pro digitalem Abtastwert eines Kanals

c: Anzahl der Kanäle

Bei einer binären Übertragung eines akustischen Signals entsteht damit die Bitrate R:

$$R = f_s\, l\, c$$

2.3 Paketgröße

Der Sender verschickt Pakete in einem festen Takt, der durch die Abtastrate f_s der gewählten Audiocodierung bestimmt ist. Sicher macht es keinen Sinn, jeden Abtastwert einzeln zu senden. Das Verhältnis zwischen der Paketkopfgröße und den Nutzdaten wäre dann sehr ungünstig; hohe Bitraten, wie in Abb. 2.1 dargestellt, wären die Folge. Aus diesem Grund werden n Abtastwerte gesammelt und in einem Paket übertragen. In Abhängigkeit von der Abtastfrequenz f_s, der Anzahl n der Abtastwerte pro Paket und der Paketkopfgröße h in Oktetts ergibt sich die Bitrate R_p:

$$R_p = \frac{(8\,h + n\,l\,c)\,f_s}{n}$$

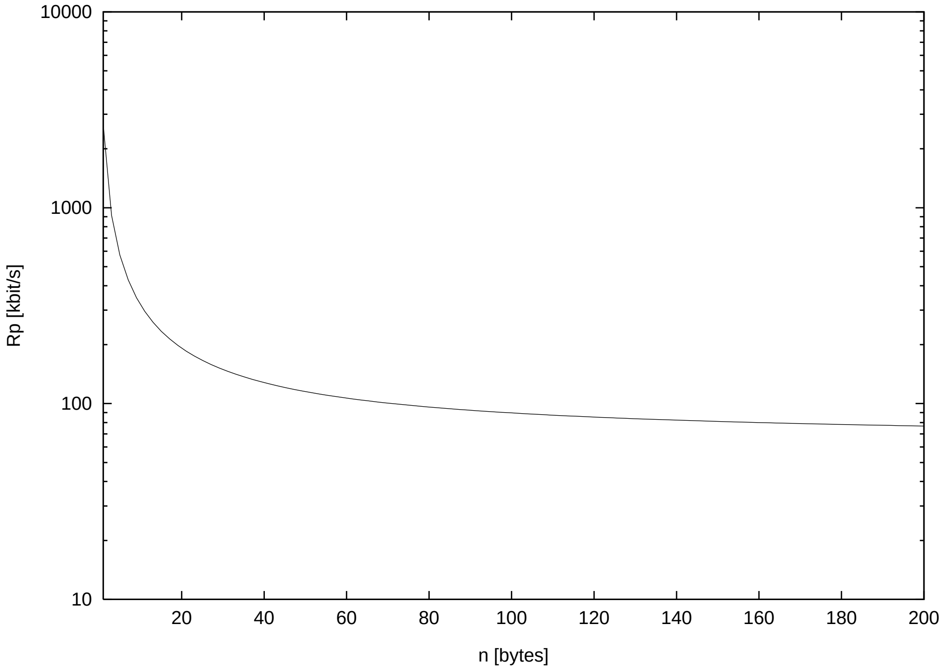

Abbildung 2.1: Byterate R_p in Abhängigkeit von Zahl der Abtastwerte n pro Paket

In Abb. 2.1 ist die Abhängigkeit der Bitrate R_p von n dargestellt. Die Darstellung bezieht sich auf eine RTP-Übertragung (Kap. 2.6.1) über UDP [Com91] von G.711 codierten Signalen [2].

Die Anzahl der Abtastwerte pro Paket darf allerdings nicht zu groß gewählt werden. Die Zeitdauer, um ein Paket zu generieren, ist mindestens so lang wie die Zeit, die für die Aufnahme des akustischen Signals, das das Paket transportieren soll, benötigt wird. Daraus resultiert eine Verzögerung G, die Generationszeit eines Paketes, die zu der Anzahl n proportional ist:

$$G = \frac{1}{f_s} n$$

In einigen Konferenzapplikationen, wie NeVoT[3] oder VAT [4], werden typischerweise alle 20 Millisekunden 160 Abtastwerte mit einem Paket übertragen. In beiden Applikationen kann die Paketgröße eingestellt werden, so daß auch andere Werte für n und f_p resultieren können. Für die Übertragung eines konti-

[2] $h = 40$ Oktetts (ein IP-Kopf besteht aus mindestens 20 Oktetts, ein UDP-Kopf besteht aus acht Oktetts und ein RTP-Kopf (Abb. 2.6) mindestens aus 12 Oktetts), $l = 8$ bits, $c = 1$, $f_s = 8$ kHz

[3] Network Voice Terminal, [Sch95a]

[4] Voice Audio Tool, Kap. 2.6.2

nuierlichen Audiosignals folgt daraus ein Takt f_p, mit dem der Sender Pakete generiert:

$$f_p = \frac{f_s}{n} = 50\,\mathrm{Hz} \tag{2.1}$$

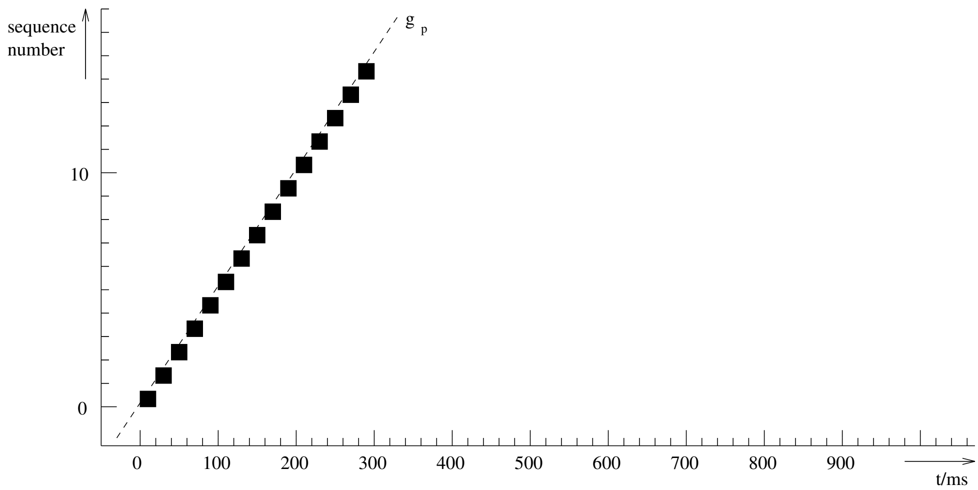

Abbildung 2.2: Senden von Audiodaten

Aus $n = 160$ resultiert eine Bitrate R_p von 80 kbit/s. Dagegen erfordern die Nutzdaten allein nur eine Bitrate R von 64 kbit/s.

Wie in der Abb. 2.2 zu sehen ist, kann eine Gerade g_p angegeben werden, die eine Funktion der Sequenznummern in Abhängigkeit von den Sendezeitpunkten darstellt:

$$g_p(t) = f_p t + g_p(0)$$

Dieser Abschnitt sollte zeigen, daß jedes Paket Daten enthält, die an einen festen Zeittakt gebunden sind. Für die Decodierung der Daten auf der Empfängerseite ist dieser Takt ebenfalls eine wesentliche Größe. Wenn die Zahl n der Abtastwerte pro Paket für jedes Paket gleich ist, so kann der Empfänger mit Hilfe des Taktes f_p arbeiten. Es ist aber auch möglich, daß sich die Paketgröße im Laufe einer Übertragung ändert. Dann dient die Abtastfrequenz f_s als Maß, das zur Synchronisation beim Empfänger verwendet werden muß.

2.4 Empfänger

Dem Empfänger, der die erhaltenen Daten ausspielen soll, müssen die Pakete im gleichen Takt f_p vorliegen, wie sie der Sender erzeugt hat. Aufgrund der Eigenschaften eines paketvermittelnden Netzwerkes kann der Empfänger aber nicht erwarten, daß die Pakete mit dem gleichen Takt bei ihm eintreffen wie sie ausgesendet worden sind. Wegewahlalgorithmen und Warteschlangen in den Netzknoten verursachen variable Netzwerkverzögerungen. Die Abbildung 2.3 soll verdeutlichen, daß die Ankunftszeitpunkte a_i nicht vorhersagbar sind. Es läßt sich keine Gerade $g_a(t)$ angeben, die eine Funktion der Sequenznummern in Abhängikeit von den Ankunftszeizpunkten a_i darstellt.

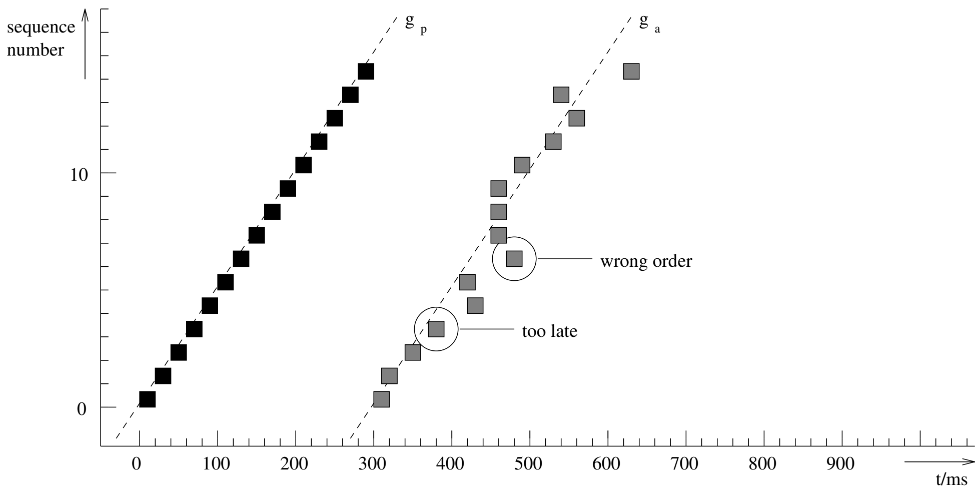

Abbildung 2.3: Variable Netzwerkverzögerung

Beginnt der Empfänger sofort mit der Decodierung und Ausspielung, wenn das erste Paket eintrifft, kommt es zum Verlust von Informationen. Alle Pakete, die dem ersten folgen, müssen mit dem Takt f_p zur Verfügung stehen. Pakete, die eine längere Verzögerung als das erste im Netzwerk erfahren haben, kommen in diesem Fall zu spät an. Sie sind nicht zum festgelegten Zeitpunkt verfügbar und gelten somit als verloren. Diese Situation ist in Abb. 2.3 gekennzeichnet, ebenso die Möglichkeit der falschen Paketfolge.

Beispielhaft ist in Abb. 2.4 der zeitliche Verlauf von ankommenden Paketen dargestellt. Bei der Abbildung handelt es sich um einen Ausschnitt einer realen Sprachübertragung von Amherst, Massachusetts (USA) nach Berlin (Deutschland).

2.5 Ausspielverzögerung

Um die variable Netzwerkverzögerung zu kompensieren, muß der Empfänger die erhaltenen Pakete zusätzlich verzögern, damit jedes Paket zur Ausspielung

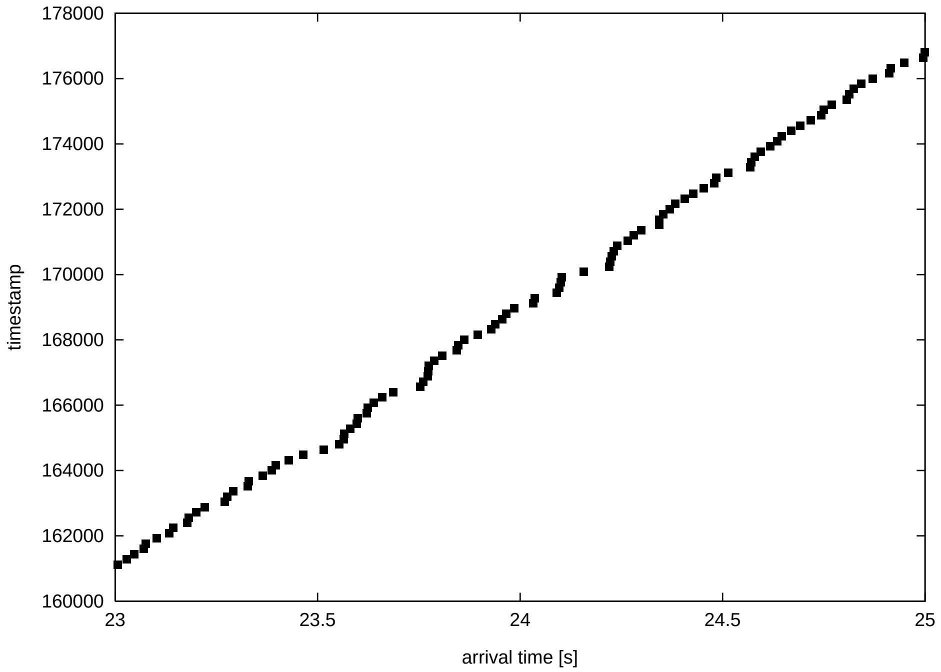

Abbildung 2.4: Ausschnitt aus einem Empfang von Audiodaten einer Weitverkehrsverbindung (Tab. 5.1, `erlang-medusa-1`). Dargestellt sind die empfangenen `timestamps` der RTP-Pakete (siehe Abb. 2.6) über der Ankunftszeit.

bereitsteht. Wenn der Empfänger den Beginn der Ausspielung mit dem ersten Paket verzögert, so verschieben sich die Ausspielzeitpunkte aller Pakete. Dadurch wird erreicht, daß Pakete, die eine längere Verzögerung innerhalb des Netzwerkes erfahren haben, trotzdem noch rechtzeitig zum Ausspieltermin bereitstehen.

In Abb. 2.5 wird die Ausspielung des ersten Paketes um 80 Millisekunden verzögert, wodurch alle Pakete mit dem Takt f_p ausgespielt werden können.

2.6 Echtzeit-Transportprotokolle

Die TCP/IP-Protokollfamilie [Com91] wurde für paketvermittelnde Kommunikationsnetzwerke entwickelt. Die standardisierten Protokolle unterstützen keine Sprach- oder Videoübertragungen.

Das UDP-Transportprotokoll ermöglicht die verbindungslose, ungesicherte Datagramm-Übertragung. Mit dem TCP-Protokoll wird der Dienst der gesicherten Übertragung eines Stroms von Oktetts angeboten. Keines der beiden Protokolle enthält Mechanismen, die den Transport von Echtzeit- und/oder CM-Daten (Continuous Media) kontrollieren.

Für die Übertragung von Echtzeit- und CM-Daten müssen zusätzliche Vereinbarungen zwischen Sender und Empfänger getroffen werden, die die vorhan-

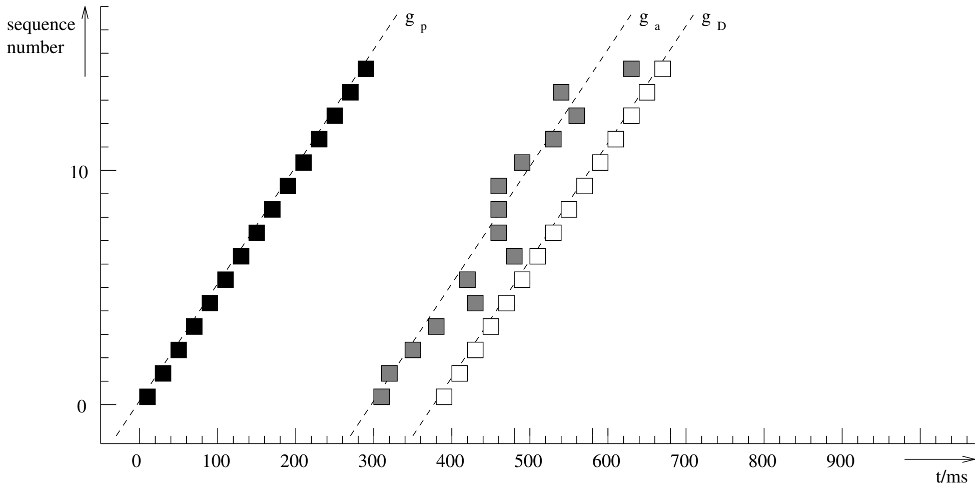

Abbildung 2.5: Verzögerung der Ausspielung

denen Protokolle nicht enthalten. Hierfür werden Echtzeit-Transportprotokolle entwickelt, die folgende Merkmale aufweisen:

- Vereinbarung über verwendetes Codierungsverfahren
- Generation von Zeitstempeln auf der Senderseite, die zur Rückgewinnung des Taktes (f_p bzw. f_s) von CM-Daten beim Empfänger nötig sind
- Kennzeichnung von markanten Punkten (z.B. Ruhephasen)
- Austausch zusätzlicher Informationen
  - Identifikation der Kommunikationspartner
  - Empfängerreport über Qualität der Übertragung
  - ...

2.6.1 RTP - Realtime Transport Protocol

Ein in der Entwicklung befindliches Protokoll ist das *Realtime Transport Protocol* (RTP) [SCFJ95]. Es handelt sich hierbei um eine sehr flexible Variante, die alle oben genannten Punkte abdeckt. RTP bietet somit die Grundfunktionalitäten eines Echtzeit-Transportprotokolls.

Der Entwurf sieht vor, daß für die verschiedenen Anwendungsmöglichkeiten das Protokoll geändert oder um Funktionalitäten ergänzt werden kann. Ergänzende Spezifikationen, die sogenannten *Profile*, dienen der Definition bestimmter Anwendungen. Ein Profil [Sch95c] dient z.B. der Beschreibung der Anwendung von RTP in Audio- und Videokonferenzsystemen.

Die Pakete des RTP-Protokolls werden meistens mit Hilfe von UDP transportiert. Dies wird aber nicht vorausgesetzt, auch andere Netzwerke oder Transportprotokolle können zur Anwendung kommen.

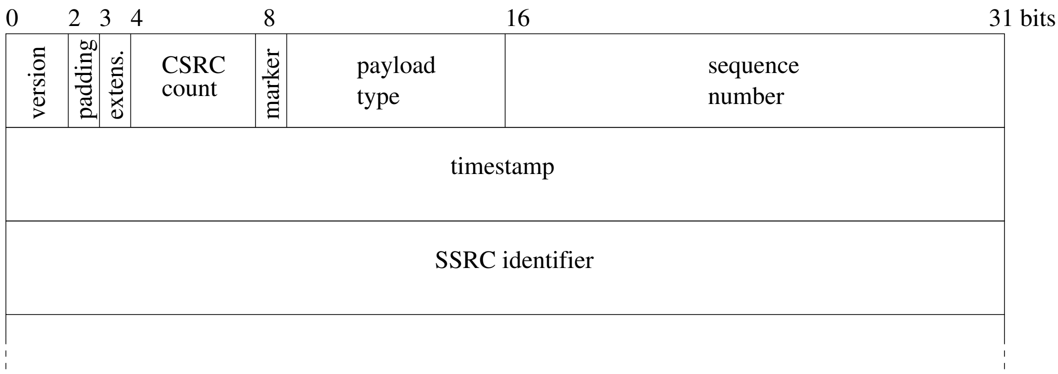

Abbildung 2.6: RTP-Paketkopf

Als nächstes seien hier die wichtigsten Elemente des RTP-Kopfes, dargestellt in Abb. 2.6, erklärt. Für diese Arbeit sind die Felder `marker`, `payload type`, `sequence number`, `timestamp` von Bedeutung.

Durch das Feld `sequence number` kann der Empfänger die richtige Paketreihenfolge wiederherstellen und Paketverluste erkennen. Der Sender führt mit diesem Wert eine fortlaufende Numerierung der Pakete durch.

Der `timestamp`-Wert liefert die Information über den Zeitpunkt der Abtastung des ersten Oktetts der Nutzdaten innerhalb des RTP-Paketes. Hier kann der Sender z.B. einen Zähler benutzen, der mit jedem Takt der Abtastfrequenz f_s (siehe Kap. 2.2) inkrementiert wird, wodurch die Abtastungen der Audiocodierung auf der Senderseite gezählt werden. Dieser Zähler läuft auch dann weiter, wenn keine Pakete versendet werden. Dies ist bei einer Audioübertragung dann der Fall, wenn das Audiosignal eine bestimmte Lautstärke unterschreitet. Der Sender führt zu diesem Zweck eine Ruheererkennung durch, indem er aus Audiosignalen, die unter einer Signalleistungsschwelle liegen (leise Lautstärken), keine Pakete generiert. Dadurch entstehen Übertragungspausen, in denen keine Pakete gesendet werden. Bei dem ersten Paket nach der Ruhephase wird das `marker`-bit gesetzt. Dadurch wird dem Empfänger explizit das Ende der Ruhephase bzw. der Beginn einer neuen Sprechphase mitgeteilt.

Durch das Feld `payload type` wird das Codierungsverfahren mitgeteilt; u.a. erhält der Empfänger dadurch den Wert der Abtastfrequenz f_s.

2.6.2 VAT - Voice Audio Tool

Das Audiokonferenz-Sytem *VAT* (Voice Audio Tool), entwickelt von Van Jacobson, benutzt ein eigenes Paketformat (Abb. 2.7), daß vergleichbare Informationen beinhaltet wie das RTP-Protokoll (Tabelle 2.1).

RTP	VAT
`marker`	`talkspurt`
`payload type`	`audio format`
`sequence number`	-
`timestamp`	`timestamp`

Tabelle 2.1: Übereinstimmende Felder der Paketköpfe des RTP- und des VAT-Protokolls

Durch das Fehlen der `sequence number` kann der Empfänger nicht eindeutig feststellen, ob es zu Paketverlusten kommt. Mit Hilfe der `timestamp`-Werte sind zwar Lücken erkennbar. Es kann aber nicht zwischen Lücken, die durch Übertragungspausen des Senders während einer Ruhephase entstehen, und Verlusten innerhalb des Netzwerkes unterschieden werden. Entdeckt der Empfänger eine Lücke gefolgt von einem Paket, bei dem das `talkspurt`-bit gesetzt ist, so kann der Empfänger davon ausgehen, daß diese durch eine Übertragungspause des Senders hervorgerufen wurde. Trotzdem können in dieser Situation auch nicht erkannte Verluste auftreten, weil dem Empfänger der Beginn der Übertragungspause unbekannt ist. In einer anderen Situation registriert der Empfänger fälschlicherweise Paketverluste, wenn das erste Paket nach der Übertragungspause, bei dem das `talkspurt`-bit gesetzt ist, verloren geht. Die richtige Paketreihenfolge kann aber durch die `timestamps` hergestellt werden.

Der Vergleich, der in Tabelle 2.1 dargestellt ist, bezieht sich nur auf die Anwendung von RTP in Audioübertragungen. Für andere Echtzeitdaten kann sich die Bedeutung der Felder, insbesondere `marker` und `payload type`, ändern.

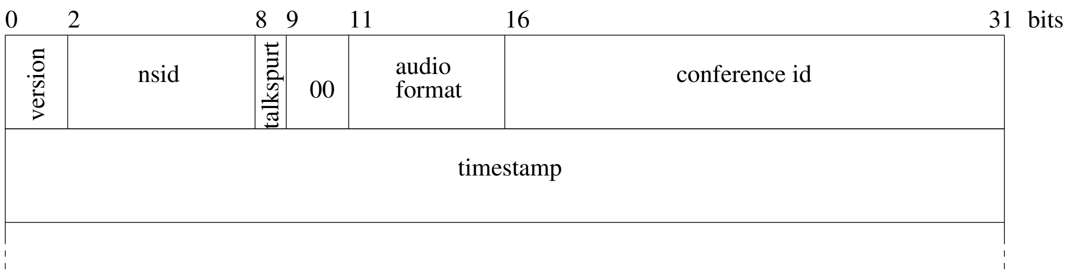

Abbildung 2.7: VAT-Paketkopf

Kapitel 3

Adaptive Ausspielverzögerung

Wie in den letzten Kapiteln begründet, ist es nötig, die empfangenen Daten verzögert auszuspielen. In diesem Kapitel wird die Bestimmung einer geeigneten Ausspielverzögerung behandelt.

Im Verlauf einer Übertragung kann sich der Zustand des Netzwerkes ändern. Deshalb wäre es ein Fehler, die Ausspielverzögerung für die gesamte Dauer der Übertragung auf einen Wert festzulegen. Benutzt ein Empfänger eine Verzögerung, die einen akzeptablen Empfang ermöglicht, kann sich diese Verzögerung einige Momente später als völlig ungeeignet erweisen, wenn sich z.B. die Belastung des Netzwerkes ändert. Eine geeignete Ausspielverzögerung darf nicht zu klein sein, um die Paketverluste gering zu halten. Gleichzeitig sollte sie aber auch nicht zu groß werden, um die Gesamtverzögerung nicht unnötig zu erhöhen. Aus diesem Grund wird das Verzögerungsverhalten des Netzwerkes ständig *beobachtet*.

Aufgrund der Eigenschaften von CM-Daten, ist die Veränderung der Ausspielverzögerung nur in bestimmten Situationen erlaubt. Wurde einmal mit der Ausspielung begonnen, so sind die Ausspielzeitpunkte der folgenden Pakete festgelegt.

Eine Möglichkeit der Anpassung besteht in Übertragungspausen, die durch Phasen der Ruhe hervorgerufen werden (siehe Kap. 2.6.1). Der Sender arbeitet mit einem Pausen-Erkennungsmechanismus und sendet in diesen Fällen keine Pakete. Es besteht aber auch die Möglichkeit, daß der Empfänger Ruhephasen entdeckt, wenn dies nicht auf der Senderseite geschieht.

Durch zeitliche Dehnung oder Stauchung der Ruhephase kann der Ausspielzeitpunkt, und somit die Ausspielverzögerung, der folgenden Sprechphase variiert werden. So setzt sich eine Audioübertragung aus einer Folge von Sprechphasen zusammen, und für jede Sprechphase wird auf der Empfängerseite eine eigene Ausspielverzögerung (siehe Kap. 2.5) bestimmt. In Abb. 3.1 ist diese Situation dargestellt. Ausgehend von diesem Zeitdiagramm werden Zeitgrößen eingeführt, die für die weiteren Betrachtungen wesentlich sind. Alle Größen werden in der Einheit von Zeitstempeln (`timestamp` eines RTP-Paketes, Kap. 2.6.1), d.h. in Abtastwerten, gemessen.

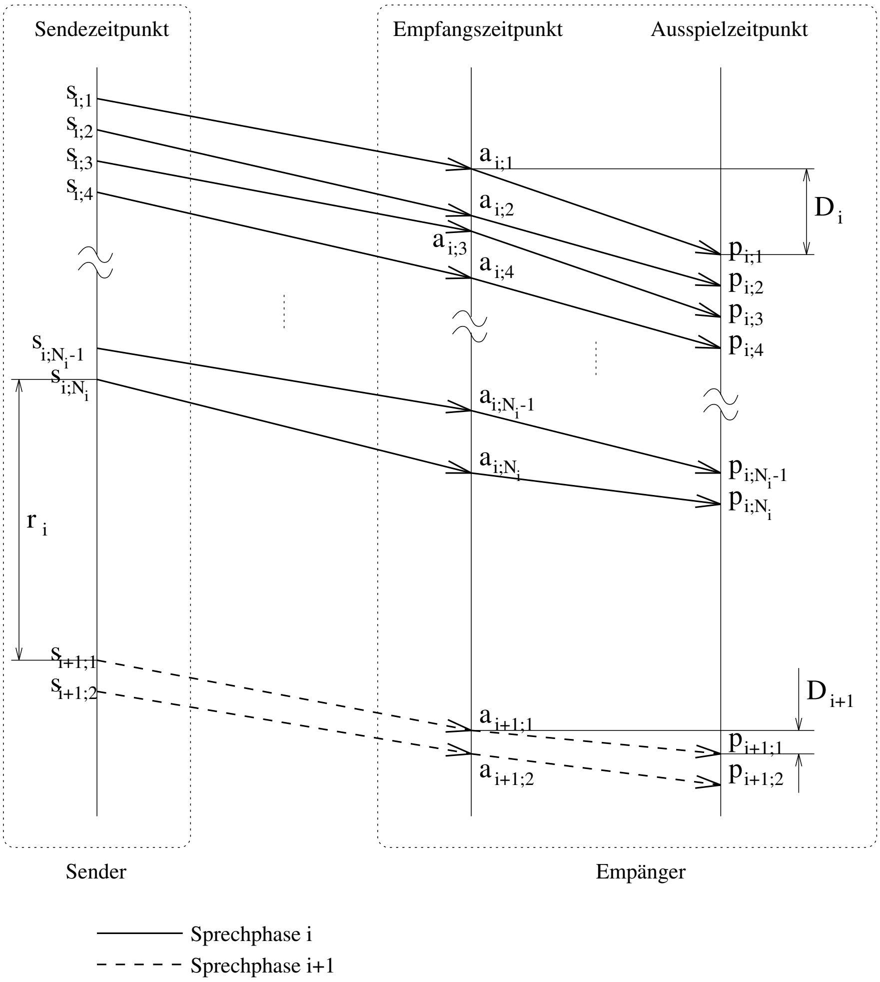

Abbildung 3.1: Zeitablaufdiagramm während eines Sprechphasenwechsels

N_i Anzahl der Pakete der Sprechphase i.

$s_{i;j}$ Zu dem Zeitpunkt $s_{i;j}$ $(1 \leq j \leq N_i)$ wird das Paket j von der Sprechphase i beim Sender generiert.

$t_{i;j}$ Zeitstempel des Paketes j der Sprechphase i (`timestamp` eines RTP-Paketes, 2.6.1); Sender und Empfänger arbeiten nicht mit einer gemeinsamen Uhr. Der Empfänger kennt durch die Werte $t_{i;j}$ der aufeinanderfolgenden Pakete die relativen Sendezeitpunkte. Die absoluten Sendezeitpunkte $s_{i;j}$ sind dem Empfänger unbekannt.

$a_{i;j}$ Ankunftszeit von Paket j der Sprechphase i auf der Empfängerseite.

$d_{i;j}$ Relative Netzwerkverzögerung, die das Paket j der Sprechphase i erfahren hat. Der Wert $d_{i;j}$ berechnet sich aus der Subtraktion $a_{i;j} - t_{i;j}$. Weil $t_{i;j}$ nicht dem tatsächlichen Sendezeitpunkt entspricht, ist auch der Wert der wirklichen Netzwerkverzögerung für den Empfänger nicht bestimmbar. Mit der Größe $d_{i;j}$ kann nur eine Aussage über die Veränderung der Netzwerkverzögerung getroffen werden.

$p_{i;j}$ Ausspielzeitpunkt der Audiodaten von Paket j der Sprechphase i beim Empfänger.

r_i Dauer der Ruhephase, die der Sprechphase i folgt.

D_i Ausspielverzögerung der Sprechphase i.

$\hat{d}_k$ Folge von Abschätzungen der Netzwerkverzögerung.

$\hat{v}_k$ Folge von Abschätzungen der Variabilität der Netzwerkverzögerung.

Zum Zeitpunkt $s_{i;1}$ wird das erste Paket der Sprechphase i generiert. Die folgenden Pakete werden beim Sender in einem festen Takt erzeugt. Wenn die Audiocodierung mit der Abtastfrequenz f_s arbeitet und ein Paket n Abtastwerte enthält, ergibt sich der Takt f_p, mit dem die Pakete erzeugt werden:

$$f_p = \frac{f_s}{n}$$

Daraus folgt, daß der Abstand zwischen zwei Paketen beim Sender konstant ist[1]:

$$s_{i;j+1} - s_{i;j} = f_s \frac{1}{f_p} \quad \text{mit } 1 \leq j \leq N_i$$

Die gleiche Beziehung gilt für die relativen Sendezeitpunkte:

$$t_{i;j+1} - t_{i;j} = f_s \frac{1}{f_p} \quad \text{mit } 1 \leq j \leq N_i \tag{3.1}$$

[1]Die Multiplikation mit der Abtastfrequenz f_s ist notwendig, weil die Größen $s_{i;j}$ in der Zahl von Abtastwerten gemessen werden.

Mit dem gleichen Takt müssen die Audiodaten auch auf der Empfängerseite ausgespielt werden. Weil die Pakete nicht in diesem festen Takt f_p beim Empfänger eintreffen, wird das erste Paket einer Sprechphase verzögert ausgespielt. Das erste Paket wird für die Dauer D_i beim Empfänger zwischengespeichert. Durch die Zeitstempel, die mit jedem Paket mitgeführt werden, werden die Ausspielzeitpunkte der folgenden Pakete bestimmt.

- Ausspielzeitpunkt des ersten Paketes der Sprechphase i:

$$p_{i;1} = a_{i;1} + D_i \tag{3.2}$$

- Ausspielzeitpunkte der Pakete der Sprechphase i, die dem ersten Paket der Sprechphase i folgen:

$$p_{i;j} = p_{i;1} + t_{i;j} - t_{i;1} \quad \text{mit } 2 < j < N_i \tag{3.3}$$

3.1 Adaptive Algorithmen

In dieser Arbeit werden vier verschiedene Algorithmen untersucht. In den folgenden Abschnitten werden sie vorgestellt und die ihnen zu Grunde liegenden Ideen erläutert. Die von R. Ramjee et al. [RKTS94] entworfenen Algorithmen basieren auf der *absolute timing method*, die von W. A. Montgomery [Mon83] beschrieben wurde.

Ziel der Anwendung der vier Algorithmen ist die Schätzung der zukünftigen Netzwerkverzögerung und ihrer Variabilität aus der Beobachtung der gemessenen Netzwerkverzögerungen $d_{i;j}$. In den folgenden vier Abschnitten werden die verschiedenen Methoden zur Bestimmung der Schätzwerte für die Netzwerkverzögerung $\hat{d}_k$ und der Variabilität $\hat{v}_k$ vorgestellt. Mit jedem erhaltenen Paket werden die Werte $\hat{d}_k$ und $\hat{v}_k$ aus der Messung von $d_{i;j}$ und der letzten Abschätzung $k-1$ bestimmt. Mit den Abschätzungen wird zum Beginn einer Sprechphase die Ausspielverzögerung D_i berechnet. Die Ausspielverzögerung ist die Zeitdauer $p_{i;1} - a_{i;1}$ mit der der Empfänger das erste Paket der Sprechphase i zwischenspeichert.

$$D_i = \hat{d}_k + 4\hat{v}_k - d_{i;1}$$

Die vier Algorithmen unterscheiden sich in der Berechnung von $\hat{d}_k$. In den folgenden Abschnitten werden die Berechnungsmethoden erklärt. Die Bestimmung von $\hat{v}_k$ ist für die ersten drei Algorithmen identisch:

$$\begin{aligned} \alpha &= 0.998002 \\ \hat{v}_k &= \alpha\, \hat{v}_{k-1} + (1-\alpha)|\hat{d}_k - d_{i;j}| \end{aligned}$$

3.1.1 Algorithmus 1 (rtt)

Der erste Algorithmus wurde aus dem Standard des *Transmission Control Protocol* [Pos81] entnommen. In TCP wird damit der Zeitgeber (RTT) für die wiederholte Sendung berechnet. Sowohl in TCP, als auch in unserem Fall, soll aus der Betrachtung der Netzwerkverzögerung, der bereits empfangenen Pakete, ein Wert für die momentane Verzögerung berechnet werden:

$$\begin{aligned} \alpha &= 0.998002 \\ \hat{d}_k &= \alpha\,\hat{d}_{k-1} + (1-\alpha)d_{i;j} \end{aligned}$$

Dieser Algorithmus wird aufgrund seiner Herkunft im weiteren Verlauf der Arbeit rtt-Algorithmus genannt.

3.1.2 Algorithmus 2 (srt)

Der zweite Algorithmus ist eine Erweiterung des ersten. Die Schätzung von $\hat{d}_k$ verhält sich bei einer Erhöhung der gemessenen Netzwerkverzögerung anders als bei einer Verminderung. Im Falle der Erhöhung liegt eine höhere Gewichtung auf der gemessenen Verzögerung $d_{i;j}$.

$$\alpha = 0.998002 \tag{3.4}$$
$$\beta = 0.75 \tag{3.5}$$
$$\hat{d}_k = \begin{cases} \beta\,\hat{d}_{k-1} + (1-\beta)d_{i;j} & \text{falls } d_{i;j} > \hat{d}_{k-1}, \\ \alpha\,\hat{d}_{k-1} + (1-\alpha)d_{i;j} & \text{sonst} \end{cases} \tag{3.6}$$

Weil Algorithmus 2 eine sensitive Variante des rtt-Algorithmusses darstellt, wird er als der srt-Algorithmus bezeichnet (sensitive rtt).

3.1.3 Algorithmus 3 (min)

Ein weiterer Ansatz resultiert aus der Betrachtung der Signalausbreitungsgeschwindigkeit. Aufgrund der Zeit, die für die reine Signalausbreitung nötig ist, erfährt jedes Paket eine Mindestverzögerung d_{prop} (propagation delay). Der min-Algorithmus ermittelt die geringste Verzögerung, die innerhalb einer Sprechphase gemessen wurde. Mit $\hat{d}_k$ soll d_{prop} approximiert werden:

$$\hat{d}_k \approx d_{prop}$$

$$\hat{d}_k = \begin{cases} d_{i;j} & \text{falls } d_{i;j} < \hat{d}_{k-1}, \\ \hat{d}_{k-1} & \text{sonst} \end{cases}$$

3.1.4 Algorithmus 4 (imp)

Die Verzögerung des Netzwerkes kann sich kurzzeitig sehr erhöhen. Dann wird eine gewisse Zeit lang kein Paket empfangen, und anschließend kommen die Pakete fast gleichzeitig an. Dieses Verhalten soll der vierte Algorithmus berücksichtigen. Es werden zwei Modi der Verarbeitung unterschieden:

Normal-Modus Die Pakete kommen mit der durchschnittlichen varianten Verzögerung beim Empfänger an. Der rtt-Algorithmus wird, unter Verwendung eines anderen Wertes für α, benutzt.

$$\begin{aligned} \alpha &= 0.875 \\ \hat{d}_k &= \alpha\, \hat{d}_{k-1} + (1-\alpha) d_{i;j} \end{aligned}$$

Impuls-Modus Nachdem eine Erhöhung der Netzwerkverzögerung festgestellt wurde, wird der folgende *Impuls*, wobei die Pakete in sehr kurzen Zeitabständen eintreffen, abgearbeitet. Die Änderung des Wertes $\hat{d}_i$ ist mit der Änderung von $d_{i;j}$ direkt gekoppelt:

$$\hat{d}_k = \hat{d}_{k-1} + d_{i;j} - d_{i;j-1}$$

In beiden Modi berechnet sich $\hat{v}_k$ mit der Gleichung:

$$\begin{aligned} \beta &= 0.875 \\ \hat{v}_k &= \beta\, \hat{v}_{k-1} + (1-\beta)|\hat{d}_k - d_{i;j}| \end{aligned}$$

Für die Unterscheidung zwischen den beiden Modi muß eine impulsartige Veränderung der Netzwerkverzögerung $d_{i;j}$ entdeckt werden. Der Wechsel vom Normal-Modus zum Impuls-Modus vollzieht sich, wenn die Änderung von $d_{i;j}$ zwischen zwei aufeinanderfolgenden Paketen eine bestimmte Schwelle übersteigt.

$$|d_{i;j} - d_{i;j-1}| > 2|\hat{v}_k| + 800$$

Während des Impuls-Modus wird mit jedem Erhalt eines Paketes ein Wert `var` berechnet (siehe Abb. 3.2). Er ist ein Maß für die Veränderung der Netzwerkverzögerung $d_{i;j}$. Unterschreitet `var` eine bestimmte Schwelle, so wird wieder in den Normal-Modus gewechselt.

3.2 Grenzen der Adaption

Die Ausspielverzögerung D_i kann nicht beliebig variiert werden. Mit dem Beginn der Sprechphase i existiert eine minimale Ausspielverzögerung; sie hängt von der Ausspielverzögerung D_{i-1} und von der Dauer der Ruhephase r_{i-1} ab:

$$D_i > D_{i-1} + r_{i-1} \tag{3.7}$$

```
if (mode == NORMAL) {
  if (abs($d_{i;j}$ - $d_{i;j-1}$) > abs($\hat{v}_k$) * 2 + 800) {
    var = 0;
    mode = IMPULSE;
  }
else {
  var = var / 2 + abs((2 * $d_{i;j}$ - $d_{i;j-1}$  - $d_{i;j-2}$) / 8);
  if (var <= 63) {
    mode = NORMAL;
    $d_{i;j-2}$ = $d_{i;j-1}$;
    $d_{i;j-1}$ = $d_{i;j}$;
    return;
  }
}
if (mode == NORMAL)
  $\hat{d}_k$ = 0.125 * $d_{i;j}$ + 0.875 * $\hat{d}_{k-1}$;
else
  $\hat{d}_k$ = $\hat{d}_{k-1}$ + $d_{i;j}$ - $d_{i;j-1}$;
$\hat{v}_k$ = 0.125 * abs($d_{i;j}$ - $\hat{d}_k$) + 0.875 * $\hat{v}_{k-1}$;
$d_{i;j-2}$ = $d_{i;j-1}$;
$d_{i;j-1}$ = $d_{i;j}$;
return;
```

Abbildung 3.2: Implementierung des vierten Algorithmusses in C-ähnlicher Notation

Wird eine geringere Ausspielverzögerung benutzt, so kommt es zur Überlappung der beiden Sprechphasen $i-1$ und i.

Aus den Anforderungen, gezeigt in Kapitel 1, folgt, daß die Verzögerung nicht zu groß werden darf, um eine interaktive Nutzung zu ermöglichen. Daher ist es sinnvoll, den Wert D_i zu begrenzen. Ein weiterer Grund ist, daß der Zwischenspeicher für die empfangenen Pakete eine statische Größe hat, wodurch eine maximale Grenze für D_i vorgegeben ist.

3.3 Adaption innerhalb von Sprechphasen

Innerhalb einer Sprechphase i kann die zu Beginn bestimmte Ausspielverzögerung D_i sich als ungeeignet erweisen. Der Zustand des Netzwerkes kann sich während des Empfangs einer längeren Sprechphase derartig verändern, daß die Pakete längere Verzögerungen erfahren als angenommen. In diesem Fall kämen viele Pakete nicht zur Ausspielung.

Um nicht zu hohe Verluste durch zu spät ankommende Pakete während einer längeren Sprechphase hinnehmen zu müssen, werden solche Pakete sofort ausgespielt; es wird also zu einem späteren Zeitpunkt ausgespielt, als nach Gleichung 3.3 vorgeschrieben wäre, wodurch eine Pause innerhalb der Sprechphase entsteht, was als Fehler in der akustischen Ausspielung hörbar wird.

Es besteht aber damit die Chance, daß sich der Empfänger an eine allgemeine Erhöhung der Netzwerkverzögerung anpaßt, wenn die Planungen der folgenden Pakete auf dem verzögerten Ausspielzeitpunkt des zu spät angekommenen Paketes basieren.

Kapitel 4

Simulation des Empfängers

Die in Kapitel 3 vorgestellten Algorithmen sollen laut Aufgabenstellung auf ihre Leistungsfähigkeit getestet werden. Dabei ist es zweckmäßig, die Tests unter möglichst realen Bedingungen, d.h. praxisnahe, durchzuführen. Um dies zu erreichen, werden Audiodaten über Netzwerke übertragen. Die empfangenen Pakete werden in Dateien aufgezeichnet. Zusätzlich wird zu jedem Paket der Ankunftszeitpunkt gespeichert, wodurch das Verzögerungsverhalten des Netzes festgehalten ist. Diese Aufzeichnungen erlauben somit die Reproduktion des Netzwerkverhaltens. Jeder Algorithmus kann in der Simulation unter identischen Bedingungen getestet werden.

Die Empfangssituation wird mit Hilfe von Ptolemy (Abs. 4.2) simuliert. Das Netzwerk wird mit Hilfe der Aufzeichnungsdateien nachgebildet, indem die Pakete zu den gespeicherten Ankunftszeitpunkten dem simulierten Empfänger übergeben werden.

4.1 Aufzeichnung von Übertragungen

Um den Einfluß verschiedener Faktoren auf das Verhalten der Algorithmen zu testen, ist es erforderlich, mehrere Aufzeichnungen zu erstellen. Es werden gleiche akustische Signalfolgen, wie z.B. der Monolog eines Radiosprechers, durch verschiedene Netztypen zu unterschiedlichen Tageszeiten übertragen.

Die Aufzeichnungen werden mit dem Programm `rtpdump` erstellt. Das Programm `rtpdump`[1] dient der Analyse und Aufzeichnung des Empfangs einer RTP-Übertragung auf einem Paar von UDP-Ports. Unter Verwendung der Option `-w` erzeugt `rtpdump` eine binäre Datei. Die Datei enthält einen Kopf, der die IP-Adressen vom Sender und Empfänger ([Com91], Kap. 4, S. 61), die Startzeit und die Version von `rtpdump` beinhaltet. Die erste Zeile ist lesbar und so aufgebaut, daß bei Aufruf der Aufzeichnungsdatei das Programm `rtpplay` gestartet wird. Gleichzeitig liefert die Zeile die Information über Adresse und Port, die bei

[1] `rtpdump` ist über anonymous-FTP erhältlich:
`ftp://gaia.cs.umass.edu/pub/hgschulz/rtp/rtpdump-1.0.tar.gz`

der Aufzeichnung benutzt worden sind. Bei einer Übertragung mit der *loopback*-Adresse als Zieladresse und dem UDP-Port 3456 steht folgende Zeile am Anfang der Datei:

```
#!rtpplay1.3 127.0.0.1/3456
```

Die folgenden Inhalte sind binär. Zunächst ist eine Struktur abgespeichert, die den Startzeitpunkt des Empfangs, die Senderadresse und den Port angibt:

```
typedef struct {
  struct timeval start;  /* start of recording (GMT) */
  u_int32 source;        /* network source (IP addr) */
  u_int16 port;          /* UDP port                 */
} RD_hdr_t;
```

Mit der Struktur `RD_hdr_t` ist der Kopf der Datei abgeschlossen. Dann folgen die empfangenen Pakete, die komplett mit Zusatzinformationen gespeichert werden. RTP-Übertragungen werden auf einem Port-Paar empfangen. Durch welchen Port die jeweiligen Pakete angekommen sind, sollte auch in der Aufzeichnung erkennbar bleiben. Über den einen Port werden die Pakete mit den Audiodaten empfangen, der andere empfängt RTCP-Pakete, die begleitende Übertragungsinformationen enthalten. Außerdem ist die Länge der einzelnen Pakete wichtig für das spätere Auslesen. Und es soll zu jedem Paket der Ankunftszeitpunkt festgehalten werden. Die Struktur `RD_packet_t` enthält diese Informationen; sie wird vor jedem Paket in die Datei geschrieben:

```
typedef struct {
  u_int16 length;      /* length of packet, including this header   */
  u_int16 type;        /* RTP or RTCP                               */
  u_int32 offset;      /* microseconds since the start of recording */
} RD_packet_t;
```

Das Programm **`rtpdump`** wurde für die Erstellung der Aufzeichnungen modifiziert. Das Originalprogramm speichert die relativen Ankunftszeitpunkte in Millisekunden. Die hier verwendete Version arbeitet mit Mikrosekunden.

4.2 Ptolemy

Ptolemy [Uni94a] ist ein umfangreiches, graphisches Simulationswerkzeug, das für verschiedene Anforderungen geeignete Umgebungen, Domains genannt, bereitstellt:

- Ereignisorientierte Simulation (Kommunikationsnetzwerke)
- digitale Signalverarbeitung

- analoge Signalverarbeitung
- Simulation digitaler Schaltungen
- DSP-Programmgeneration

Die Liste ist sicher nicht vollständig, da Ptolemy ein System ist, das sich fast beliebig erweitern läßt. Der Benutzer soll durch Hinzufügen von eigenen Modulen die Umgebungen an seine Bedürfnisse anpassen und erweitern. Das System erlaubt auch die Erstellung von völlig neuen Umgebungen.

Für Simulationen von Kommunikationsnetzwerken bzw. -protokollen eignen sich die Umgebungen: CP-Domain (concurrent process) oder DE-Domain (discrete event). Jedes prozessorientierte Modell läßt sich auch in ein ereignisgesteuertes Modell umformen [SH95]. Für die Simulation der Empfangssituation ist die ereignisgesteuerte Variante gewählt worden.

4.3 Aufbau der Simulation

Die Simulation gliedert sich in drei Teile. Die Abbildung des Netzwerkes wird durch das Lesen der erstellten Aufzeichnungen erreicht. In der zweiten Komponente ist ein Modell des Empfängers enthalten, der die Berechnungen nach den vorgestellten Algorithmen durchführt und, wie in der Realität, eine akustische Ausspielung des empfangenen Signals ermöglicht. Die letzte Komponente dient der Aufnahme von statistischen Werten des Empfangsverhaltens und besteht aus verschiedenen Modulen, die unterschiedliche Meßergebnisse protokollieren.

Abb. 4.1 zeigt das erstellte Modell in Form einer Ptolemy-Galaxie. Der Stern `DEReadRTPTrace` liest die Aufzeichnungsdateien und bildet die erste Komponente. Der Verbund der Sterne `DERTPScheduler` und `DERTPOut` stellt den Empfänger dar, wobei `DERTPScheduler` die Berechnungen durchführt, d.h. die Ausspielzeitpunkte der ankommenden Pakete bestimmt. Das Zwischenspeichern und die akustische Ausspielung werden von dem Modul `DERTPOut` übernommen. Die übrigen Module gehören der letzten Komponente an. In den nächsten Abschnitten werden die Funktionen der einzelnen Module erörtert.

4.3.1 RTP-Pakete

Das Modell beschreibt im wesentlichen die beiden Ereignisse: Empfang eines RTP-Paketes und Ausspielung der Nutzdaten des Paketes. Zum Zweck der Anschaulichkeit ist die Form von RTP-Paketen in Ptolemy integriert worden.

Ptolemy erlaubt die Definition von eigenen Datentypen, die als Partikel zwischen den Sternen versendet werden können. So ist es z.B. möglich, RTP-Pakete (siehe Abb. 2.6) [SCFJ95, Sch95c] in Simulationen zu verwenden. Hierfür muß eine neue `Message`-Klasse abgeleitet [Uni94b, Kap. 4.5.1] werden.

Für die Simulation ist die Klasse `RTPPacket` erstellt worden. Der Konstruktor von `RTPPacket` übernimmt die Informationen aus den Aufzeichnungsdateien,

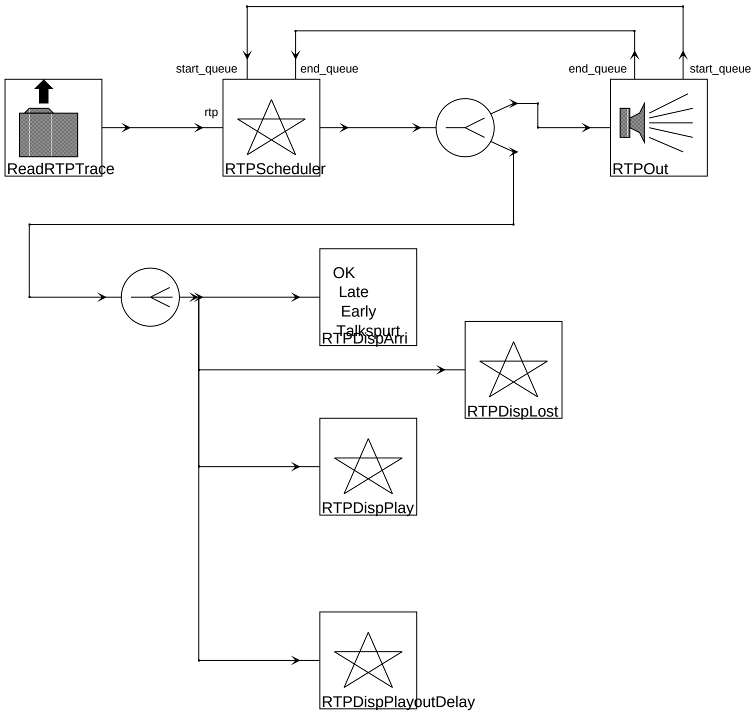

Abbildung 4.1: Galaxie des RTP-Empfängers

um die RTP-Pakete zu initialisieren. In der Tabelle 4.2 sind die Elementfunktionen für den Zugriff auf die jeweiligen Felder eines RTP-Paketes aufgelistet.

Die Klasse `RTPPacket` enthält zusätzlich zu den Informationen eines RTP-Paketes Elemente, wie `arrival_time`, `play_time`, `delay` und `status`, die für Berechnungen innerhalb der Simulation verwendet werden. Bei der Initialisierung wird in `arrival_time` die lokale Ankunftszeit geschrieben und die Variable `status` auf `RTP_UNDEFINED` gesetzt. Erst nach Ermittlung eines Ausspielzeitpunktes (`play_time`), kann zwischen den drei Zuständen `RTP_OK`, `RTP_LATE` und `RTP_EARLY` unterschieden werden (siehe Tab. 4.1).

Die Simulation setzt die Verwendung der Klasse `RTPPacket` nicht voraus. Die Ein- und Ausgänge der Sterne akzeptieren alle Klassen, die von der Klasse `Message` abgeleitet sind. Durch den Vererbungsmechanismus, in Form von Klassenableitungen, die die Sprache C++ liefert [Lip90], können leicht andere Paketformen unterstützt werden. Voraussetzung ist, daß abgeleitete `Message`-Klassen erstellt werden, die die gleichen Elementfunktionen besitzen wie `RTPPacket` (Tab. 4.2). Um Untersuchungen anhand von VAT-Übertragungen zu ermöglichen, muß die Simulation lediglich um eine Klasse, z.B. `VATPacket`, erweitert werden; sonst sind keine Änderungen erforderlich.

Status	Bedeutung
RTP_OK	Für das Audiosignal des Paketes konnte ein Ausspielzeitpunkt ermittelt werden.
RTP_LATE	Der ermittelte Ausspielzeitpunkt liegt in der Vergangenheit.
RTP_EARLY	Der ermittelte Ausspielzeitpunkt liegt zu weit in der Zukunft, er paßt nicht in den Zwischenspeicher von `DERTPOut`.
RTP_LATE_TALKSPURT	Ein zu spät angekommenes Paket, mit dem eine neue Sprechphase erzeugt wird (siehe Abs. 3.3).
RTP_UNDEFINED	Initialer Wert, der vom Modul `DEReadRTPTrace` gesetzt wird. Erst mit Behandlung durch `DERTPScheduler` erhält das Paket einen definierten Status.

Tabelle 4.1: Zustände eines RTP-Paketes innerhalb der Simulation

Funktionsname	Beschreibung
`GetType()`	Verwendete RTP-Vesion
`GetP()`	Inhalt des *padding* Bits
`GetCc()`	Anzahl der nach dem Kopf folgenden *CSRC identifiers*
`GetX()`	Inhalt des *extension* Bits
`GetM()`	Inhalt des *marker* Bits
`GetPt()`	Identifizierung des Formats der Nutzdaten
`GetSeq()`	Sequenznummer des Paketes
`GetTs()`	Zeitstempel des ersten Oktetts der Nutzdaten
`GetBuf()`	Zeiger auf den Anfang des Speicherbereichs, der die Nutzdaten enthält
`GetLen()`	Länge der Nutzdaten
`SetPlayTime()`	Ausspielzeit bestimmen
`GetPlayTime()`	Ausspielzeit abfragen
`SetStatus()`	Paketstatus setzen
`GetStatus()`	Paketstatus abfragen
`SetArrivalTime()`	Ankunftszeit des Paketes setzen
`GetArrivalTime()`	Ankunftszeit abfragen
`SetNetDelay()`	Netzwerkverzögerung setzen
`GetNetDelay()`	Netzwerkverzögerung abfragen

Tabelle 4.2: Elementfunktionen der `RTPPacket`-Klasse

4.3.2 Netzwerk

In der erstellten Simulation ist nur der Empfang von Paketen von Bedeutung. Das Verhalten des Netzwerkes kann durch die Aufzeichnungsdateien rekonstruiert werden. Das Modul `DEReadRTPTrace` dient als Quelle von Ereignissen, in Form von Nachrichten des Typs `RTPPacket`. Die Nachrichten werden zu den mitgeschnittenen relativen Ankunftszeitpunkten (Komponente `offset` der Struktur `RD_packet_t`, Abs. 4.1) an das Modul `DERTPScheduler` übergeben. Somit wird innerhalb der Simulation das gleiche zeitliche Verhalten erzeugt, wie es während der realen Übertragung existierte. Nach jedem Weiterleiten eines Paketes, stößt sich der Stern mit der Funktion `refireAtTime()` wieder selbst an, um das nächste Paket aus der Datei zu lesen.

In der DE-Domain kommt ein Stern nur zur Ausführung, wenn ein Ereignis am Eingang anliegt. Um Quellen erstellen zu können, die keine Eingänge besitzen, existiert die Klasse `DERepeatStar`. Die Klasse stellt im wesentlichen die Funktion `refireAtTime()` zur Verfügung, die es einem Stern ermöglicht, sich selbst zu einem bestimmten Zeitpunkt wieder zur Ausführung zu bringen. `DEReadRTPTrace` ist eine Ableitung der Klasse `DERepeatStar`.

4.3.3 Empfänger

Der Empfänger ist durch den Verbund der Sterne `DERTPScheduler` und `DERTPOut` realisiert. In `DERTPSchedulder` werden die Ausspielzeitpunkte abhängig vom gewählten Algorithmus und vom Zustand des Empfangsspeichers, enthalten in `DERTPOut`, berechnet. Pakete, deren Ausspielzeitpunkt bestimmt ist, werden an `DERTPOut` weitergeleitet. Dort werden sie in einen *Ringspeicher* einsortiert. Aus dem Ringspeicher werden die Audiodaten unabhängig vom Empfangszustand ausgespielt. Mit jedem Ausspieltakt meldet `DERTPOut` dem Stern `DERTPScheduler` den aktuellen Zustand des Ringspeichers, d.h. das Zeitintervall, das für die Planung weiterer Ausspielzeitpunkte zur Verfügung steht. Die untere Grenze ist der Termin der nächsten Ausspielung. Die obere Grenze ist durch die Größe des statischen Ringspeichers festgelegt. Für die Überprüfung des Intervalls stehen in `DERTPScheduler` die beiden Eingänge `start_queue` und `end_queue` zur Verfügung.

4.3.3.1 DERTPScheduler

Für jedes ankommende Paket wird der Ausspielzeitpunkt ermittelt, oder es wird festgestellt, daß eine Ausspielung nicht möglich ist, da es nicht in das Intervall, beschrieben durch `start_queue` und `end_queue`, paßt. Der Stern `DERTPScheduler` arbeitet mit den in Tab. 4.3 dargestellten Variablen.

Beginn der Übertragung. Der Ausspielzeitpunkt des ersten Paketes einer Übertragung bestimmt sich aus seinem Ankunftszeitpunkt `arrivalTime`

Variablenname	Bedeutung
ts_stamp	Zeitstempel des ersten Paketes der aktuellen Sprechphase
ts_time	Beginn der Ausspielung der aktuellen Sprechphase
begin_stamp	Zeitstempel des aktuellen Paketes
play_time	Zu planender Ausspielzeitpunkt des aktuellen Paketes
sample_cnt	Anzahl der Abtastwerte des aktuellen Paketes
last_inserted_time	Spätester Zeitpunkt, der bisher geplant wurde
last_stamp	Größter Zeitstempel der bisher geplant wurde
fa	Abtastfrequenz f_s
ed	$\hat{d}_k$
ev	$\hat{v}_k$

Tabelle 4.3: Variablen innerhalb des Moduls `DERTPScheduler`

und der initialen Ausspielverzögerung D_1. Eine initialer Wert für die Ausspielverzögerung ist nötig, da noch keine Schätzwerte für $\hat{d}_k$ und $\hat{v}_k$ vorliegen. Der Wert D_1 ist als der Parameter `initial_delay` vorhanden. Der Ausspielzeitpunkt bestimmt sich gemäß der Gleichung 3.2:

```
play_time = arrivalTime + initial_delay;
```

Planung der Ausspielzeitpunkte innerhalb von Sprechphasen. Zunächst wird der Ausspielzeitpunkt laut Gleichung 3.3 ermittelt:

```
play_time = ts_time + begin_stamp - ts_stamp;
```

Anschließend wird untersucht, ob das Paket zu spät angekommen ist. Dies ist der Fall, wenn `play_time` kleiner als `start_queue` ist, d.h., daß der berechnete Ausspielzeitpunkt `play_time` in der Vergangenheit liegt; die Ausspielung innerhalb des Moduls `DERTPOut` ist schon weiter vorangeschritten. Wenn der Empfänger mit dem Verfahren arbeitet, das bei zu spät angekommenen Paketen mit dem Beginn einer neuen Sprechphase reagiert, vollzieht sich die weitere Behandlung wie in Abs. 4.3.3.1 beschrieben. Ansonsten erhält das Paket den Status `RTP_LATE`, und das Modul `DERTPOut` wird das Paket ignorieren.

Ein Paket erhält auch dann den Status `RTP_LATE`, wenn es einer früheren Sprechphase zuzuordnen ist. Dies ist der Fall, wenn `play_time` kleiner als `ts_time` ist.

Für die zu planende Ausspielung existiert eine obere Grenze (`end_queue`); sie ist durch die Größe des Zwischenspeichers innerhalb `DERTPOut` und dem nächsten

Ausspieltermin (`start_queue`) festgelegt. Wird sie überschritten, erhält das Paket den Status `RTP_EARLY`.

Ist der Ausspielzeitpunkt ein möglicher Ausspieltermin der Ausspielung, so wird das Paket mit dem Status `RTP_OK` an das Modul `DERTPOut` weitergereicht.

Beginn einer neuen Sprechphase. Liefert die Funktion `rtp->GetM()` eine Eins, d.h. das Marker-Bit des RTP-Paketes ist auf eins gesetzt, so beginnt mit diesem Paket eine neue Sprechphase. In dieser Situation kann eine neue Ausspielverzögerung festgelegt werden. Dies passiert innerhalb der Funktion `NewTalkspurt`. Zunächst wird gemäß der Gleichungen 3.2 und 3.4 der Ausspielzeitpunkt des Paketes berechnet, wobei die Variablen `ed`, `ev` und `delay` den Werten $\hat{d}_k$, $\hat{v}_k$ und $d_{i;j}$ entsprechen.

```
play_time = arrivalTime + (ed + (4 * ev) - delay);
```

Mit der Überprüfung, ob `play_time` innerhalb der Grenzen von `start_queue` und `end_queue`, die den Zustand des Ringspeichers beinhalten, und den Grenzen, beschrieben in Kap.3.2, wird eventuell eine Modifikation von `play_time` nötig.

4.3.4 Aufnahme von statistischen Werten

Die letzte Komponente dient der Aufnahme von statistischen Werten für die Analyse der Algorithmen. Das Ergebnis der Simulation soll eine Aussage liefern, mit welchen Ausspielverzögerungen auf das Verhalten des Netzwerkes reagiert wird, und welche Paketverluste diese zur Folge haben. Zur Verdeutlichung des Empfangsverhaltens werden die zeitlichen Verläufe von Ankunft und Ausspielung der Pakete graphisch dargestellt.

4.3.4.1 Netzwerkverzögerung

Für jedes Paket wird die Netzwerkverzögerung $d_{i;j}$ protokolliert. Hiermit können die unterschiedlichen Übertragungen in bezug auf das Netzwerkverhalten verglichen werden, und das Reaktionsverhalten, auf z.B. impulsartige Erhöhungen der Netzwerkverzögerung $d_{i;j}$, des Empfängers analysiert werden.

4.3.4.2 Verlustrate

Die Verlustrate wird aus dem Verhältnis der verspäteten zu den angekommenen Paketen gebildet. Das Modul `DERTPDispLost` ermittelt drei verschiedene Verlustraten. Es wird der Verlust, bezogen auf die gesamte Übertragung und pro Sprechphase, angegeben. Die dritte Möglichkeit sieht vor, daß der Verlust pro Sekunde bestimmt wird.

4.3.4.3 Ausspielverzögerung

Das Modul `DERTPDispPlayoutDelay` speichert die Ausspielverzögerungen der einzelnen Sprechphasen in einer Datei ab. Es werden Start- und Endzeit mit dem Wert der Ausspielverzögerung D_i für jede Sprechphase protokolliert.

Zusätzlich wird für jedes Paket der Ankunftszeitpunkt $a_{i;j}$ und, wenn vorhanden, der ermittelte Ausspielzeitpunkt $p_{i;j}$ in einer weiteren Datei abgespeichert. Somit können die Verzögerungen der einzelnen Pakete innerhalb des Empfängers $p_{i;j} - a_{i;j}$ ermittelt werden.

Kapitel 5

Auswertung

5.1 Aufzeichnungen

Es wurden sechs Aufzeichnungen erstellt, wobei immer das gleiche PCM-codierte (μ-Quantisierung, Kap. 2.2) Audiosignal verwendet wurde. Übertragen wurde ein Interview mit dem Entwickler des *Archie*-Dienstes, daß in Dateien[1] durch *Internet Talk Radio* zu Verfügung steht. Alle Übertragungen hatten eine Länge von ca. 21 Minuten.

Zwei Übertragungen fanden auf dem lokalen Netzwerk der GMD-Fokus statt. Für eine Übertragung wurde eine ATM-Verbindung gewählt, die andere lief in der Gegenrichtung über das Ethernet. Diese beiden Übertragungen in entgegengesetzter Richtung gestatten den Nachweis, daß die Uhren der benutzten Rechner nicht synchron laufen (Siehe Abs. 5.2.1).

Zwei weitere Übertragungen fanden innerhalb Berlins statt. Gesendet wurde von der TU-Berlin zur GMD-Fokus zu zwei verschiedenen Zeitpunkten, einmal morgens und einmal mittags. Diese beiden Aufzeichnungen sollen der Untersuchnug des Verhaltens innerhalb eines stadtweiten Netzes (MAN), in dem mehrere Teilnetzwerke miteinander verbunden sind, dienen. Insbesondere kann der Einfluß von Gateways zwischen den Teilnetzwerken gezeigt werden.

Die letzten beiden Aufzeichnungen demonstrieren die Eigenschaften von Weitverkehrsverbindungen (WAN). Hierbei wurde das Audiosignal von Amherst, Massachusetts (USA) gesendet und in der GMD-Fokus (Berlin, Deutschland) empfangen und aufgezeichnet.

In der Tabelle 5.1 sind die erstellten Aufzeichnungen aufgelistet. Die verwendeten Rechnerarchitekturen und Betriebssyteme der Sender- und Empfangsstationen sind in Tabelle 5.2 festgehalten. Dies ist insofern wichtig, da die Messungen und Untersuchungen in der Applikationsebene der Protokollhierarchie stattfinden.

Mit dem Programm `traceroute` wurden Routen ermittelt. Um einen ungefähren Anhaltspunkt zu haben, über welche Rechner die MAN- und WAN-

[1] URL: `http://town.hall.org/Archives/radio/IMS/Geek/062693_geek_ITR.html`

Bezeichnung	Sender Empfänger	Beginn der Aufzeichnung	Netztyp
medusa-clyde-a	`medusa.fokus.gmd.de` `clyde-a.fokus.gmd.de`	25.08.95 (Fr) 10:03	LAN ATM
clyde-medusa	`clyde.fokus.gmd.de` `medusa.fokus.gmd.de`	28.08.95 (Mo) 11:57	LAN Ethernet
heute-medusa	`heute.cs.tu-berlin.de` `medusa.fokus.gmd.de`	28.08.95 (Mo) 12:31	MAN Internet
heute-omega	`heute.cs.tu-berlin.de` `omega.fokus.gmd.de`	28.08.95 (Mo) 08:16	MAN Internet
erlang-medusa-1	`erlang.cs.umass.edu` `medusa.fokus.gmd.de`	25.08.95 (Fr) 13:11	WAN Internet
erlang-medusa-2	`erlang.cs.umass.edu` `medusa.fokus.gmd.de`	30.08.95 (Mi) 21:58	WAN Internet

Tabelle 5.1: Liste der erstellten Aufzeichnungen

Netzadresse	Architektur	OS	Standort
`clyde.fokus.gmd.de`	SS20	SunOS 5.4	GMD Fokus, Berlin
`erlang.cs.umass.edu`	Sun SPARC IPX	SunOS 4.1.2	Amherst, MA (USA)
`heute.cs.tu-berlin.de`	Sun SPARC I	SunOS 4.1.3	TU-Berlin
`medusa.fokus.gmd.de`	Sun10/40	SunOS 5.4	GMD Fokus, Berlin
`omega.fokus.gmd.de`	Sun10/50	SunOS 5.3	GMD Fokus, Berlin

Tabelle 5.2: Architekturen und Betriebssyteme (OS) der Sende- und Empfangsstationen

Übertragungen liefen, wurde das Programm `traceroute` mit der Netzwerkadresse des verwendeten Senders aufgerufen. Die Ausgaben von `traceroute` sind in Abb. 5.1 dargestellt.

```
#traceroute erlang.cs.umass.edu
 1  gmdbgate (192.35.149.248)  2 ms  4 ms  2 ms
 2  gmdbigate.GMD.DE (188.1.132.142)  85 ms  55 ms  67 ms
 3  gmdisgate.gmd.de (192.54.35.68)  84 ms  78 ms  80 ms
 4  icm-bonn-1.gmd.de (192.76.246.17)  103 ms  124 ms  77 ms
 5  icm-dc-1-S2/6-512k.icp.net (192.157.65.209)  243 ms  174 ms  218 ms
 6  icm-mae-e-H1/0-T3.icp.net (198.67.131.9)  268 ms  319 ms  281 ms
 7  cpe2.Washington.mci.net (192.41.177.181)  239 ms  266 ms  272 ms
 8  border2-hssi4-0.Washington.mci.net (204.70.57.9)  284 ms  278 ms  216 ms
 9  core-fddi-1.Washington.mci.net (204.70.3.1)  274 ms  234 ms  273 ms
10  core-hssi-3.NewYork.mci.net (204.70.1.6)  290 ms  259 ms  316 ms
11  core-hssi-3.Boston.mci.net (204.70.1.2)  290 ms  295 ms  217 ms
12  border1-fddi-0.Boston.mci.net (204.70.2.34)  218 ms  177 ms  180 ms
13  nearnet.Boston.mci.net (204.70.20.6)  213 ms  224 ms  215 ms
14  mit3-gw.near.net (192.233.33.10)  212 ms  283 ms  239 ms
15  umass1-gw.near.net (199.94.201.66)  243 ms  244 ms  177 ms
16  lgrc-gw.gw.umass.edu (192.80.83.1)  216 ms  257 ms  229 ms
17  cs-gw.cs.umass.edu (128.119.44.1)  209 ms  210 ms  300 ms
18  erlang.cs.umass.edu (128.119.40.203)  228 ms  196 ms  189 ms

#traceroute heute.cs.tu-berlin.de
 1  gmdbgate (192.35.149.248)  2 ms  3 ms  2 ms
 2  cisco1.ZRZ.TU-Berlin.DE (188.1.132.30)  17 ms  15 ms  12 ms
 3  cisco1.gate.TU-Berlin.DE (130.149.1.1)  19 ms  23 ms  13 ms
 4  130.149.6.4 (130.149.6.4)  12 ms  17 ms  13 ms
 5  heute.cs.tu-berlin.de (130.149.19.19)  13 ms  22 ms  15 ms
```

Abbildung 5.1: Ermittlung der Routen mit Hilfe des Programmes `traceroute`. Das Programm wurde auf dem Rechner mit der IP-Adresse `daisy.fokus.gmd.de` innerhalb der GMD-Fokus (Berlin) aufgerufen.

In Tabelle 5.3 sind die Verluste, die innerhalb des Netzwerkes auftraten dargestellt. Die gemessenen Verluste, die aus dem zu späten Eintreffen von Paketen resultieren, werden im Abschnitt 5.4 behandelt. Bei allen Übertragungen konnten keine Pakete festgestellt werden, die im Netzwerk dupliziert worden sind. In den beiden Übertragungen zwischen USA und Deutschland kamen einige Pakete in falscher Reihenfolge beim Empfänger an.

5.2 Untersuchung der Netzwerkverzögerung

Die Netzwerkverzögerung $d_{i;j}$ setzt sich aus einem konstanten und einem variablen Anteil zusammen. Der konstante Anteil ist durch die Signalausbreitungsverzögerung $d^p_{i;j}$ (propagation delay) bestimmt. Der Variable Anteil $d^v_{i;j}$ enthält mehrere Komponenten, von denen einige, die mit den erstellten Aufzeichnungen nachweisbar sind, in den nächsten Abschnitten erläutert werden.

$$d_{i;j} = d^p_{i;j} + d^v_{i;j} \tag{5.1}$$

Aufzeichnung	Verlust [%]	Duplikate [%]	Falsche Reihenfolge [%]
medusa-clyde-a	0	0	0
clyde-medusa	0	0	0
heute-medusa	0.0651	0	0
heute-omega	0.0016	0	0
erlang-medusa-1	0.4094	0	0.0381
erlang-medusa-2	0.5934	0	0.0508

Tabelle 5.3: Paketverluste, Duplizierungen und Pakete in falscher Reihenfolge

Die Dichtefunktionen der Netzwerkverzögerungen, dargestellt in Abb. 5.3, zeigen, daß sich der variable Anteil $d^v_{i;j}$ innerhalb von lokalen Netzwerken (medusa-clyde-a und clyde-medusa) geringer bemerkbar macht, als in Übertragungen, die über die Grenze eines lokalen Netzwerkes hinausgehen.

5.2.1 Uhrendrift zwischen Sender und Empfänger

Wie in Abb. 5.2 zu sehen, ist in der Änderung der Verzögerung $d_{i;j}$ eine Komponente enthalten, die zu dem Zeitfortschritt proportional ist. Es ist unwahrscheinlich, daß sich die Verzögerung des Netzwerkes tatsächlich derartig ändert. Eher ist davon auszugehen, daß sich hier der asynchrone Lauf der Uhren von Sender und Empfänger bemerkbar macht, weil bei der Bestimmung der Netzwerkverzögerung eines Paketes sowohl die Zeitmessung des Senders, timestamp des RTP-Paketes, als auch die Messung des Empfängers (Ankunftszeitpunkt) eingehen. Der Vergleich der Übertragungen medusa-clyde-a und clyde-medusa bestätigt diese Annahme. Es resultieren betragmäßig gleiche Zeitabweichungen pro Sekunde mit umgekehrten Vorzeichen (siehe Tab. 5.4). Die Uhr des Rechners `medusa.fokus.gmd.de` läuft um 9 μs pro Sekunde schneller als die von `clyde.fokus.gmd.de`.

Aufzeichnung	Drift [μs/s]
medusa-clyde-a	-9
clyde-medusa	9
heute-medusa	-80
heute-omega	226
erlang-medusa-1	-71
erlang-medusa-2	-71

Tabelle 5.4: Uhrendrift zwischen Sender und Empfänger

Aus dieser Betrachtung folgt, daß nicht nur die Variabilität der Verzögerung, hervorgerufen durch das Netzwerk, eine Adaption der Ausspielverzögerung

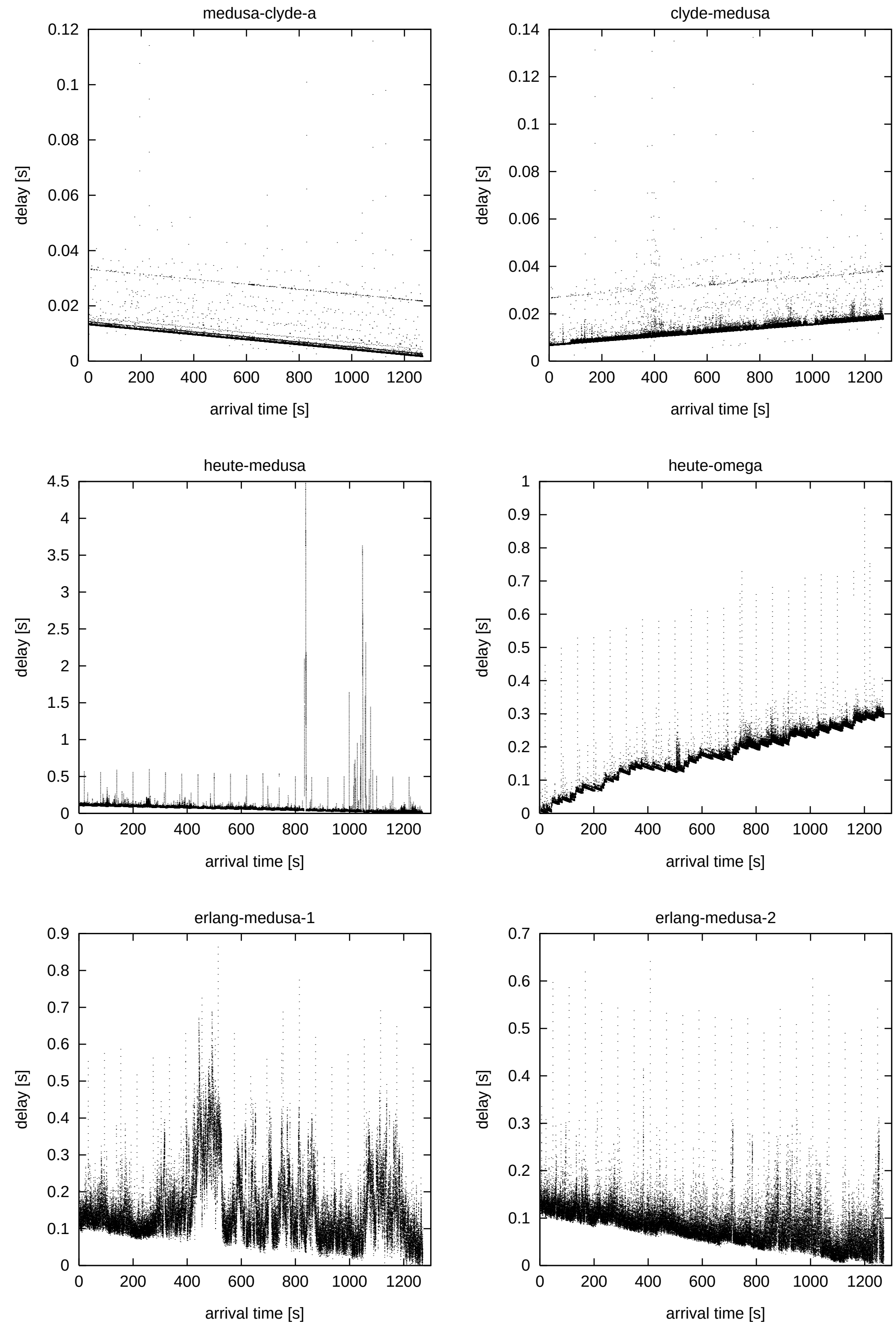

Abbildung 5.2: Darstellung der Netzwerkverzögerung $d_{i;j}$ in Abhängigkeit von der Ankunftszeit $a_{i;j}$. In den Grafiken ist die kleinste gemessene Verzögerung subtrahiert worden.

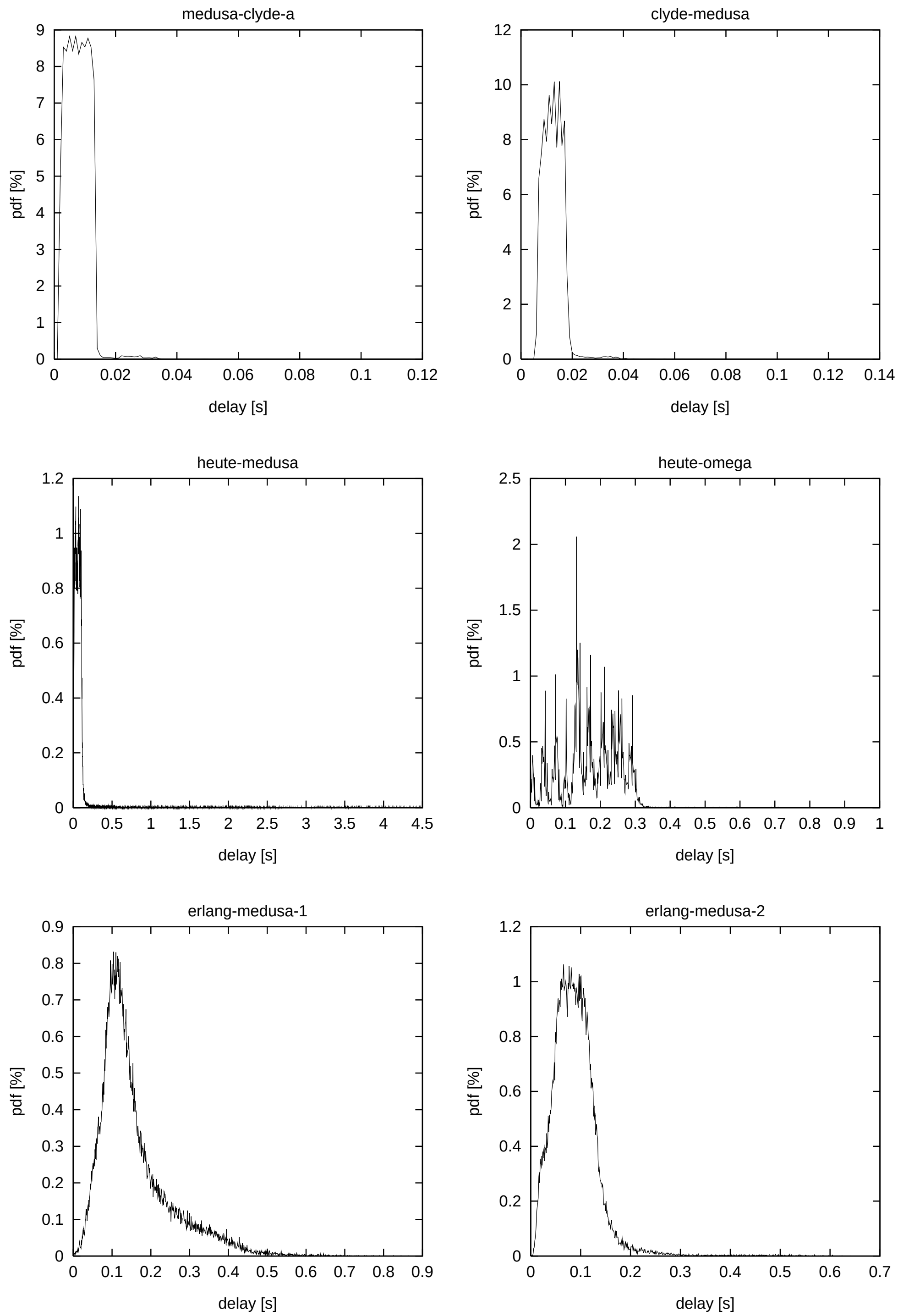

Abbildung 5.3: Verteilungen der Netzwerkverzögerung $d_{i;j}$. Die kleinste gemessene Verzögerung ist subtrahiert worden.

nötig macht. Bei längeren Übertragungen muß sich der Empfänger der Uhr des Senders anpassen.

Läuft die Uhr des Empfängers schneller als die des Senders, erfolgt die Anpassung durch Verlängerung der Sprechpausen. Im umgekehrten Fall werden Sprechpausen verkürzt. Dies wird durch die adaptive Ausspielverzögerung erreicht, weil die Zeitabweichung durch den Empfänger als Veränderung der Verzögerung $d_{i;j}$ interpretiert wird.

5.2.2 Verzögerungsspitzen

Bei Übertragungen, die über die Grenzen des lokalen Netzwerkes hinausgehen machen sich periodisch wiederkehrende Verzögerungsspitzen bemerkbar. Die dargestellten Verzögerungen der Übertragungen heute-medusa und heute-omega in Abb. 5.2 zeigen Verzögerungsspitzen mit einer Erhöhung von $d_{i;j}$ um ca. 0,5 s, die in einem Takt von 60 Sekunden auftreten.

Eine Erklärung für die periodischen Verzögerungsspitzen ist der Austausch der Wegewahlinformationen zwischen Gateways[2]. Innerhalb der TCP/IP-Protokollfamilie sieht das *Routing Information Protocol* (RIP) ([CS91], Kap. 17, S. 341 und [Hed88]) einen periodischen Austausch von derartigen Informationen vor. Die Zeitdauer einer Periode ist bei RIP 30 Sekunden lang, was bedeutet, daß dieses Protokoll nicht diese periodischen Verzögerungsspitzen hervorruft. RIP soll als Hinweis dienen, daß Protokollmechanismen mit periodischem Charakter existieren.

Es treten aber auch nicht periodische Verzögerungserhöhungen auf. Sowohl das zeitliche Auftreten, als auch der Betrag der Verzögerungsspitze haben einen zufälligen Charakter. Dies ist durch variierende Netzswerkbelastungen erklärbar. Die größte Verzögerungsspitze von ca. fünf Sekunden trat bei der Übertragung heute-medusa auf.

5.2.3 Verzögerung durch Mechanismen des Senders

Ein Sender, der mit Ruhe-Erkennungsmechanismen arbeitet, sendet während der Ruhephasen keine Pakete. Zum Beginn einer neuen Sprechphase werden die letzten n_c Pakete aus der davor liegenden Ruhephase gesendet. Hierfür existiert auch beim Sender ein Zwischenspeicher, der während einer Ruhephase die letzten n_c Pakete aufnimmt. Abhängig von n_c resultiert daraus eine Verzögerung $d^c_{i;j}$ des Signals, die durch den Sender hervorgerufen wird:

$$d^c_{i;j} = \frac{n_c}{f_p}$$

Der Sender, dessen Übertragungen hier ausgewertet werden, arbeitet mit $n_c = 1$; mit dem Beginn einer neuen Sprechphase wird das letzte Ruhepaket mitgesendet. Die Frequenz f_p, mit der der Sender die Pakete verschickt,

[2] Verbindungsrechner zwischen verschiedenen Teilnetzen

beträgt 50 Hz (siehe Gleichung 2.1). Daraus folgt eine Verzögerung $d^c_{i;j}$ von 20 ms.

Die Verzögerung $d^c_{i;j} = 20$ ms ist bei den Übertragungen medusa-clyde-a und clyde-medusa in Abb. 5.2 zu erkennen. In den Grafiken sind jeweils zwei geradlinige Verläufe zu sehen, die sich um $d^c_{i;j}$ unterscheiden.

5.2.4 Zusammenfassung der Anteile von $d^v_{i;j}$

Mit den vorhergehenden Abschnitten sind verschiedene Gründe für die Variabilität der Netzwerkverzögerung genannt worden, so daß Gleichung 5.1 wie folgt angegeben werden kann:

$$d(t) = d^p_{i;j} + d^d_{i;j} + d^r_{i;j} + d^c_{i;j} + d^s_{i;j}$$

Eine Aufschlüsselung nach den einzelnen Komponenten ist in Tabelle 5.5 enthalten.

$d^d_i i;j$	Asynchronität der Uhren zwischen Sender und Empfänger (Abs. 5.2.1).
$d^r_{i;j}$	Variabilität durch Verwaltungsaufwand innerhalb des Netzwerkes (siehe z.B. RIP, Abs. 5.2.2).
$d^c_{i;j}$	Verzögerungen durch Zwischenspeichern des Senders (Abs. 5.2.3).
$d^s_{i;j}$	Sonstige Verzögerungen, die aus stochastischen Prozessen innerhalb von paketvermittelnden Systemen resultieren (Warteschlangen).

Tabelle 5.5: Komponenten der Variabilität $d^v_{i;j}$

5.3 Abschätzung der Netzwerkverzögerung

Die vier Algorithmen unterscheiden sich im wesentlichen in dem Verfahren der Bestimmung von $\hat{d}_k$. Die Simulationsläufe ergaben, daß sich auch die Ergebnisse, die die Verfahren liefern, deutlich unterscheiden.

Jedem Algorithmus läßt sich ein charakteristischer Wert zuordnen:

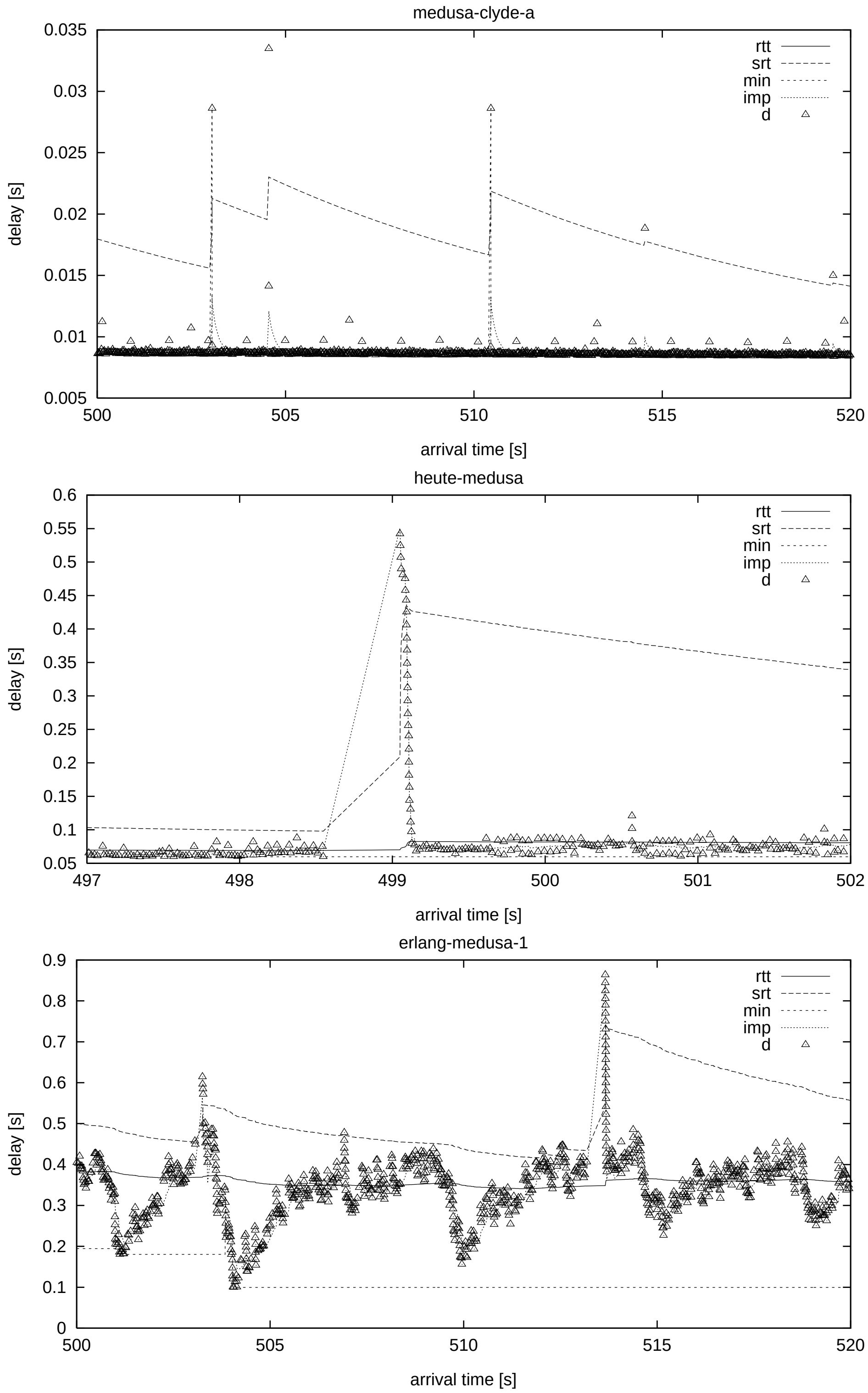

Abbildung 5.4: Darstellung der Werte $\hat{d}_k$ der einzelnen Algorithmen als Linien und der gemessenen Verzögerung $d_{i;j}$ durch Dreiecke über der Ankunftszeit. Bei den Abbildungen handelt es sich um Ausschnitte aus der Abb. 5.2 der jeweiligen Übertragung.

rtt	$\hat{d}_k \approx \bar{d}_i$	Mittelwert von $d_{i;j}$
srt	$\hat{d}_k \approx d_{max}$	obere Grenze von $d_{i;j}$
min	$\hat{d}_k = d_{min}$	untere Grenze von $d_{i;j}$
imp	$\hat{d}_k \approx d_{i;j}$	folgt dem Wert $d_{i;j}$ sehr schnell

Tabelle 5.6: Charakteristika der Algorithmen

In Abb. 5.4 ist das Verhalten der Algorithmen anhand von Ausschnitten der Übertragungen dargestellt.

In dem Ausschnitt der Übertragung medusa-clyde-a sind die Werte vom rtt-Algorithmus nicht zu erkennen, weil sie von den Meßwerten $d_{i;j}$ (Dreiecke) verdeckt werden. Die beiden Geraden im Ausschnitt der Übertragung heute-medusa bei Sekunde 498.5 entsprechen nicht den Verläufen der Algorithmen srt und imp. In den Darstellungen aus Abb.5.4 werden die einzelnen Werte von $\hat{d}_k$ der Algorithmen mit Linien verbunden. Normalerweise liegen die Werte zeitlich dicht nebeneinander (Zeitabstand ca. 0.02 s, siehe f_p aus Gleichung 2.1). Wenn aber große Lücken auftreten, ergeben sich Geraden, die nicht den tatsächlichen Werten entsprechen.

Der srt- und imp-Algorithmus reagieren auf die impulsartige Erhöhung ihrerseits mit der Erhöhung des Schätzwertes $\hat{d}_k$. Während der imp-Algorithmus die Verminderung der gemessenen Verzögerung verfolgt, vermindern sich die Werte von srt mit einer wesentlich höheren Zeitkonstante.

Für den Fall, daß das Verhalten der gemessenen Verzögerung $d_{i;j}$ einer Sprungfunktion (Gleichung 5.2) entspricht, können die Algorithmen srt und imp in eine Exponetialfunktionen umgeformt werden.

$$d_{i;j} \quad = \quad \begin{cases} a & \text{falls } t < T_s, \qquad\qquad T_s\text{: Zeitpunkt des Sprungs} \\ b & \text{falls } t \geq T_s \end{cases} \tag{5.2}$$

Dies dient der Beurteilung, wie schnell die Algorithmen dem Wert $d_{i;j}$ folgen. Es muß zwischen den Situationen der Erhöhung ($a < b$) und der Verminderung ($a > b$) der Verzögerung $d_{i;j}$ unterschieden werden, weil auch der srt-Algorithmus diese unterscheidet (siehe α und β aus Gleichung 3.4). Der srt-Algorithmus wird der Exponetialfunktion zum Zeitpunkt $t > T_s$ gleichgesetzt ($a > b$), wobei die erste Bestimmung von d_k nach T_s gemeint ist. Dann entspricht d_{k-1} dem Wert a:

$$\begin{aligned} \alpha\, a + (1-\alpha) b &= b + (a-b)\, e^{-T_\alpha t} \\ \alpha(a-b) &= (a-b)\, e^{-T_\alpha t} \\ \alpha &= e^{-T_\alpha t} \\ T_\alpha &= -\frac{\ln \alpha}{t} \end{aligned}$$

Für den zweiten Fall, wenn sich $d_{i;j}$ erhöht ($a < b$), ergibt sich analog die Zeitkonstante T_β:

$$T_\beta = -\frac{\ln \beta}{t}$$

Mit $\alpha = 0.998002$, $\beta = 0.75$ und $t = 0.02\,\mathrm{s}$ folgt:

$$\begin{aligned} T_\alpha &\approx 0.1\,\mathrm{s}^{-1} \\ T_\beta &\approx 14.4\,\mathrm{s}^{-1} \end{aligned}$$

Diese Werte verdeutlichen, daß unter Verwendung des srt-Algorithmusses, der Wert $\hat{d}_k$ sehr schnell (T_β) auf impulsartige Erhöhungen der gemessenen Verzögerung $d_{i;j}$ reagiert und sich nur sehr langsam (T_α) niedrigeren Werten von $d_{i;j}$ nähert. Treten bei einer Übertragung positive Verzögerungsspitzen auf, liegt der Wert $\hat{d}_k$ über den gemessenen Werten von $d_{i;j}$.

Der imp-Algorithmus arbeitet im Normal-Modus immer mit der Zeitkonstante T_β, unabhängig davon, ob $d_{i;j}$ kleiner oder größer wird. Der Verlauf von $\hat{d}_k$ ist in Abb. 5.4 nur schwer zu erkennen, weil dieser der gemessenen Größe $d_{i;j}$ fast entspricht. Im Impuls-Modus (Übertragung heute-medusa) ist die direkte Kopplung der Änderung von $\hat{d}_k$ an den Verlauf von $d_{i;j}$ zu erkennen.

Der rtt-Algorithmus arbeitet mit der Zeitkonstante T_α. In der Übertragung erlang-medusa-1 ist das langsame Antwortverhalten zu sehen.

5.4 Verluste

Durch die unterschiedlichen Ausspielverzögerungen, die die vier Algorithmen bestimmen, ergeben sich auch verschiedene Verlustraten. In Tab. 5.7 sind die resultierenden Verlustraten aufgelistet. Für jede Übertragung sind die Verlustraten mit einer Rangfolge versehen; der Algorithmus mit dem geringsten Wert hat den Rang (1).

Die Ergebnisse des srt-Algorithmusses weisen die geringsten Verluste auf; sie liegen, bis auf den Sprechphasenverlust der Übertragung heute-medusa, unter einem Prozent, in den meisten Fällen unter einem Promille. Es ist keine Abhängigkeit des Verlustes von der Variabililität der Netzwerkverzögerung d_i erkennbar. Die Verluste sind bei der Übertragung heute-medusa am größten. Die Werte der anderen Übertragungen liegen alle unter einem Promille.

Die größten Verluste weist der imp-Algorithmus auf. Mit Zunahme der Variabilität von d_i steigen die Verluste auf über drei Prozent. Im lokalen Bereich liefert er identische Ergebnisse wie die Algorithmen rtt und min.

Die beiden Algorithmen rtt und min liegen mit ihren Ergebnissen im Mittelfeld. Interessant ist, daß der min-Algorithmus innerhalb der MAN-Übertragungen (heute-medusa und heute-omega) größere Verluste aufweist, als der

Aufzeichnung	Alg.	Gesamtverlust in [%]	durchschnittlicher Verlust in [%] pro Sekunde	durchschnittlicher Verlust in [%] pro Sprechphase
medusa-clyde-a	rtt	(2) 0.0983768	(2) 0.097999	(2) 0.078176
	srt	(1) 0.0190407	(1) 0.018942	(1) 0.017439
	min	(2) 0.0983768	(2) 0.097999	(2) 0.078176
	imp	(2) 0.0983768	(2) 0.097999	(2) 0.078176
clyde-medusa	rtt	(2) 0.149152	(2) 0.148548	(2) 0.130112
	srt	(1) 0.0190407	(1) 0.018942	(1) 0.007814
	min	(2) 0.149152	(2) 0.148548	(2) 0.130112
	imp	(2) 0.149152	(2) 0.148548	(2) 0.130112
heute-medusa	rtt	(2) 1.3226	(2) 1.519354	(1) 1.039241
	srt	(1) 0.574767	(1) 0.750102	(3) 1.222720
	min	(3) 1.56394	(3) 1.659415	(2) 1.217209
	imp	(4) 1.91007	(4) 1.948653	(4) 1.474353
heute-omega	rtt	(2) 1.74542	(2) 1.720966	(3) 1.349545
	srt	(1) 0.0698169	(1) 0.068069	(1) 0.042645
	min	(3) 1.8803	(3) 1.853560	(2) 1.291724
	imp	(4) 2.18654	(4) 2.169141	(4) 1.673406
erlang-medusa-1	rtt	(3) 1.45941	(3) 1.469313	(3) 1.126957
	srt	(1) 0.012746	(1) 0.012628	(1) 0.014696
	min	(2) 0.693061	(2) 0.689872	(2) 0.510528
	imp	(4) 3.08611	(4) 3.126913	(4) 2.483837
erlang-medusa-2	rtt	(3) 2.03036	(3) 2.018293	(3) 1.413599
	srt	(1) 0.049482	(1) 0.047944	(1) 0.049335
	min	(2) 1.58821	(2) 1.572135	(2) 1.093415
	imp	(4) 2.71992	(4) 2.720699	(4) 2.239026

Tabelle 5.7: Auflistung der Verlustraten durch Verspätungsverlust. Angegeben sind die Verluste der gesamten Übertragung und die Durchschnittswerte der Verluste pro Sekunde und pro Sprechphase.

srt	geringste Verluste
min	
rtt	
imp	größte Verluste

Tabelle 5.8: Reihenfolge der Algorithmen nach Verlusten

rtt-Algorithmus. Bei den Übertragungen erlang-medusa-1 und erlang-medusa-2 (WAN) kehrt sich das Ergebnis um. Mit Blick auf die Ergebnisse innerhalb der WAN-Übertragungen, fällt der Gesamtvergleich zu Gunsten des min-Algorithmus aus.

Zusammenfassend ist das Ergebnis, nach Verlusten sortiert, in Tabelle 5.8 dargestellt.

Die Diagramme in den Abbildungen 5.5 und 5.6 zeigen die Verteilungsfunktionen der Verluste pro Sekunde und pro Sprechphase.

5.5 Verzögerungsverhalten des Empfängers

In den Abbildungen 5.7 und 5.8 sind die Ausspielverzögerungen D_i der einzelnen Algorithmen dargestellt. Horizontale Anteile der Linien entsprechen der Dauer der jeweiligen Sprechphase. Die Höhe gibt den Wert der Verzögerung D_i an, mit dem die Sprechphasen ausgespielt worden sind. Zusätzlich sind die *optimalen* Werte der Ausspielverzögerung dargestellt. Mit diesen Ausspielverzögerungen würden gerade keine Verluste auftreten. Wenn die Ausspielverzögerung eines Algorithmuses größer wird als die optimale Verzögerung, kommt es zu Verlusten durch zu späte Pakete.

In Abb. 5.7 ist das Verhalten zu beobachten, wie der Empfänger auf eine sehr große Erhöhung der Netzwerkverzögerung (heute-medusa ungefähr bei Sekunde 820) reagiert. Nach dem Auftreten des Impulses steigt die Ausspielverzögerung D_i aller Algorithmen. Anschließend machen sich die Grenzen der Verringerung der Ausspielverzögerung bemerkbar (siehe Gleichung 3.7 in Kap. 3.2), wenn die folgenden Ruhephasen kurz sind und die Netzwerkverzögerung wieder gesunken ist.

Die Abbildungen 5.9 und 5.10 zeigen die Verteilungsfunktionen der Verzögerung, die durch den Empfänger hervorgerufen wird $(p_{i;j} - a_{i;j})$.

Der srt-Algorithmus erzeugt die größten Verzögerungswerte, die die Grenzen aus Tab. 1.1 in Kap. 1 meist übersteigen. Bis auf die Benutzung im lokalen Bereich (medusa-clyde-a und clyde-medusa) ist zu erwarten, daß der srt-Algorithmus die Interaktivität beeinträchtigt.

Die restlichen drei Algorithmen (rtt, min und imp) weisen Verzögerungswerte auf, die sich den Forderungen aus Kap. 1 nähern, wobei nicht gezeigt werden

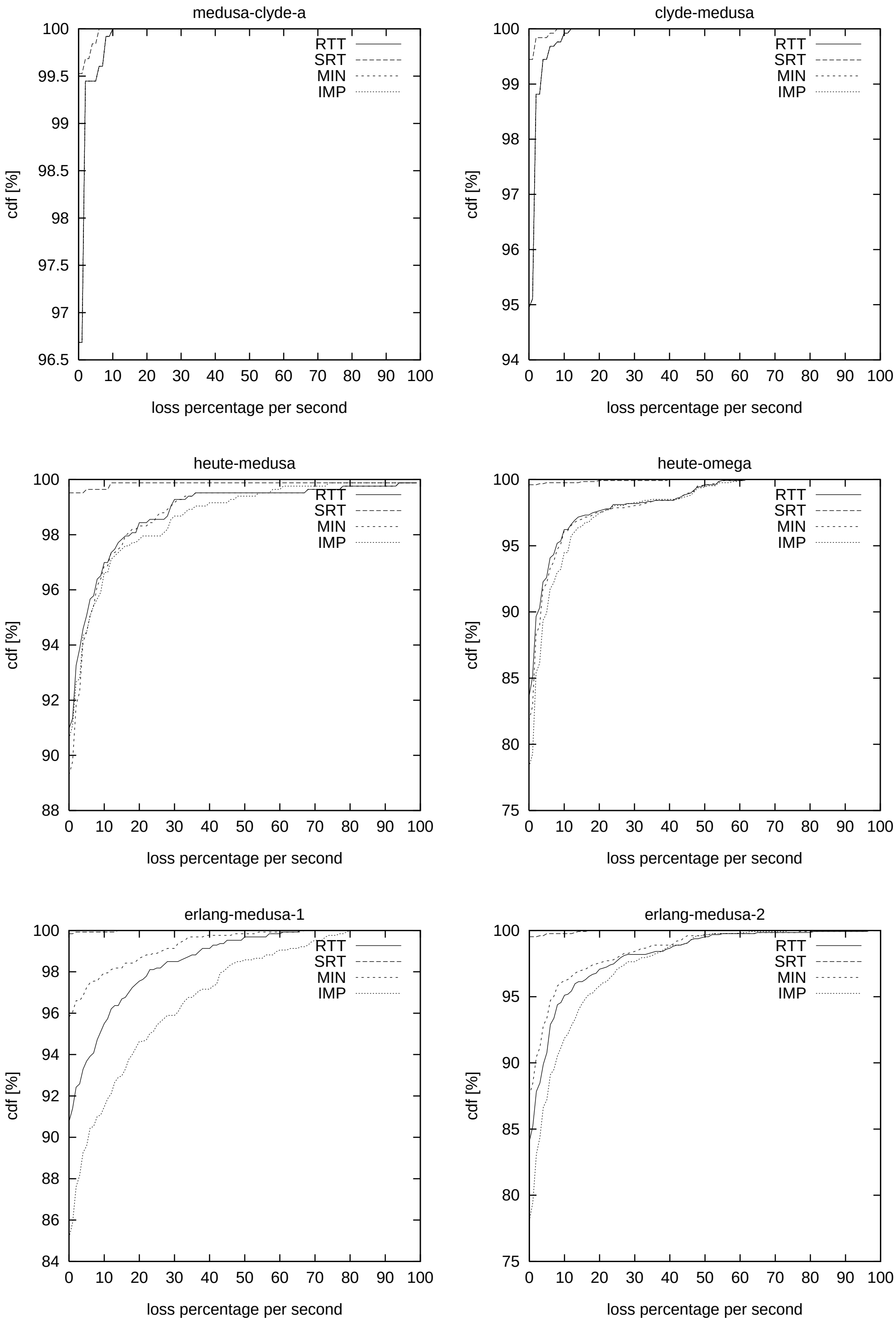

Abbildung 5.5: Verteilungsfunktionen der Verspätungsverluste pro Sekunde

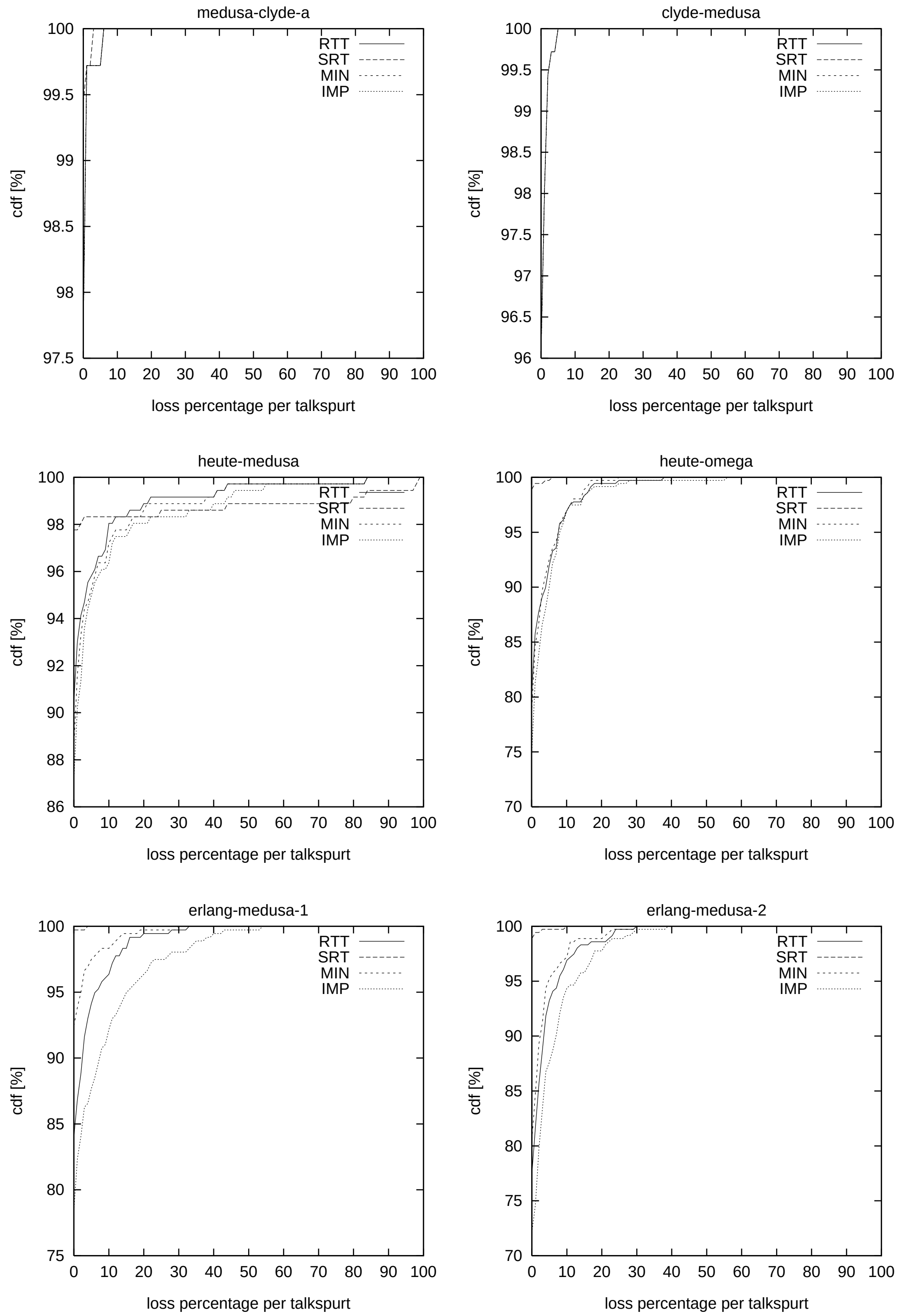

Abbildung 5.6: Verteilungsfunktionen der Verspätungsverluste pro Sprechphase

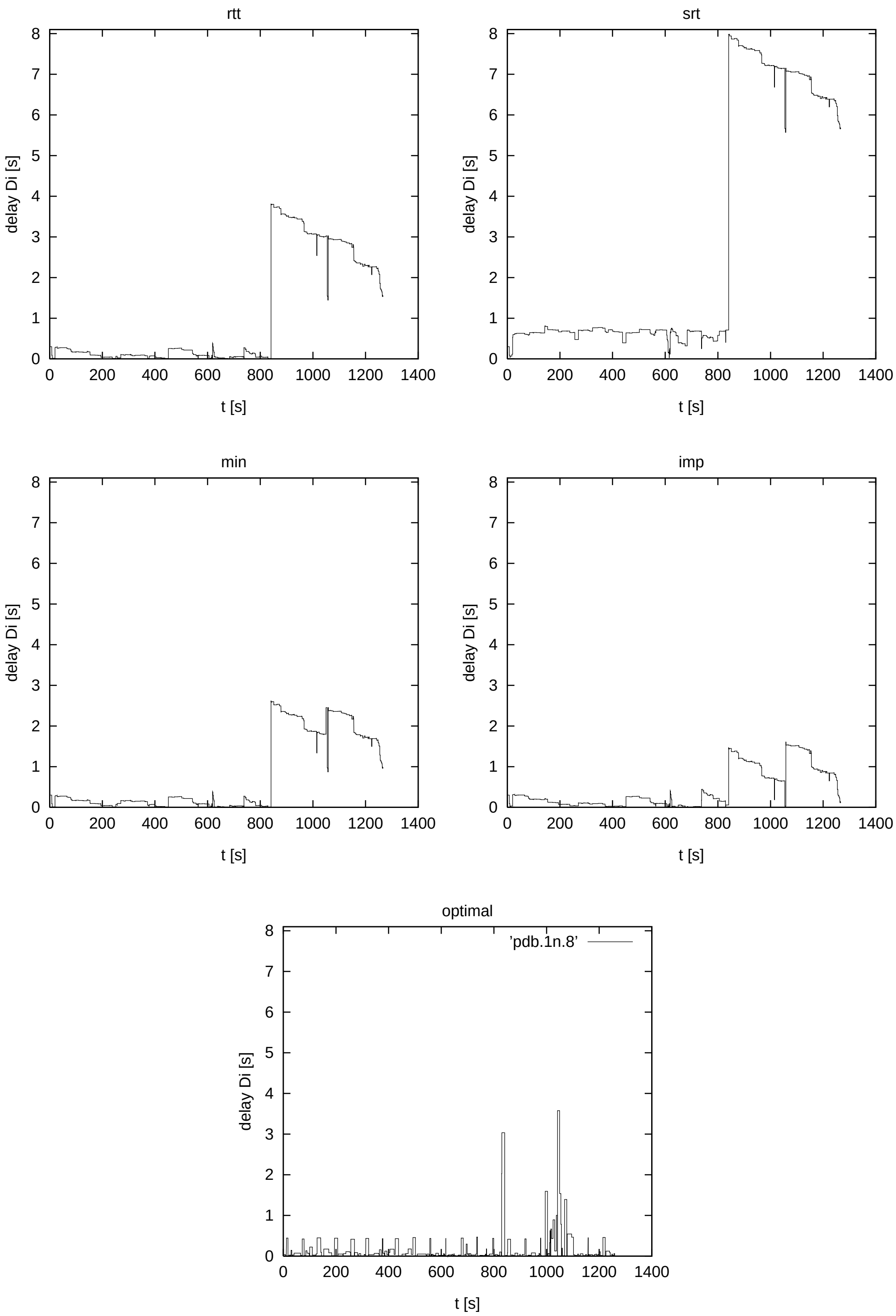

Abbildung 5.7: Ausspielverzögerungen D_i der Übertragung heute-medusa

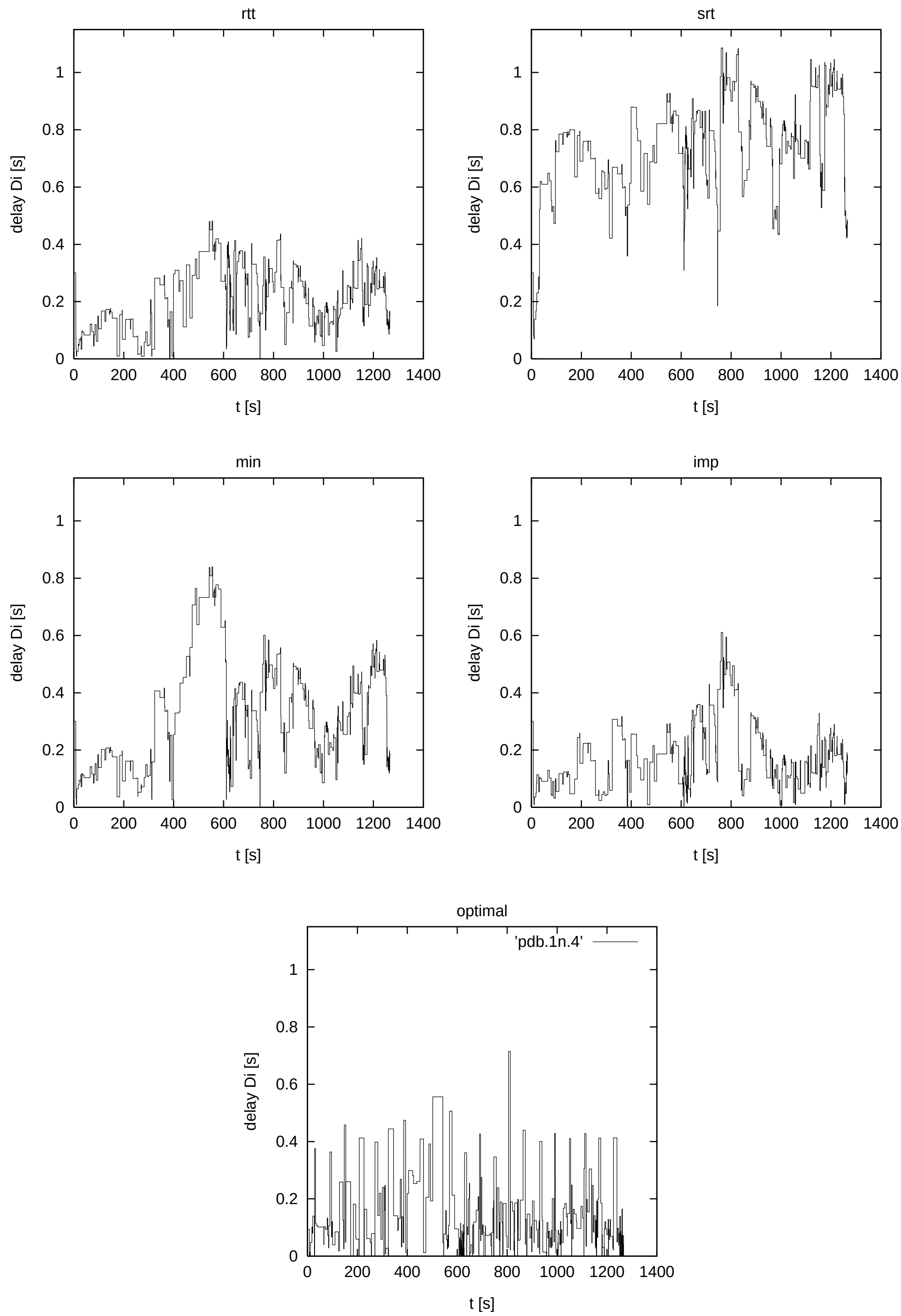

Abbildung 5.8: Ausspielverzögerungen D_i der Übertragung erlang-medusa-1

kann, daß die Verzögerungen in einem akzeptablen Rahmen liegen, weil die absolute Gesamtverzögerung unbekannt ist (siehe Kap. 3).

5.6 Adaption innerhalb von Sprechphasen

Wenn der Empfänger bei Paketen, die zu spät ankommen, indem er diese Pakete sofort ausspielt (siehe Kap. 3.3), kann erreicht werden, daß keine Information verloren geht, vorausgesetzt, der Empfänger arbeitet mit einem genügend großen Empfangsspeicher zum Zwischenpuffern der empfangenen Pakete.

Trifft ein zu spätes Paket ein, wird es sofort ausgespielt. Dadurch wird die Ausspielverzögerung indirekt erhöht. Dies macht sich vor allem bemerkbar, wenn Pakete mit sehr großen Verzögerungen eintreffen. In der Übertragung *heute-medusa* treten zwei Verzögerungsspitzen mit einer Erhöhung von vier bis fünf Sekunden auf. Zusätzlich sind periodische Spitzen mit 0.5 s Verzögerung vorhanden, die alle 60 s in Erscheinung treten (Kap. 5.2.2).

Wenn der Empfänger zum Beginn einer Spitze das Paket mit dieser hohen Verzögerung ausspielt, bestimmen sich die Ausspielzeitpunkte der folgenden Pakete entsprechend danach. Also erhöht sich für alle Pakete die Verzögerung um diesen Spitzenwert. Erst nach dem Ende der Sprechphase, in der die Spitze auftritt, kann die Ausspielverzögerung gemindert werden, wobei dies auch nur in den Grenzen geschehen kann, wie sie in Kap. 3.2 beschrieben sind. Sind die folgenden Ruhephasen um einiges kürzer als der Spitzenwert, so kann erst nach mehreren Sprechphasen die meist geringere Ausspielverzögerung, bestimmt durch die adaptiven Algorithmen, verwendet werden.

Treten also große Verzögerungsspitzen innerhalb einer Übertragung auf, so resultiert daraus eine allgemeine Erhöhung der Ausspielverzögerung. Dies ist auch im Vergleich der Abbildungen 5.9 und 5.10 erkennbar, in der die Verteilungen der Verzögerungen der Pakete beim Empfänger dargestellt sind. In Abb. 5.10 arbeitet der Empfänger mit dem hier beschriebenen Verfahren. Abb. 5.9 zeigt die Ergebnisse, wenn der Empfänger Pakete, die zu spät ankommen, ignoriert. Alle Übertragungen weisen in Abb. 5.10 größere Verzögerungswerte auf als in Abb. 5.9. Außer im lokalen Bereich, werden die meisten Pakete mit mehr als 200 Millisekunden verzögert (Abb. 5.10), während sie ohne dieses Verfahren mit deutlich geringeren Werten verzögert werden. In der Übertragung erlang-medusa-2 werden 50 Prozent der Pakete mit unter 118 Millisekunden verzögert. Mit der Ausspielung von zu spät angekommenen Paketen liegt die 50-Prozentmarke bei 409 Millsekunden, was den Anforderungen aus Tabelle 1.1 nicht mehr entsprechen kann.

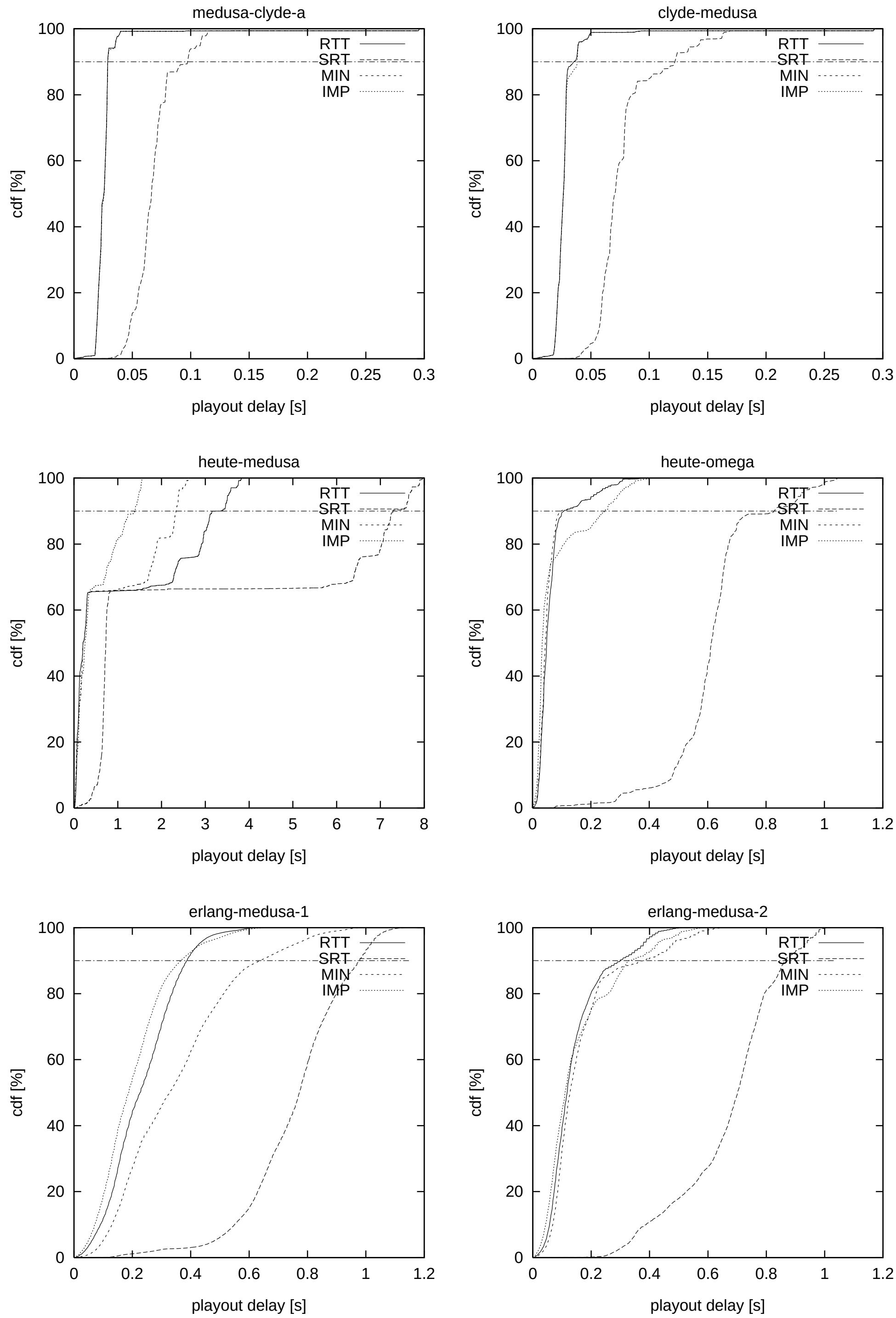

Abbildung 5.9: Verteilungsfunktionen der Ausspielverzögerungen von Paketen $(p_{i;j} - a_{i;j})$. Die 90%-Linie dient dem Vergleich der Grafiken.

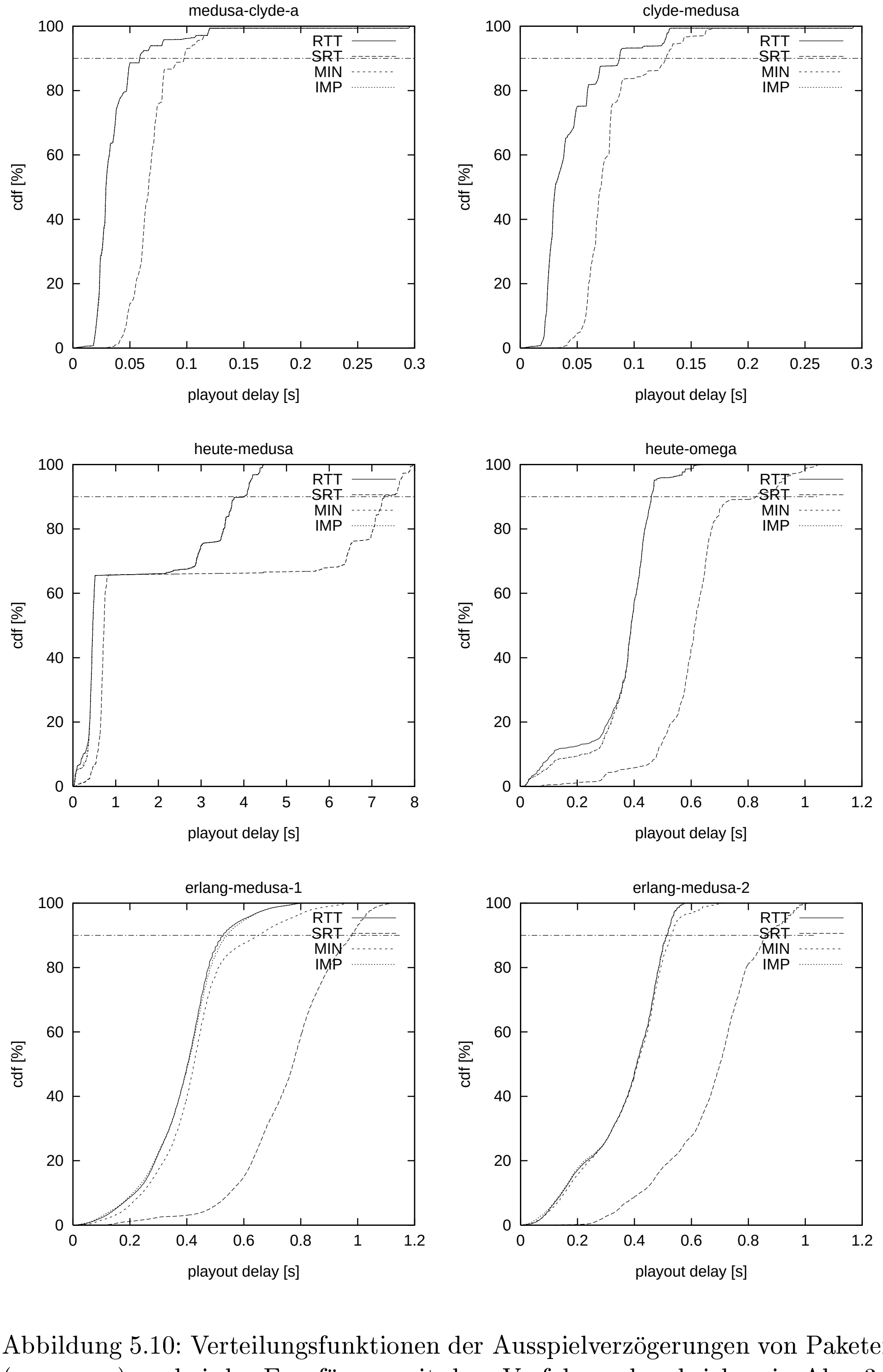

Abbildung 5.10: Verteilungsfunktionen der Ausspielverzögerungen von Paketen $(p_{i;j} - a_{i;j})$, wobei der Empfänger mit dem Verfahren, beschrieben in Abs. 3.3 arbeitet. Die 90%-Linie dient dem Vergleich der Grafiken.

Anhang A

Liste der Formelzeichen

D_i	Ausspielverzögerung einer Sprechphase
G	Generationszeit eines Paketes durch den Sender
N_i	Anzahl der Pakete einer Sprechphase
R	Bitrate eines binär-codierten Audiosignals
R_p	Bitrate für Pakete eines binär-codierten Audiosignals
$a_{i;j}$	Ankunftszeit eines Paketes beim Empfänger
c	Anzahl der Kanäle einer Audiocodierung
$d_{i;j}$	Netzwerkverzögerung eines Paketes
$\hat{d}_k$	Folge von Abschätzungen der Netzwerkverzögerung
$d^c_{i;j}$	Verzögerung, hervorgerufen durch Caching-Mechanismen des Senders
$d^d_{i;j}$	Verzögerung, hervorgerufen durch asynchronen Lauf der Uhren des Senders und des Empfängers
$d^r_{i;j}$	Variable Verzögerung durch Routingprotokolle (Wegewahl)
$d^s_{i;j}$	Variable Verzögerung durch stochastische Prozesse innerhalb eines paketvermittelnden Netzwerkes (Warteschlangen)
$d^v_{i;j}$	Variabler Anteil der Verzögerung $d_{i;j}$
f_p	Frequenz mit der Pakete in einer Continuous Media-Übertragung erzeugt werden
f_s	Abtastfrequenz einer Audiocodierung (sampling rate)
h	Anzahl der Oktetts eines Paketkopfes
l	Anzahl der Bits pro digitalem Anbtastwert
n	Anzahl der Abtastwerte, die mit einem Paket versendet werden.

n_c Anzahl der Pakete, die der Sender innerhalb einer Ruhephase zwischenspeichert.

$p_{i;j}$ Ausspielzeitpunkt eines Paketes

r_i Dauer der Pause hinter einer Sprechphase

$s_{i;j}$ Sendezeitpunkt eines Paketes

$t_{i;j}$ `timestamp` eines Paketes (Kap. 2.6.1)

$\hat{v}_k$ Folge von Abschätzungen der Variabilität der Netzwerkverzögerung

Anhang B

Quelltexte

B.1 RTPPacket.h

```
// RTPPacket.h
//
#ifndef _RTPPacket_h_
#define _RTPPacket_h_
#include "Message.h"
#include "types.h"
#include "rtp.h"

//
// Konstanten fuer das Statusflag
//
#define RTP_OK              1
#define RTP_LATE            2
#define RTP_EARLY           3
#define RTP_LATE_TALKSPURT  4
#define RTP_UNDEFINED       0

class RTPPacket : public Message {
protected:
  char buf[1024];
  rtp_hdr_t *rtp_h;     // RTP-Kopf
  char *data;           // Paketnutzdaten
  u_int32 length;       // Laenge der Nutzdaten

  //
  // Simulationsspezifisch
  //
  int status;           // Statusflag
  double play_time;     // geplante Ausspielzeit
  double arrival_time;  // Ankunftszeit
  double delay;         // Netzwerkverzoegerung
public:
  const char* dataType() const { return "RTPPacket";}
  ISA_INLINE(RTPPacket,Message);
```

```
//
// Konstruktor RTPPacket::RTPPacket(char *buf, u_int32 len)
//
// Mit dem Konstruktor wird ein RTPPacket-Objekt initialisiert.
// 'buf' zeigt auf die Daten eines empfangenen Paketes (Kopf + Daten).
// 'len' gibt die Groesse des Paketes an (Kopf + Daten).
//
RTPPacket(const char *_buf, u_int32 len) {
  memcpy(buf, _buf, len);
  rtp_h = (rtp_hdr_t *)buf;
  data = buf + sizeof(rtp_hdr_t);
  length = len - sizeof(rtp_hdr_t);
  status = RTP_UNDEFINED;
}

//
// Copy-Konstruktor RTPPacket::RTPPacket(const RTPPacket& src)
//
// Aus einem vorhandenen RTPPacket-Objekt wird ein neues RTPPacket-Objekt
// initialisiert.
//
RTPPacket(const RTPPacket& src) {
  memcpy(buf, src.buf, sizeof(rtp_hdr_t) + src.length);
  rtp_h = (rtp_hdr_t *)buf;
  data = buf + sizeof(rtp_hdr_t);
  length = src.length;
  status = src.status;
}


//
// Umwandlung des Paketinhaltes in Host-Darstellung (Byte-Order)
//
void Net2Host(void) {
  *((u_int16*)rtp_h) = ntohs(*((u_int16*)rtp_h));
  rtp_h->seq = ntohs(rtp_h->seq);
  rtp_h->ts = ntohl(rtp_h->ts);
  rtp_h->ssrc = ntohl(rtp_h->ssrc);
}


//
// Umwandlung des Paketinhaltes in Netzwerk-Darstellung (Byte-Order)
//
void Host2Net(void) {
  *((u_int16*)rtp_h) = htons(*((u_int16*)rtp_h));
  rtp_h->seq = htons(rtp_h->seq);
  rtp_h->ts = htonl(rtp_h->ts);
  rtp_h->ssrc = htonl(rtp_h->ssrc);
}
```

```
//
// Elementfunktionen zur Abfrage von RTP-Werten
//
unsigned int GetType(void) const {
  return ((*((u_int16 *)rtp_h) >> 14) & 3);
}
unsigned int GetP(void) const {
  return ((*((u_int16 *)rtp_h) >> 13) & 1);
}
unsigned int GetX(void) const {
  return ((*((u_int16*)rtp_h) >> 12) & 1);
}
unsigned int GetCc(void) const {
  return ((*((u_int16*)rtp_h) >> 8) & 15);
}
unsigned int GetM(void) const {
  return ((*((u_int16*)rtp_h) >> 7) & 1);
}
unsigned int GetPt(void) const {
  return (*((u_int16*)rtp_h) & 127);
}

u_int16 GetSeq(void)        const { return(rtp_h->seq); }
u_int32 GetTs(void)         const { return(rtp_h->ts); }
u_int32 GetSsrc(void)       const { return(rtp_h->ssrc); }

char *GetBuf(void)          const { return (data); }
u_int32 GetLength(void)     const { return(length); }

void SetPlayTime(double t)        { play_time = t; }
double GetPlayTime(void)    const { return (play_time); }
void SetStatus(int s)             { status = s; }
int GetStatus(void)         const { return(status); }
void SetArrivalTime(double t)     { arrival_time = t; }
double GetArrivalTime(void) const { return (arrival_time); }
void SetNetDelay(double t)        { delay = t; };
double GetNetDelay(void)    const { return (delay); }

//
// virtuelle Funktion 'print' der Klasse 'Message',
// dient der Visualisierung des Inhaltes von Message-Objekten
//
StringList print() const {
  StringList out = "RTP> ver:";
  out += this->GetType();
  out += "  p:";        out += (int)this->GetP();
  out += "  x:";        out += this->GetX();
  out += "  cc";        out += this->GetCc();
  out += "  m:";        out += this->GetM();
  out += "  pt:";       out += this->GetPt();
  out += "  seq:";      out += this->GetSeq();
  out += "  ts:";       out += this->GetTs();
```

```
    out += "  ssrc:";     out += this->GetSsrc();
    out += " <> pl.t.: "; out += this->GetPlayTime();
    out += " stat: ";     out += this->GetStatus();
    return out;
  }
};
#endif //_RTPPacket_h_
// That's the last line of 'RTPPacket.h'
```

B.2 DEReadRTPTrace.pl

```
defstar
{
  name { ReadRTPTrace }
  derivedFrom { RepeatStar }
  domain { DE }
  descriptor {
Reads RTP & RTCP packets of a file generated with 'rtpdump -w'.
Produces RTP-Ptolemy-Messages.
  }
  hinclude { <sys/time.h>, <netinet/in.h>, "RTPPacket.h", "rtpdump.h" }
  output {
    name { rtp }
    type { message }
  }

  defstate {
    name { fileName }
    type { string }
    default { "/dev/null" }
    descriptor { Input file. }
  }

  protected {
    int input;
    double play_time;
    RTPPacket *rtp_packet;
  }

  constructor {
    delayType = TRUE;
  }

  begin {
    // open input file
    char c;
    RD_hdr_t hdr;
    if (input) {
      close(input);
      input = 0;
    }
```

```
    if (-1 == (input = open(expandPathName(fileName), O_RDONLY)))
      Error::abortRun(*this, "can't open file ", fileName);

    do
      read(input, &c, 1);
    while ((c != '\n') && c != '\0');


    read(input, &hdr, sizeof(RD_hdr_t));
    feedbackOut->put(completionTime) << 0.0;
  }

  method {
    name { ReadRTPPacket }
    type { int }
    code {
      char buf[2048];
      RD_packet_t hdr;
      do {
        if (sizeof(RD_packet_t) != read(input, &hdr, sizeof(RD_packet_t))) {
          return (-1);
        }
        hdr.length = ntohs(hdr.length) - sizeof(RD_packet_t);
        if (hdr.length != read(input, &buf, hdr.length)) {
          return (-1);
        }
      } while (ntohs(hdr.type));
      play_time = ntohl(hdr.offset) / 1000000.0;
      rtp_packet = new RTPPacket(buf, hdr.length);
      return (0);
    }
  }

  go {
    if (-1 == ReadRTPPacket())
      return;

    rtp_packet->Net2Host();

    Envelope pkt(*rtp_packet);
    rtp.put(play_time) << pkt;
    refireAtTime(play_time);
  }
}
```

B.3 DERTPScheduler.pl

```
defstar
{
  name { RTPScheduler }
  domain { DE }
```

```
  //
  // Parameter von DERTPScheduler
  //
  defstate {
    name { algorithm }
    type { int }
    default { "Algorithm" }
    desc {
    Algorithmus: 1 = rtt, 2 = srt, 3 = min, 4 = imp
    }
  }
  defstate {
    name { adaption_on_late }
    type { int }
    default { "LateMakesTalkspurt" }
    desc {
      0: keine Adaption bei spaeten Paketen, !0: Adpation bei sp. Pak.
      }
  }
  defstate {
    name { initial_delay }
    type { float }
    default { "0.3" }
    desc {
      Initiale Ausspielverzgerung
      }
  }
  defstate {
    name { delay_0 }
    type { float }
    default { "0.3" }
    desc {
Verzoegerung des ersten Paketes
    }
  }

  defstate {
    name  {fa }
    type { float }
    default { "8000.0" }
    desc {
Abtastfrequenz
    }
  }

  defstate {
    name { dir }
    type { string }
    default { "{OutputDir}" }
  }

  defstate {
```

```
    name { trace }
    type { string }
    default { "{Tracename}" }
  }

  //
  // Die Funktion 'NewTalkspurt' berechnet den Ausspielzeitpunkt
  // der neuen Sprechphase. Wenn ein Ausspielzeitpunkt geplant
  // werden konnte wird eine '0' zuruecgegeben. Konnte keine
  // geeignete Zeit gefunden werden wird eine '1' zurueckgegeben.
  //
  method {
    name { NewTalkspurt }
    type { int }
    code {
      // Bestimmung des Ausspielzeitpunktes
      play_time = arrivalTime + (ed + (4 * ev) - delay);

      // Ende der Ausspielung
      end_time = (double)((double)(sample_cnt - 1) / (double)fa);
      end_time += play_time;


      //
      // Wenn durch 'play_time' bzw. 'end_time' die Ausspielung zu weit in
      // der Zukunft liegt (end_queue), wird 'play_time' verringert,
      // so da"s die Ausspielung gerade noch m"oglich ist.
      //
      if (end_time > double(end_queue%0)) {
        play_time = - (double)((double)(sample_cnt - 1) / (double)fa);
        play_time += double(end_queue%0);
        if (play_time < (last_inserted_time +
           (double)((double)1/(double)fa))) {
          return (1);
        }
      }

      //
      // Ueberlappung mit der letzten Sprechphase ??
      //
      if (play_time < (last_inserted_time + (double)((double)1/(double)fa))) {
        play_time = last_inserted_time + (double)((double)1/(double)fa);
      }

      //
      // Ausspielzeitpunkt schon vorbei (start_queue) ??
      //
      if (play_time < double(start_queue%0)) {
        play_time = double(start_queue%0);
      }
      end_time = (double)((double)(sample_cnt - 1) / (double)fa);
      end_time += play_time;
      if (end_time > double(end_queue%0)) {
```

```
      return (1);
    }

    ts_time = play_time;
    ts_stamp = begin_stamp;

    if (3 == algorithm)
      ed = HUGE;
    return (0);
  }
}

//
// Die Funktion 'Calc_estimates' fuehrt fuer jedes ankommende
// Paket die Berechnung der Schaetzwerte von \hat{d} und \hat{v} durch.
// Abhaengig von der Variable 'algorithm' wird der Algorithmus
// gewaehlt.
//
method {
  name { Calc_estimates }
  code {
    switch (algorithm) {
    case 1:
      ed = (alpha * ed) + ((1 - alpha) * delay);
      break;
    case 2:
      if (delay > ed)
        ed = (beta * ed) + ((1 - beta) * delay);
      else
        ed = (alpha * ed) + ((1 - alpha) * delay);
      break;
    case 3:
      if (delay < ed)
        ed = delay;
      break;
    case 4:
      if (NORMAL == mode) {
        if (fabs(delay - delay_1) >
            fabs(ev) * 2 + 0.1) {
          var = 0;
          mode = IMPULSE;
        }
      }
      else {
        var = var / 2 +
          ((int)fabs((2 * delay - delay_1 - delay_2) * 8000.0) / 8);
        if (var <= 36) {
          mode = NORMAL;
          delay_2 = delay_1;
          delay_1 = delay;
          break;
        }
      }
```

```
        if (NORMAL == mode)
          ed = 0.125 * delay + 0.875 * ed;
        else
          ed = ed + delay - delay_1;
        ev = 0.125 * fabs(delay - ed) + 0.875 * ev;
        delay_2 = delay_1;
        delay_1 = delay;
        break;
      case 5:
        ed = delay_1;
        delay_1 = delay;
        break;
      default:
        cerr << "Dieser Alg. ist nicht vorhanden!!!\n";
        break;
      }
      if (4 != algorithm)
        ev = alpha * ev + ((1 - alpha) * fabs(ed - delay));
      ed_out    << arrivalTime
        << " " << ed    << "\n";
      ev_out    << (double)((double)begin_stamp / (double)fa)
        << " " << ev    << "\n";
    }
  }

  hinclude { <netinet/in.h>, "streamCompat.h", "RTPPacket.h" }
  header {
    typedef struct {
      char set;
      char buf[160];
      double arrivalTime;
      int talkspurt;
      unsigned long stamp;
    } slot_t;
    #define time2slot(t) (unsigned int)(((unsigned int)(t*(double)fa))%(unsigned int)ringsize)
    #define LOST_RATE 0.1
    #define NORMAL 1
    #define IMPULSE 2
  }
  input {
    name { rtp }
    type { message }
    desc { Talkspurt Indication }
  }

  input {
    name { start_queue }
    type { float }
  }
  input {
    name { end_queue }
    type { float }
  }
```

```
output {
  name { rtp_out }
  type { message }
}

protected {
  // Werte zur Berechnung der Netzwerkverzoegerung
  double oldArrivalTime;  // Ankunftszeit des letzten Paketes
  unsigned long oldTs;    // Zeitstempel des letzten Paketes

  // Werte des aktuellen Paketes
  double delay;

  // Werte des aktuellen talkspurts
  unsigned long ts_stamp;     // Timestamp des ersten Paketes der aktuellen
                              // Sprechphase
  double ts_time;             // Beginn der Ausspielung der Sprechphase
  unsigned long begin_stamp;  // Timestamp des aktuellen Paketes
  unsigned long end_stamp;    // Letztes Oktett des aktuellen Paketes
  double play_time;           // Ausspielzeit des aktuellen Paketes
  double end_time;            // Endzeit des aktuellen Paketes
  int sample_cnt;             // Anzahl der Abtastwerte im aktuellen Paket

  int start;
  double        next_play_time;

  double        last_inserted_time;
  unsigned long last_stamp;


  /* Algorithmen */
  double delay_1;      // Verzoegerung des letzten Paketes
  double delay_2;      // Verzoegerung des vorletzten Paketes
  double ed;           // Schaetzwert Verzoegerung
  double ev;           // Schaetzwert Variabilitaet
  double alpha;        // Verwendete Konstante der Algorithmen
  double beta;         // Verwendete Konstante der Algorithmen
  int mode;            // Zustand imp-Algorithmus (IMPULSE oder NORMAL)
  int var;             //


  /* Outputs */
  ofstream protokoll;  // Protokoll des Simulationslaufs
  ofstream ed_out;
  ofstream ev_out;
  ofstream delay_out;
  ofstream delay_m_out;

}

constructor {
  start_queue.triggers();
  end_queue.triggers();
```

```
    start_queue.before(rtp);
    end_queue.before(rtp);
  }

  begin {
    StringList fname;
    char a, l;

    switch ((int) algorithm) {
    case 1:
      a = '1'; break;
    case 2:
      a = '2'; break;
    case 3:
      a = '3'; break;
    case 4:
      a = '4'; break;
    default:
      a = 'u'; break;
    }

    if ((int) adaption_on_late)
      l = 'y';
    else
      l = 'n';

    cout << "begin\n";
    cout << "trace: " << trace << " alg: " << algorithm
      << " talkspurt on late: " << l << "\n";


    /** Protokolldatei oeffnen      **/
    if (protokoll.rdbuf()->is_open)
      protokoll.close();
    protokoll.open("protokoll", ios::out);

    if (ed_out.rdbuf()->is_open)
      ed_out.close();

    fname = dir;
    fname << "/de." << a << l << "." << trace;

    ed_out.open(fname, ios::out);

    if (ev_out.rdbuf()->is_open)
      ev_out.close();

    fname = dir;
    fname << "/dv." << a << l << "." << trace;

    ev_out.open(fname, ios::out);

    if (delay_out.rdbuf()->is_open)
```

```
    delay_out.close();

  fname = dir;
  fname << "/dm." << trace;

  delay_out.open(fname, ios::out);

  if (delay_m_out.rdbuf()->is_open)
    delay_m_out.close();

  fname = dir;
  fname << "/dmm." << trace;

  delay_m_out.open(fname, ios::out);
}

setup {
  cout << "setup\n";
  mode = NORMAL;
  alpha = 0.998002;
  beta = 0.75;
  ev = 0;

  start = 1;
}

destructor {
  cout << "destructor\n\n";
  /** Protokolldatei schliessen          **/
  if (protokoll.rdbuf()->is_open)
    protokoll.close();

  if (ed_out.rdbuf()->is_open)
    ed_out.close();

  if (ev_out.rdbuf()->is_open)
    ev_out.close();

  if (delay_out.rdbuf()->is_open)
    delay_out.close();

  if (delay_m_out.rdbuf()->is_open)
    delay_m_out.close();
}

wrapup {
  cout << "wrapup\n\n";
  /** Protokolldatei schliessen          **/
  if (protokoll.rdbuf()->is_open)
    protokoll.close();

  if (ev_out.rdbuf()->is_open)
    ev_out.close();
```

```
    if (delay_out.rdbuf()->is_open)
      delay_out.close();

    if (delay_m_out.rdbuf()->is_open)
      delay_m_out.close();
  }


  go {
    if (rtp.dataNew) {
      Envelope env;

      rtp.get().getMessage(env);
      RTPPacket *rtp_packet = (RTPPacket *) env.myData();

      sample_cnt = rtp_packet->GetLength();
      rtp_packet->SetArrivalTime(arrivalTime);

      begin_stamp = rtp_packet->GetTs();
      end_stamp = begin_stamp + sample_cnt - 1;

      if (start) {        // Erstes Paket der Uebertragung empfangen
        start = 0;
        delay = ed = delay_1 = delay_2 = arrivalTime -
          (double)((double)begin_stamp / (double)fa);
        ev = 0.0;
        if (3 == algorithm)
          ed = HUGE;
        oldArrivalTime = arrivalTime;
        oldTs = begin_stamp;

        next_play_time = arrivalTime + initial_delay;

        rtp_packet->SetPlayTime(next_play_time);
        rtp_packet->SetStatus(RTP_OK);
        rtp_out.put(arrivalTime) << env;

        last_inserted_time  = (double)((double)(sample_cnt - 1) / (double)fa);
        last_inserted_time  += next_play_time;

        last_stamp = end_stamp;

        ts_stamp = begin_stamp;
        ts_time  = next_play_time;
        delay_out << (double)((double)begin_stamp / (double)fa)
          << " " << delay << "\n";
        delay_m_out << arrivalTime
          << " " << delay << "\n";
        rtp_packet->SetNetDelay(delay);
        Calc_estimates();
        return;
      }
```

```
delay = arrivalTime -
  (double)((double)begin_stamp / (double)fa);
rtp_packet->SetNetDelay(delay);
delay_out    << (double)((double)begin_stamp / (double)fa)
  << " " << delay << "\n";
delay_m_out    << arrivalTime
  << " " << delay << "\n";

if (rtp_packet->GetM()) {
  /*** Ein neuer talkspurt beginnt ***/
  if (NewTalkspurt()) {
    rtp_packet->SetStatus(RTP_EARLY);
    rtp_packet->SetPlayTime(arrivalTime);
    rtp_out.put(arrivalTime) << env;
    Calc_estimates();
    return;
  }
  Calc_estimates();
}
else {
  play_time = (double)((begin_stamp - ts_stamp) / (double)fa);
  play_time += ts_time;

  if ((play_time < double(start_queue%0)) && adaption_on_late) {
    if (NewTalkspurt()) {
      rtp_packet->SetStatus(RTP_EARLY);
      rtp_packet->SetPlayTime(arrivalTime);
      rtp_out.put(arrivalTime) << env;
      Calc_estimates();
      return;
    }
    Calc_estimates();
    rtp_packet->SetStatus(RTP_LATE_TALKSPURT);
  }
  else  {
    if (play_time < double(start_queue%0)) {
      /*** Paket ist zu spaet ***/
      rtp_packet->SetStatus(RTP_LATE);
      rtp_packet->SetPlayTime(arrivalTime);
      rtp_out.put(arrivalTime) << env;
      Calc_estimates();
      return;
    }
    end_time = play_time + (double)((double)(sample_cnt - 1) /
                                  (double) fa);

    /*** Gehoert das Paket zu einem frueheren talkspurt ??? ***/
    /*** Dann ist es zu spaet.                              ***/
    if (play_time < ts_time) {
      cerr <<  "Here it is...\n";
      rtp_packet->SetStatus(RTP_LATE);
      rtp_packet->SetPlayTime(arrivalTime);
```

```
            rtp_out.put(arrivalTime) << env;
            Calc_estimates();
            return;
          }

          /*** Ist das Paket zu frueh ???                          ***/
          if (end_time > double(end_queue%0)) {
            rtp_packet->SetStatus(RTP_EARLY);
            rtp_packet->SetPlayTime(arrivalTime);
            rtp_out.put(arrivalTime) << env;
            Calc_estimates();
            return;
          }
        }
      }

      rtp_packet->SetPlayTime(play_time);
      if (RTP_LATE_TALKSPURT != rtp_packet->GetStatus()) {
        rtp_packet->SetStatus(RTP_OK);
        Calc_estimates();
      }

      rtp_out.put(arrivalTime) << env;

      if (end_stamp > last_stamp) {
        last_inserted_time = end_time;
        last_stamp = end_stamp;
      }
    } // rtp.dataNew
  } // go
}
```

B.4 DERTPOut.pl

```
defstar
{
  name { RTPOut }
  domain { DE }
  derivedfrom { RepeatStar }

  //
  // Parameter
  //
  defstate {
    name { ringsize }
    type { int }
    default { "64000" }
    desc {
      Groesse der Warteschlange...
      }
  }
```

```
defstate {
  name { speaker }
  type { int }
  default { "0" }
  desc {Lautsprecher aus/ein}
}
defstate {
  name { blocksize }
  type { int }
}

hinclude { <netinet/in.h>, "streamCompat.h", "RTPPacket.h" }
header {
  #define time2slot(t) (unsigned int)(((unsigned int)((double)(t)/T)) \
                                     %(unsigned int)ringsize)
  #define SILENCE 0377
  #define T (double)125e-6
}

//
// Terminals
//
input {
  name { rtp }
  type { message }
  desc { Talkspurt Indication }
}

output {
  name { start_queue }
  type { float }
}

output {
  name { end_queue }
  type { float }
}

protected {
  int start;
  double next_play_time;   // Naechster Ausspieltermin
  int audio_out;           // Ausgabedatei
  char *slots;             // Ringspeicher
}

constructor {
  audio_out = 0;
  slots = NULL;
  delayType = TRUE;
}

setup {
  start = 1;
```

```
    if (audio_out)
      close(audio_out);

    if (speaker)
      if (-1 == (audio_out = creat("play.au", 0666)))
        Error::abortRun(*this, "can't open 'play.au'");


    // Reservieren des Ringspeichers
    if (slots) {
      LOG_DEL; delete [] slots;
    }
    LOG_NEW; slots = new char[ringsize];
    if (NULL == slots)
      Error::abortRun(*this , "Not enough memory for the ringbuffer.");
    memset(slots, SILENCE, ringsize);
  }

  destructor {
    if (NULL != slots) {
      LOG_DEL; delete [] slots;
      slots = NULL;
    }
    if (audio_out) {
      close(audio_out);
      audio_out = 0;
    }

  }

  wrapup {
    if (NULL != slots) {
      LOG_DEL; delete [] slots;
      slots = NULL;
    }
    if (audio_out) {
      close(audio_out);
      audio_out = 0;
    }
  }

  go {
    //
    // Der Stern wird in zwei Faellen aufgerufen: Neue Daten sind
    // angekommen oder eine weitere Ausspielung steht an.
    // Mit 'rtp.dataMew' kann ueberprueft werden, ob neue Daten
    // angekommen sind. Die Funktion 'canGetFired()' zeigt, ob
    // sich der Stern selbst aufgerufen hat ('refireAtTime()'),
    // was im Takt der blockweisen Ausspielung ('blocksize') geschieht.
    //
    if (rtp.dataNew) {
      Envelope env;
      int len;
```

```
    int start_slot;
    int end_slot;
    double end_time;

    rtp.get().getMessage(env);
    const RTPPacket *rtp_packet = (const RTPPacket *) env.myData();

    if (rtp_packet->GetStatus() != RTP_OK)
      return;

    len = rtp_packet->GetLength();

    end_time = (double)(len - 1) / 8000.0;
    end_time += rtp_packet->GetPlayTime();
    start_slot = time2slot(rtp_packet->GetPlayTime());
    end_slot = time2slot(end_time);

    // Die Daten des ersten Paketes werden einsortiert
    if (start) {
      start = 0;
      next_play_time = rtp_packet->GetPlayTime();
      start_queue.put(arrivalTime) << next_play_time;
      end_queue.put(arrivalTime)   << (next_play_time +
                                       ((ringsize - 1) * T));
      // Zum ersten Ausspieltermin aufrufen
      refireAtTime(next_play_time);
    }

    // Einsortieren der Daten der Pakete
    if (end_slot >= start_slot)
      memcpy(slots + start_slot, rtp_packet->GetBuf(), len);
    else {
      // ring-wrap
      int up_to_wrap = ringsize - start_slot;
      memcpy(slots + start_slot, rtp_packet->GetBuf(), up_to_wrap);
      memcpy(slots, rtp_packet->GetBuf() + up_to_wrap, end_slot + 1);
    }
  } // rtp.dataNew

  if (canGetFired()) {
    int start_slot;
    int end_slot;

    if ((arrivalTime != next_play_time) || start)
      return;

    start_slot = time2slot(next_play_time);
    end_slot = time2slot(next_play_time +
                         (T * (double)(blocksize - 1)));

    if (end_slot >= start_slot) {
      if (speaker)
        write(audio_out, slots + start_slot, blocksize);
```

```
        memset(slots + start_slot, SILENCE, blocksize);
      }
      else {
        // ring-wrap
        int up_to_wrap = ringsize - start_slot;
        if (speaker)
          write(audio_out, slots + start_slot, up_to_wrap);
        memset(slots + start_slot, SILENCE, up_to_wrap);
        if (speaker)
          write(audio_out, slots, end_slot + 1);
        memset(slots, SILENCE, end_slot + 1);
      }

      next_play_time += T * (double)blocksize;

      start_queue.put(arrivalTime) << next_play_time;
      end_queue.put(arrivalTime)   << (next_play_time +
                                       ((ringsize - 1) * T));

      // Aufruf zum naechsten Ausspieltermin
      refireAtTime(next_play_time);
    }
  }
}
```

Literaturverzeichnis

[Com91] Comer, Douglas E.: *Principles, Protocols, And Architecture*, Band 1 der Reihe *Internetworking with TCP/IP*. Prentice-Hall, Englewood Cliffs, New Jersey, Zweite Auflage, 1991.

[CS91] Comer, Douglas E. und David L. Stevens: *Design, Implementation, and Internals*, Band 2 der Reihe *Internetworking with TCP/IP*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Dee89] Deering, S.: *Host extensions for IP multicasting.* Request for Comments (Standard) STD 5, RFC 1112, Internet Engineering Task Force, August 1989. (Obsoletes RFC0988).

[Hed88] Hedrick, C.: *Routing Information Protocol.* Request for Comments (Standard) STD 34, RFC 1058, Internet Engineering Task Force, Juni 1988. (Updated by RFC1723, RFC1388).

[ITU92] ITU: *G.711 (64 kbit/s 8kHz 8-bit PCM audio encoding)*, Dezember 1992. Recommendation currently in force.

[Lip90] Lippman, Stanley B.: *C++ Einführung und Leitfaden.* Addison-Wesley (Deutschland) GmbH, 1990.

[Mon83] Montgomery, Warren A.: *Techniques for Packet Voice Synchronization.* IEEE Journal on Selected Areas in Communications, SAC-1(6):1022–1028, Dezember 1983.

[Nol92] Noll, Peter: *Nachrichten Übertragung I.* Skript, Technische Universität Berlin – Institut für Fernmeldetechnik, 1992.

[Par93] Partridge, Craig: *Gigabit networking.* Addison-Wesley, Reading, Massachusetts, 1993.

[Pos81] Postel, J.: *Transmission Control Protocol.* Request for Comments (Standard) STD 7, RFC 793, Internet Engineering Task Force, September 1981.

[RKTS94] Ramjee, Ramachandran, Jim Kurose, Don Towsley und Henning Schulzrinne: *Adaptive playout mechanisms for packetized audio applications in wide-area networks.* In: *infocom*, Toronto, Canada, Juni 1994.

[SCFJ95] Schulzrinne, Henning, Stephen Casner, Ron Frederick und Van Jacobson: *RTP: A Transport Protocol for Real-Time Applications*. Internet draft (work-in-progress) *draft-ietf-avt-rtp-*.txt*, März 1995.

[Sch95a] Schulzrinne, Henning: *Guide to NeVoT*. GMD Fokus, Berlin, Germany, 3.32 Auflage, September 1995.

[Sch95b] Schulzrinne, Henning: *Internet Services: from Electronic Mail to Real-Time Multimedia*. KIVS'95 (Kommunikation in verteilten Systemen), 2 1995.

[Sch95c] Schulzrinne, Henning: *RTP Profile for Audio and Video Conferences with Minimal Control*. Internet Draft, GMD Fokus, März 1995. Work in progress.

[SH95] Spaniol, Otto und Simon Hoff: *Ereignisorientierte Simulation - Konzepte und Systemrealisierung*. International Thomson Publishing GmbH, Bonn, Erste Auflage, 1 1995.

[Uni94a] University of California at Berkeley – Department of electrival and computer sciences, Berkeley, California: *The Almagast Vol. 1 - Ptolemy 0.5.1 User's Manual*, 1994.

[Uni94b] University of California at Berkeley – Department of electrival and computer sciences, Berkeley, California: *The Almagast Vol. 3 - Ptolemy 0.5.1 Programmer's Manual*, 1994.