Spezifikation und Implementierung eines erweiterten RTP-Translators mit dynamischen Konfigurationsmerkmalen

am Beispiel eines Audio-Transcoders

Mathias Kretschmer
GMD-IMK.IBS

30. Januar 1998

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
SMP	Service Management Protocol
CNI	Client Network Interface
CSRC	Contributing Source (RTP)
DD	Data Description
DVD	Digital Versatile Disk
DVMRP	Distance Vector Multicast Routing Protocol
EA	End Application
FEC	Forward Error Correction
FIFO	First In First Out
IETF	Internet Engineering Task Force
IMDCT	Inverse Modified Discrete Cosine Transform
IP	Internet Protocol
ISDN	Integrated Services Digital Network
ITU	International Telecommunication Union
KISS	Kommunikation-Infrastruktur für Streaming-Services
LAN	Local Area Network
MAN	Metropolitan Area Network
MDCT	Modified Discrete Cosine Transform
MPEG	Moving Pictures Expert Group
MT	Multi Threading
NAP	Network Access Point
NNI	Network-Network-Interface
PCM	Pulse Code Modulation
PIM	Protocol Independend Multicast
QoS	Quality of Service
RFC	Request For Comment
RSVP	Resource Reservation Protocol
RTP	Realtime Transport Protocol
RTCP	RTP Control Protcol
SA	Service Agent
SD	Service Description
SSRC	Synchronisation Source (RTP)
TCP	Transmission Control Protocol
TS	Timestamp (RTP)
TTL	Time To Live
UDP	User Datagram Protocol
WAN	Wide Area Network

Kapitel 1

Einleitung

Seit einigen Jahren erfeuen sich *live*-Audio- und Videoübertragungen im Internet, wie Tele-Conferencing oder Live-Broadcasting, immer größerer Beliebtheit. Diese *Realtime Streaming* genannte Technik ermöglicht es, digitalisierte Audio-/Videoinformationen - im Folgenden Multimedia-Daten genannt - über das Internet von einem Benutzer zum anderen zu übertragen. Bedingt durch die Vielzahl von verfügbaren aber zueinander inkompatiblen Streaming-Protokollen muß sich der Benutzer für jedes verwendete System eine spezielle Client-Applikation, den sogenannten "Player", installieren und diesen ggf. auch warten und auf dem neuesten Stand halten. Hier wäre es wünschenswert, wenn der Benutzer nur einen Client installieren müßte und entsprechende Protokollumsetzungen transparent im Netz stattfinden könnten. Mit diesem Ansatz wäre es ebenfalls möglich, einen Multimediastrom in mehreren verschiedenen Bitraten anzubieten, um die Benutzer je nach zur Verfügung stehender Bandbreite (z.B 28.8 kBit/s Modem, 64 kBit/s ISDN, 10 MBit/s Ethernet) mit dem für sie besten Strom zu versorgen. Um zu verhindern, daß bei mehreren Zuhörern für jeden einzelnen derselbe Multimediastrom übertragen werden muß, sollte soweit möglich, ein Multicast-fähiges Protokoll verwendet werden.

In dem Artikel "Forget The Net" von K. Jonas [1] sind Vorschläge für eine "Kommunikations-Infrastruktur für multimediale Streaming-Dienste im heterogenen Umfeld" vorgestellt worden, die im Laufe des letzten Jahres konkretisiert und teilweise realisiert wurden. Als Transport-Protokoll für die Multimediaströme wurde das "Realtime Transport Protocol" (RTP) [4] implementiert, welches von den meisten MBone-Anwendungen und auch einigen kommerziellen Streaming-Systemen verwendet wird. Im Rahmen dieser Arbeit wurde ein Audiotranscoder spezifiziert und implementiert, der es gestattet, in Echtzeit Audioströme von einem Coding-Format in ein anderes umzurechnen. Als weiteren Dienst bietet der Transcoder die Möglichkeit, die Datenrate eines vorhandenen Stromes zu verringern, um Teilnehmer mit schmalbandigerem Internetzugang gesondert bedienen zu können, ohne die Datenrate und damit auch die Qualität des Audiostromes für alle Teilnehmer verringern zu müssen. Dabei werden die Synchronisationsinformationen entsprechend angepaßt, so daß ein Client bei Bedarf (z.B. Schwankungen der zur Verfügung stehenden Bandbreite) zwischen den verschiedenen Strömen wechseln kann. Abbildung 1.1 zeigt ein solches Szenario, welches der Funktionalität eines RTP-Translators entspricht, wie er im Zusammenhang mit RTP spezifiziert wurde. Dieser in der RTP-Spezifikation angedachte Ansatz wird um ein Protokoll erweitert, das es gestattet, die Dienste des Audio-Transcoders dynamisch anzufordern.

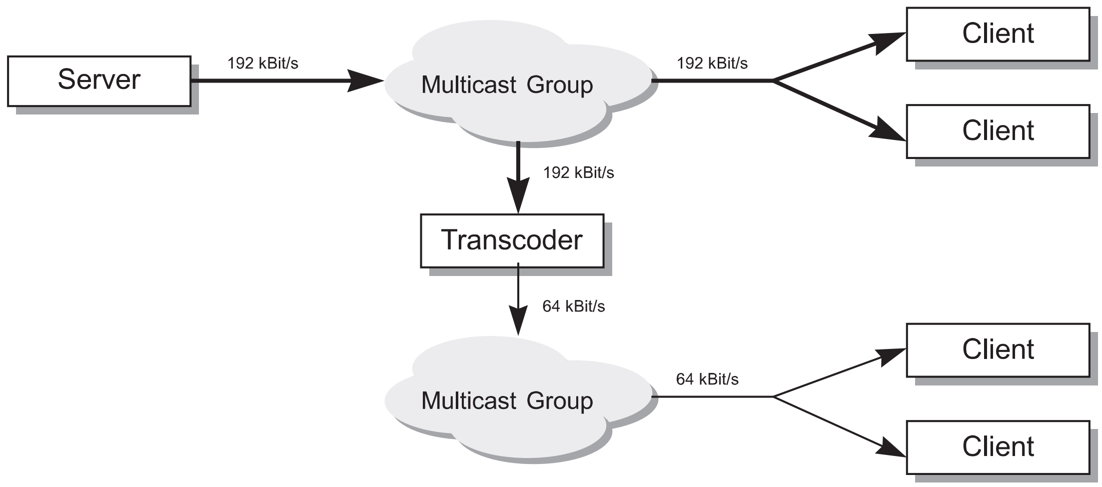

Abbildung 1.1: Der Audio-Transcoder als Bitraten-Umsetzer

Die folgenden Kapitel beschreiben die wissenschaftlichen und technischen Grundlagen dieser Arbeit, gefolgt von der Beschreibung des KISS-Konzeptes auf dem das hier vorgestellte Szenario beruht. Anschließend werden typische Anwendungsfälle für den Audio-Transcoder vorgestellt und auf diesen basierend der Audio-Transcoder entworfen. Im darauf folgenden Kapitel wird die Implementierung des Audio-Transcoders beschrieben. Abschließend erfolgt eine Bewertung des Audio-Transcoders.

Da das gesamte Szenario in enger Zusammenarbeit mit Jens Mödeker entstanden ist, wurden Teile der Grundlagen sowie das KISS-Konzept gemeinsam bearbeitet.

Kapitel 2

Grundlagen

Dieses Kapitel beschreibt die technischen und wissenschaftlichen Grundlagen dieser Arbeit. Dabei werden die Netzwerkprotokolle unter dem Aspekt ihrer Tauglichkeit für Realtime-Streaming-Anwendungen betrachtet.

2.1 Anforderungen von Realtime-Streaming

Die wohl bekanntesten Anwendungen von Realtime-Streaming sind Radio, Fernsehen und das Telefon. Bei diesen wird jeweils ein kontinuierlicher Multimediastrom vom Sender zu einem oder mehreren Empfängern, mit möglichst geringer Verzögerung, übertragen. Dies stellt hohe Anforderungen an das Übertragungsmedium, da eine konstante Bandbreite bereitgestellt werden sollte. Für Radio- und Fernsehübertragungen z.B. werden deshalb feste Kanäle mit einer festen Bandbreite verwendet. Verzögerungen der Datenübertragung, mit Ausnahme der physikalischen Laufzeit, treten hier in der Regel nicht auf. Daraus resultieren die folgenden Anforderungen an ein Realtime-Streaming-System:

- kurze konstante Laufzeiten der Daten
- garantierte Mindestbandbreite
- geringe Fehlerrate

Seit einiger Zeit wächst die Bedeutung des Internets als Übertragungsmedium für Multimediaströme. Im Folgenden wird daher anhand der verfügbaren Protokolle untersucht, inwieweit das Internet den o.g. Anforderungen genügt.

2.2 Aufbau des Internets

Das Internet ist das größte momentan existierende heterogene Netzwerk, das Rechner mit völlig verschiedener Hard- und Software miteinander verbindet, solange diese das "Internet Protocol" (IP, RFC 791 [5]) unterstützen. IP ist ein unzuverlässiges paket-orientiertes Netzwerkprotokoll, das unabhängig von der installierten Netzwerktopologie (Ethernet, Token Ring, ATM, X.25, etc.) verwendet werden kann. Dabei bedeutet unzuverlässig, daß gemäß dem "best effort"-Prinzip nicht gewährleistet ist, daß ein versendetes Paket auch wirklich den Empfänger erreicht. Hinzu kommt, daß die Laufzeiten der einzelnen Pakete (z.B. durch die starken Schwankungen der Netzauslastung) variieren. Damit entsteht eine Varianz des Intervalls in dem die Pakete empfangen werden (Jitter). Dies erfüllt nicht die Anforderungen von Streaming-Diensten wie z.B. Telefonieren oder Live Broadcasting, die einen konstanten kontinuierlichen Datenfluß erfordern. Die auf IP basierenden Protokolle TCP (RFC 761 [6]) und UDP (RFC 768 [7]) werden in der Regel von jedem IP-fähigen Betriebssystem unterstützt. Es liegt daher nahe, sie als Basisprotokolle für die Implementierung von Realtime-Streaming-Diensten zu verwenden.

2.2.1 Das TCP Protokoll

Das "Transmission Control Protocol" überträgt einen byte-orientierten Datenstrom fehlerfrei über das paket-orientierte IP-Netz und simuliert damit eine bidirektionale "Punkt-zu-Punkt" Verbindung. Dabei wird, gemäß dem "best effort"-Prinzip, keine Mindestdatenrate garantiert. TCP organisiert Bestätigungen (Acknowledgements) für empfangene Pakete und ggf. die Wiederversendungen (Retransmission) verlorener Pakete, um die fehlerfreie Übertragung des Datenstroms zu garantieren. Diese Merkmale sind für übergeordnete Anwendungen transparent und von diesen nicht beeinflußbar. Sendet nun eine Multimediaanwendung mit einer konstanten Datenrate und entstehen dabei Netzwerkprobleme, so wird der sendende Prozeß blockiert bis z.B. durch Retransmissions fehlende Daten übertragen wurden. Je nach Umfang dieser Probleme kann dies zum "Abreißen" des Datenstromes und zum Stocken der Wiedergabe der Multimediadaten auf Empfängerseite führen.

2.2.2 Das UDP Protokoll

Das "User Datagram Protocol" fügt dem IP-Header im wesentlichen die Möglichkeit hinzu, mehrere virtuelle "Punkt-zu-Punkt" Verbindungen unterscheiden zu können. Hierzu wird zusätzlich die sogenannte Portnummer übertragen. Ansonsten ist UDP ebenso unzuverlässig wie IP selbst. Durch seine einfache Spezifikation eignet es sich als Basis, um darauf neue, speziell auf die Anforderungen von Realtime-Streaming-Applikationen zugeschnittene, Protokolle aufzubauen. Diese sollten Paketverluste, Paket-Jitter[1] sowie in falscher Reihenfolge eintreffende Pakete erkennen, und darauf geeignet reagieren.

2.2.3 Das IP-Multicast Protokoll

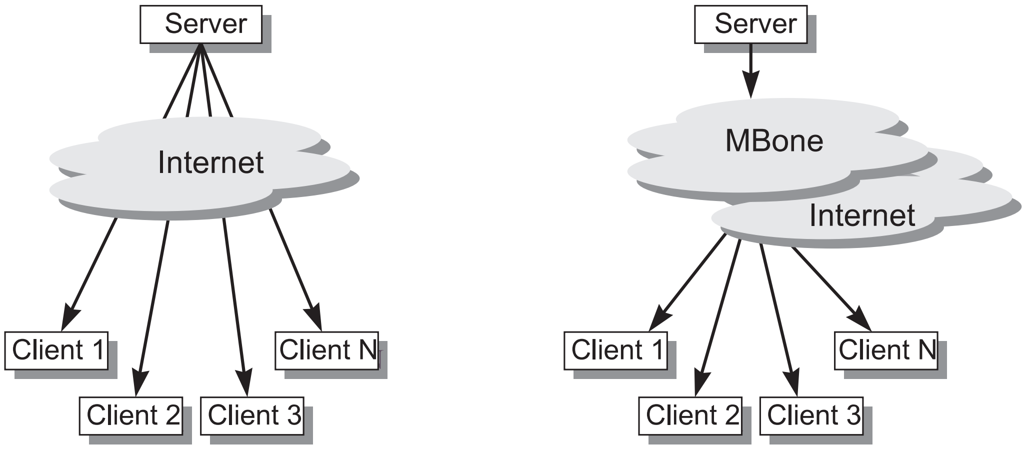

Abbildung 2.1: IP Unicast vs. IP-Multicast

IP-Multicast (IP-MC) stellt eine Erweiterung des IP Protokolls dar, die n:m-Verbindungen sowie ein möglichst effizientes Verteilen (Routing) der Pakete unterstützt. Dies ist besonders für Realtime-Streaming-Anwendungen interessant, da hier in der Regel ein Sender viele Empfänger mit dem selben Datenstrom versorgt (Abbildung 2.1). Im Unterschied zu IP adressiert IP-Multicast nicht einen Rechner sondern eine sogenannte Multicast-Gruppe, deren Adresse aus einer IP-Adresse des "Class D"-Netzes 224.0.0.0 bis 239.255.255.255 besteht. Spezielle Routing-Algorithmen wie DVM-RP (RFC 1075 [10]) oder PIM (RFC 2117 [11]) sorgen dafür, daß jeder Rechner,

[1] vgl. Abschnitt 2.2

der Mitglied einer Multicast-Gruppe ist, auch die entsprechenden Daten erhält. IP-Multicast wird im LAN von fast allen gängigen Betriebssystemen unterstützt. Multicast-Routing dagegen befindet sich zur Zeit noch in der Erprobungsphase. So sind am MBone, einem IP-Multicast Overlay-Netz im Internet, nur Forschungseinrichtungen und einige wenige Firmen angeschlossen. Es ist daher praktisch unmöglich eine breite Masse von Endbenutzern direkt per IP-Multicast zu erreichen.

2.2.4 Realtime-Streaming im Internet

Streaming in heterogenen Datennetzen unterscheidet sich grundsätzlich von herkömmlichen Übertragungstechniken, da die Kommunikation nicht mehr auf der Basis eines kontinuierlichen Stromes stattfindet, sondern auf einem virtuellen Strom von Datenpaketen (datagrams) fester oder variabler Länge beruht. In der Regel werden viele virtuelle Kanäle gemultiplext über eine physikalische Leitung übertragen. Dies ist für Netzwerke mit garantierten Mindestbandbreiten und konstanten Paketlaufzeiten wie z.B. ATM (Asynchronous Transfer Mode) spezifiziert [12]. Ein IP-basiertes Netz wie das Internet arbeitet allerdings nach dem "best effort" Prinzip. Es ist weder möglich, eine Mindestbandbreite zu garantieren, noch konstante Paketlaufzeiten anzufordern. In der Praxis zeigt sich, daß Bandbreite und Paketlaufzeiten innerhalb von Sekunden sehr starken Schwankungen unterliegen können. Es ist sogar möglich, daß ein Paket ein anderes "überholt" (Disordering). Dazu kommt, daß die Paketverluste, die aufgrund überlasteter Verbindungen bzw. Router auftreten, die "Leitungsqualität" des virtuellen Kanals teilweise drastisch verschlechtern.

Als Lösung hierfür existieren Ansätze wie das Resource Reservation Protocol (RSVP, IEFT Draft RVSP-MIB [13]), die es erlauben sollen, die Dienstqualität (Quality of Service) für IP-Verbindungen zu definieren. U.a. wegen deren geringer Verbreitung ist man momentan auf die Verwendung der Standard-Protokolle UDP und TCP beschränkt.

TCP ist für Streaming-Anwendungen nur bedingt geeignet. Übertragungsfehler äußern sich bei TCP in einer Verzögerung bei der Wiedergabe. Bei einer "schlechten Leitung" kann es dazu führen, daß sich die Verzögerung zwischen Aufnahme beim Sender und Wiedergabe beim Empfänger immer mehr vergrößert. Im schlimmsten Falle folgt ein Verbindungsabbruch. Eine steigende Verzögerung widerspricht jedoch den Anforderungen eines Realtime-Streaming-Systems.

Bei UDP hingegen äußern sich Übertragungsfehler durch "Aussetzer", also kurzzeitigem Informationsverlust. Dieser könnte ggf. durch eine Vorpufferung und gezielte Retransmission ausgeglichen werden. Ein solcher Mechanismus muß vom darüber liegenden Protokoll implementiert werden, was bzgl. Retransmissions bei TCP nicht nötig wäre. Ein weiterer Vorteil von TCP ist der einfache Aufbau von Verbindungen durch eine Firewall hindurch, jedoch wird sein Verhalten bei Übertragungsfehlern als gravierenderer Nachteil gewertet und deshalb ein erhöhter Aufwand bei der Implementierung eines UDP-basierten Protokolls inkaufgenommen. Für UDP spricht auch, daß es mit IP-Multicast zusammenarbeitet und so unnötige Mehrfachversendungen vermieden werden. Es ist daher sinnvoll, ein geeignetes Protokoll basierend auf UDP zu entwickeln.

2.3 Das Realtime Transport Protocol (RTP)

Das “Realtime Transport Protocol” (RTP, RFC 1889 [4]) wurde speziell für Echtzeitanwendungen (z.B. Videokonferenzen) entwickelt. Es eignet sich besonders gut für die Verwendung mit Multicast-Protokollen, läßt sich aber auch mit Unicast-Protokollen kombinieren. Eine Identifizierung eines Multimediastromes erfolgt mit Hilfe einer sogenannten “Synchronisation Source” (SSRC), die nicht an eine Netzwerkadresse gebunden ist. Der Begriff “Realtime Transport Protocol” ist insofern irreführend, da das RTP keine Garantien bezüglich QoS oder für die pünktliche Auslieferung von Paketen macht. “Realtime” ist hier vielmehr im Zusammenhang mit dem Charakter der zu übertragenden Daten zu sehen.

RTP kann als eigene Protokollschicht implementiert werden. In der Regel wird es allerdings in die Streaming-Applikationen integriert und stellt diesen zusätzliche Möglichkeiten zur Synchronisation, Identifikation (bzw. Demultiplexing) und Fehlererkennung von Multimediaströmen in verteilten Multimediasystemen zur Verfügung. Hierzu sollte für die beteiligten Datenströme ein “RTP Profile” spezifiziert werden, in dem festgelegt wird, welchem “Payload Type” (Art/Kodierung der transportierten Daten) welches Coding-Format zugeordnet wird. Muß dieses Coding-Format noch genauer spezifiziert werden (z.B. Übermittlung der Abtastrate, etc.), so kann die Erweiterungsmöglichkeit (Extension) des RTP-Headers benutzt werden. In RFC 1890 [9] werden einige “Standardprofiles” definiert, die z.B. von den MBone-Tools verwendet werden.

Das optional zu implementierende “RTP Control Protocol” (RTCP, RFC 1889) erweitert RTP um die Möglichkeit, Kontrollmeldungen zu verschicken sowie zu empfangen (Feedback). Dadurch wird das Streaming-System um folgende Möglichkeiten erweitert:

- Überwachung der Datenübertragungsqualität (Monitoring)
- Übertragung von Sessioninformationen (Name, etc.)
- Identifizierung von Gruppenteilnehmern

Für die Realisierung einer RTCP-Verbindung wird eine weitere Netzwerkverbindung benötigt, bei UDP-IP wird hierzu ein benachbartes Port benutzt. Da RTCP nicht Inhalt dieser Arbeit ist, wird für detailliertere Angaben über Funktionalität und Möglichkeiten auf RFC 1889 Kapitel 6 verwiesen.

RTP berücksichtigt bereits den Bedarf, Empfänger mit unterschiedlichen Dienstmerkmalen zeitgleich zu bedienen. So ist es nicht notwendig, daß Gruppenmitglieder mit geringen Qualitätsansprüchen oder Mitglieder, denen eine geringere Bandbreite zur Verfügung steht, die Dienstmerkmale für alle Kommunikationspartner vorgeben. Hierfür werden im RFC 1889 der sogenannte Translator sowie ein Mixer definiert. Beide Module sind auf der RTP-Ebene angeordnet. In der Praxis jedoch finden beide Konzepte bisher kaum Anwendung, werden aber im Folgenden beschrieben, da diese Arbeit ein darauf aufbauendes, erweitertes Konzept verfolgt.

2.3.1 Der Translator

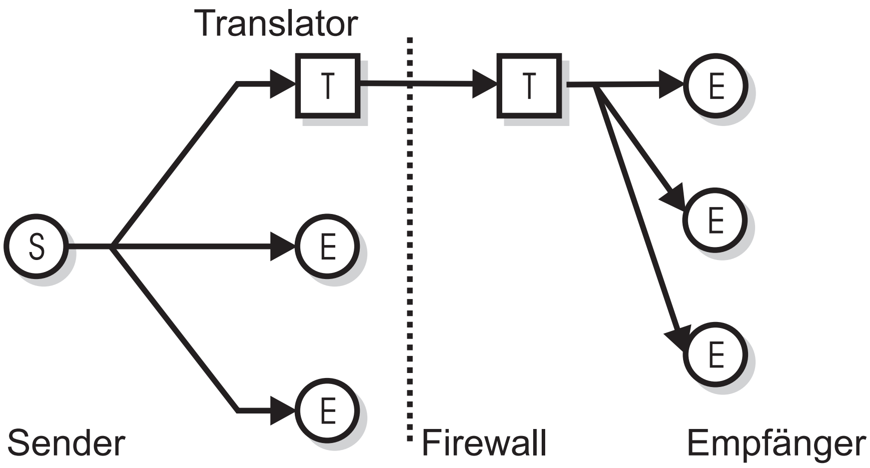

Abbildung 2.2: RTP Translator als Firewall-Tunnel

Der Translator ist ein Modul, das RTP-Pakete eines Stromes durchreicht (Forwarding), ohne die "Synchronisation Source" (SSRC) zu verändern. Dadurch können z.B. in Abbildung 2.2 die Empfänger hinter der Firewall auch weiterhin den originalen Sender des Multimediastromes identifizieren. Ein Translator kann z.B. verwendet werden, um einen bereits existierenden Multimediastrom auf eine niedrigere Datenrate umzusetzen, das Coding-Format zu ändern oder um eine Firewall zu durchtunneln. Beim Erzeugen des neuen Multimediastromes muß darauf geachtet werden, daß die Zeitstempel (Timestamps) angepaßt werden. Somit ist es möglich, daß ein Empfänger aufgrund der identischen SSRC erkennt, daß beide Ströme den gleichen Inhalt haben. Er kann daher beliebig zwischen beiden Strömen hin- und herschalten und dabei mit Hilfe der Zeitstempel die Ströme synchronisieren.

2.3.2 Der Mixer

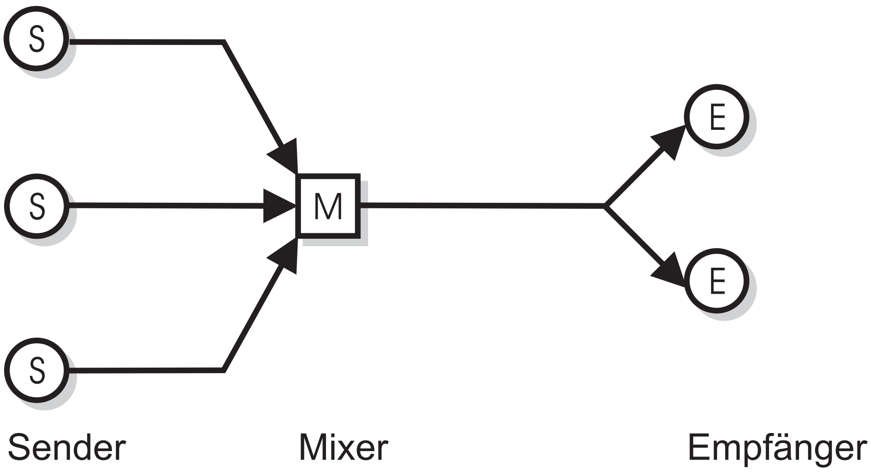

Abbildung 2.3: RTP Mixer

Der Mixer empfängt RTP-Pakete mehrerer Ströme, mischt diese zusammen und erzeugt aus ihnen einen einzigen Strom. Da hier nicht aus den verschiedenen Quellen ein gemeinsamer Zeitstempel generiert werden kann, wird eine neue SSRC belegt und ein neuer Zeitstempel erzeugt. Alle beteiligten Quellen werden als sogenannte "Contributing Source" (CSRC) an den neuen RTP-Header angehängt, um die Originalströme auch weiterhin identifizieren zu können.

Neben der eigentlichen Datenübertragung, wie RTP sie unterstützt, muß ein reales Streaming-System noch weitere Eigenschaften aufweisen. Diese werden im Folgenden beschrieben.

2.4 Ein generisches Streaming-System

Wie in dem Artikel “Audio Streaming On The Internet” [2] beschrieben, besteht ein generisches Internet-Streaming-System (Abbildung 2.4) aus folgenden Komponenten:

- einer Audioquelle (CD-Player, Sat.-Receiver, etc.)
- einem Digitalisierer (A/D-Wandler)
- einem echtzeitfähigen Encoder-Modul
- dem Streaming-Server
- dem Streaming-Client

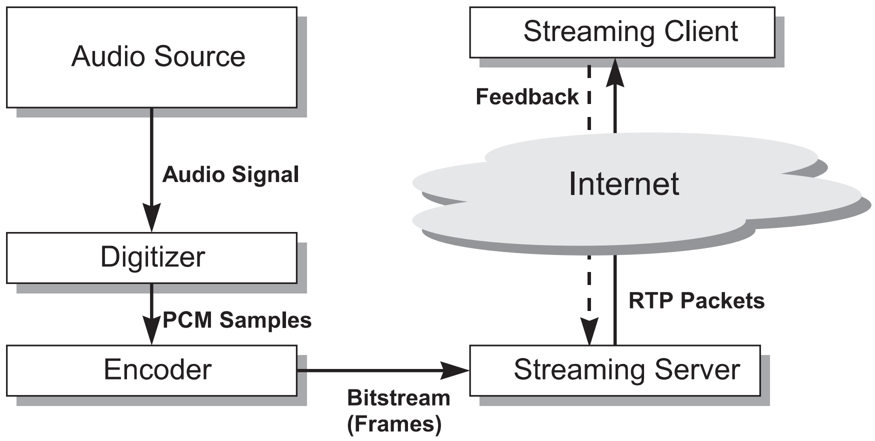

Abbildung 2.4: Generisches Streaming-System

Das Analogsignal der Audioquelle wird digitalisiert und im PCM-Format (Pulse Code Modulation) dem Encoder übergeben. Dieser encodiert sie in Echtzeit und erzeugt sogenannte Bitframes[2], die je nach verwendetem Encoder eine feste oder variable Länge haben. Das Servermodul packt einen oder mehrere solcher Bitframes in ein Paket, stellt einen Kontroll-Header (z.B. RTP), der Informationen wie Sequenznummer und Timestamps enthält, voran und sendet dieses Paket zu den jeweiligen Empfängern. Bei der Entscheidung, wieviele Bitframes in ein Paket gepackt werden, muß zwischen dem Overhead, den der Kontroll-Header erzeugt, und dem Informationsverlust, der im Falle

[2] komprimierte Pakete, die die encodierten PCM-Samples enthalten

eines Paketfehlers entsteht, abgewogen werden. Im RTP-Kontext wird eine maximale Audio-Spielzeit eines Paketes von 20 ms vorgeschlagen.

Auf der Client-Seite werden die empfangenen Pakete in einem FIFO-artigem Zwischenpuffer gespeichert, um Jitter auszugleichen oder um ggf. ein Reordering der Pakete vornehmen zu können. Fehlende Pakete könnten erneut angefordert werden (Retransmission), sofern die Vorpufferzeit dafür ausreicht. Ein verbleibender Paketverlust wird evtl. vom Decoder automatisch korrigiert (Error Concealing) oder muß explizit durch einen "leeren" PCM-Block ersetzt werden, damit Server und Client synchronisiert bleiben. Je nach Implementierung und Anwendungsszenario kann der Client Feedbacks schicken und dem Server und/oder den anderen Teilnehmern Informationen über sich (Benutzername, Paketverlust-Rate, etc.) zu kommen lassen.

2.5 Komprimierung von Audio-Informationen

Komprimierunsverfahren für Audio-Informationen wurden zur Reduzierung der Datenrate, die für die Übertragung und Speicherung benötigt wird, entwickelt. Die wohl bekanntesten Verfahren sind von der Moving Pictures Expert Group (MPEG) unter MPEG-I [14] und MPEG-II [15] standardisiert worden und waren ursprünglich für qualitativ hochwertige Anwendungen wie z.B. die digitale Audio-/Video-Verarbeitung gedacht. Auf dem amerikanischen Markt ist auch der von den Dolby Laboratories entwickelte AC-3 [17] Standard für die Audiokompression von großer Bedeutung. Des weiteren gibt es standardisierte Verfahren, die speziell für niedrige Datenraten (z.B. bei ISDN- oder Mobiltelefon) gedacht sind. Hier sind ITU-T G.723.1 [18], G.711 (A-law) [19] oder GSM zu nennen. Der Nachteil dieser Verfahren ist allerdings, daß sie speziell für Sprachübertragung optimiert wurden. Deshalb werden in den auf dem Markt verfügbaren Audio-Streaming-Systemen hauptsächlich die universelleren MPEG bzw. AC-3 Audiocoder verwendet. Diese enthalten in der Regel einige herstellerspezifische Erweiterungen, da weder MPEG noch AC-3 die derzeit im Internet üblichen Bitraten von ca. 20 kBit/s unterstützen.

2.5.1 Funktionsweise eines Audio Encoders

Fast alle universellen Audio-Encoder arbeiten nach dem verlustbehafteten Prinzip des "Perceptual Audio Coding". Das analoge Eingangssignal wird digitalisiert und die so gewonnenen digitalen Abtastwerte (Samples) werden in festgelegten Blöcken (Window Size) mit Hilfe einer MDCT (Modified Discrete Cosine Transform) Fouriertransformiert. Die MDCT wird verwendet, da sie unabhängig vom Eingangssignal, nur gerade Koeffizienten liefert und somit das Phasenspektrum nicht beachtet werden muß. Im Frequenzbereich (Frequency Domain) wird dann das Amplitudenspektrum mit Hilfe des sogenannten "psychoakustischen Modells" analysiert. Hierbei nutzt man die Schwäche des menschlichen Gehörsystems aus, verschiedene gleichzeitig auftretende Frequenzen teilweise nicht differenzieren zu können.

Abbildung 2.5 (Quelle: http://www.dolby.com/tech/parametr.html) zeigt, wie die Hörschwellenkurve des menschlichen Gehörs sich verändert, wenn dem Nutzsignal ein konstanter 1kHz-Sinus-Ton (Maskierungston) mit einer Lautstärke (Acoustic Level) von 41...101 dB SPL (0 dB SPL entsprechen 20 μpa) überlagert wird. Man erkennt deutlich, daß die Empfindlichkeit des Gehörsystems für Frequenzen im Bereich von

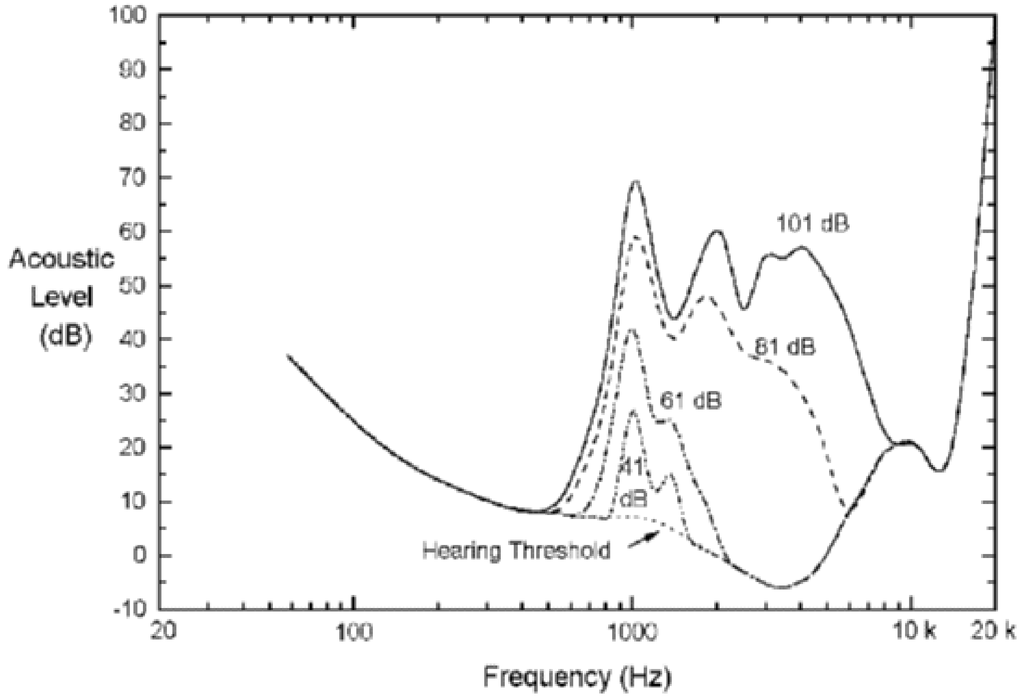

Abbildung 2.5: 1 kHz Masking-Threshold-Kurve

800 Hz...10 kHz erheblich abnimmt. Variiert man den Maskierungston, so verändert sich in Abhängigkeit davon auch die Hörschwellenkurve. Das Amplitudenspektrum wird nun daraufhin untersucht, welche Koeffizienten von den anderen maskiert werden. Darauf basierend wird eine sogenannte Maskierungs-Kurve berechnet, und alle maskierten Koeffizienten werden eliminiert. Die übriggebliebenen werden quantisiert und mit Hilfe von normierten Tabellen (Huffman-Codebooks) kodiert, da man bestrebt ist, die übrigen Koeffizienten mit möglichst wenigen Bits darzustellen. Wenn die geforderte Datenrate so niedrig ist, daß nicht mehr alle Koeffizienten übertragen werden können, werden zusätzlich die Koeffizienten eliminiert, deren Verlust am wenigsten hörbar ist. Aus einem Block PCM-Samples ist nun ein Bitframe erzeugt worden, der alle relevanten Informationen enthält, um die PCM-Samples weitgehend rekonstruieren zu können.

2.5.2 Funktionsweise eines Audio Decoders

Die vom Encoder erzeugten Bitframes werden nun übertragen und beim Empfänger wieder zu einem Amplitudenspektrum decodiert. Durch das Anwenden einer IMDCT (Inverse Modified Discrete Cosine Transform) erhält man wieder die zugehörigen Samples im Zeitbereich (Time Domain). Diese können nun mittels eines D/A-Konverters analogisiert und wiedergegeben werden.

2.5.3 Vergleich mit verlustfreien Verfahren

Die hier vorgestellten verlustbehafteten Verfahren unterscheiden sich grundsätzlich von verlustfreien Komprimierungsverfahren wie etwa ZIP oder LHA. Ihre Aufgabe ist nicht die meßtechnisch fehlerfreie Kodierung bzw. Rekonstruktion des Audio-Signals, sondern eine Rekonstruktion des Originals, so daß das menschliche Gehörsystem keine oder möglichst geringe Unterschiede feststellen kann.

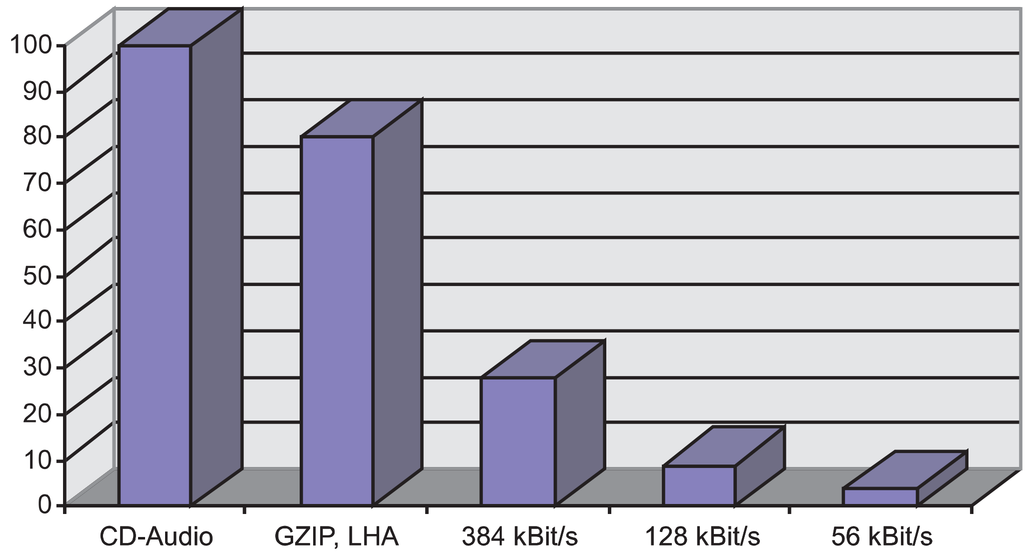

Abbildung 2.6: Vergleich der Kompressionsraten

Verlustfreie Komprimierungsverfahren erzeugen eine Datenmenge von ca. 85 % des Originals. Im Zusammenhang mit verlustbehafteten Audiocodecs gelten Audio-Ströme, die mit einer Datenrate von 384 kBit/s (28 %) codiert sind, als für das menschliche Ohr nicht unterscheidbar. Auch bei Strömen mit 128 kBit/s Datenrate (9 %) können von einem Durchschnittsgehör in der Regel keine Unterschiede mehr wahrgenommen werden.

Selbst bei einer Datenrate 56 kBit/s (4 %) erreicht man immer noch näherungsweise UKW-Radioqualität. Abbildung 2.6 veranschaulicht diesen Zusammenhang graphisch.

2.5.4 Mehrfachkodierungen

Wird ein Audiosignal mehrfach mit verlustbehafteten Verfahren kodiert, so führt dies unweigerlich zu immer größeren Qualitätsverlusten im Vergleich zum Original, da diese Verfahren nur versuchen (können) das *jeweilige* Einganssignal, für ein menschliches Gehör nicht unterscheidbar, zu rekonstruieren. Meßtechnisch enstehen hier bisweilen beträchliche Fehler, die sich fortpflanzen und auch die Klangqualität immer mehr beeinflussen. Bei höheren Bitraten bewirken Mehrfachkodierungen geringfügigere Verschlechterungen des Audiosignals als bei niedrigeren Bitraten. Eine Mehrfachkodierung von einer hohen Bitrate (>=128 kBit/s) auf eine niedrige Bitrate (<=32 kBit/s) dagegen erzeugt fast keine zusätzlichen hörbaren Verluste.

2.6 Zusammenfassung

Mit dem Internet als Medium, UDP als paketorientiertem Protokoll und den Synchronisationsmechanismen von RTP hat sich für homogene Multicast Szenarien ein Realtime-Streaming-Ansatz bewährt. Für ein heterogenes Umfeld fehlen jedoch Protokollmechanismen, die das in der RTP Spezifikation vorgesehene Translator/Mixer Konzept dynamisch und flexibel verwalten. Im folgenden Kapitel wird ein Konzept entwickelt, daß entsprechende Protokollmechanismen unterstützt.

Kapitel 3

Das KISS Konzept

Dieses Kapitel erläutert das Konzept von KISS, einer "**K**ommunikations-**I**nfrastruktur für **S**treaming-**S**ervices" im heterogenen Umfeld. Die Basis für diese Beschreibung bildet der 1997 veröffentlichte Artikel "Forget the Net" von K. Jonas [1]. KISS bildet eine Schnittstelle für Multimedia-Anwendungen und ist verantwortlich für die Bekanntgabe verfügbarer Multimediaströme sowie die Organisation im Netzwerk. Anhand eines Beispiels sollen zunächst die Merkmale des KISS-Konzepts aufgezeigt werden.

3.1 Ein Beispiel-Szenario

Die "Kommunikations-Infrastruktur für Streaming-Services" (KISS) besteht aus den folgenden drei Teilen:

- **Network Access Point (NAP)**
  Der NAP organisiert das interne KISS-Netz und bildet einen Zugangspunkt für die End Applications
- **Service Agent (SA)**
  Ein SA bietet zusätzliche inhaltsunabhängige Dienste wie z.B. die Umsetzung von Datenformaten (Transcoding, etc.) im KISS-Verbund an.
- **KISS-API zur Kommunikation mit End Applications (EA)**
  Das KISS-API stellt das Programmier-Interface für EAs dar.

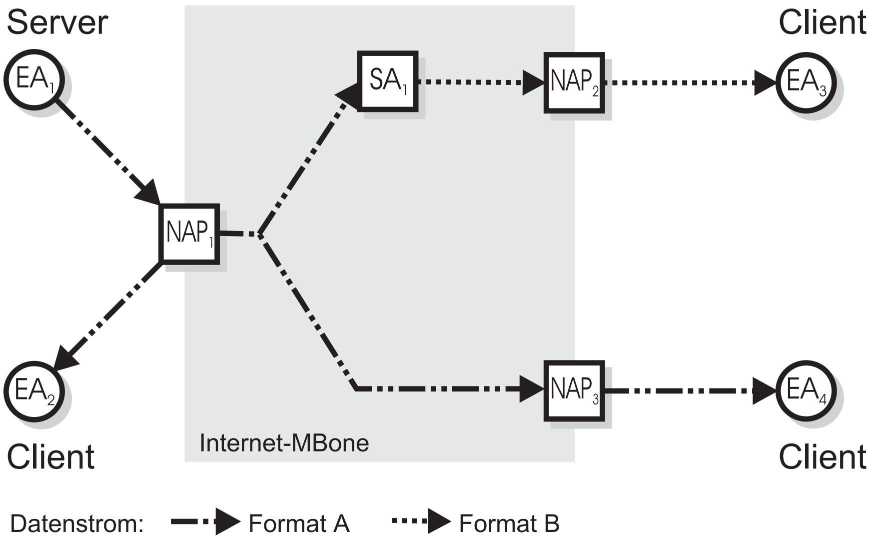

Abbildung 3.1: Beispiel Szenario

Die Ausgangskonfiguration des Beispiels (vgl. Abbildung 3.1) besteht aus drei NAPs und einem SA, die darauf warten, Anfragen von Endapplikationen zu beantworten. Keine der Komponenten weiß zu diesem Zeitpunkt etwas von der Existenz der übrigen. Möchte der Server EA_1 einen Service anbieten, muß er zunächst einen NAP (in diesem Falle NAP_1) auffinden. Danach bietet er diesem NAP seinen Service (in diesem Beispiel "KISS FM") an. Akzeptiert der NAP den Service, so übernimmt er die Bekanntgabe per Multicast im Netzwerk. Zu diesem Zeitpunkt werden noch keine Multimediadaten verschickt.

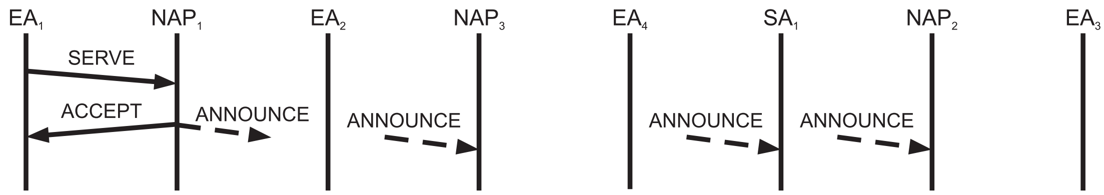

Abbildung 3.2: Beispiel-Szenario Schritt 1

Client EA_2 möchte einen Multimediastrom anfordern und wendet sich an den NAP_1, da dieser der Nächstgelegene ist. Eine Anfrage nach angebotenen Services wird mit dem Service "KISS FM" beantwortet. Daraufhin fordert der Client EA_2 nun den Service "KISS FM" an. Aufgrund der Anforderung von Client EA_2 wird der Server EA_1 vom NAP_1 benachrichtigt, worauf der Server mit der Übertragung der Multimediadaten beginnen kann. Dem Server ist dabei nicht bekannt, wer seinen Service "KISS FM" angefordert hat.

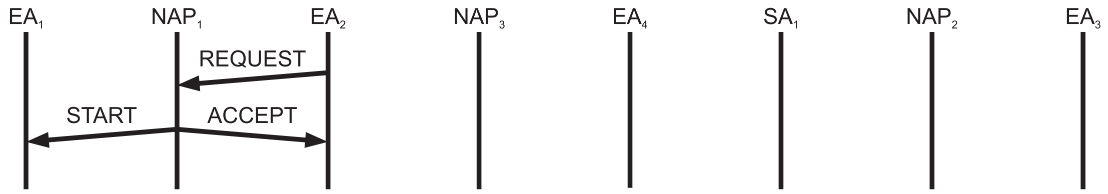

Abbildung 3.3: Beispiel-Szenario Schritt 2

Ein weiterer Client EA_4 findet als Zugangspunkt den NAP_3. Ebenso wie Client EA_2 fordert dieser eine Liste der angebotenen Services an. Der NAP_3 hat über die Multicast-Bekanntgabemechanismen den Service "KISS FM" vom NAP_1 registriert und schickt diesen als Beantwortung an Client EA_4. Fordert nun der Client EA_4 den Multimediastrom vom NAP_3 an, so fordert dieser den Multimediastrom vom NAP_1 an und wartet auf eine Antwort, die an den Clients weitergegeben wird. Wurde die Anforderung akzeptiert, so wird der Multimediastrom verschickt.

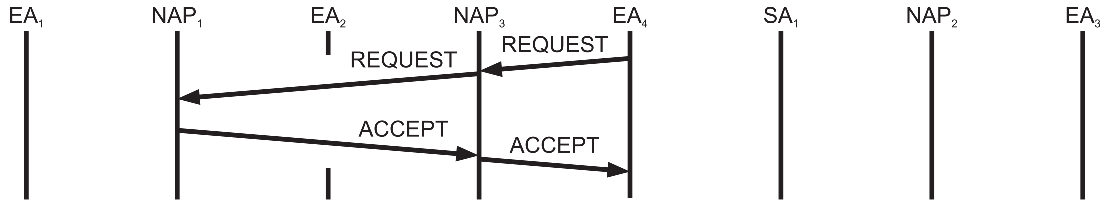

Abbildung 3.4: Beispiel-Szenario Schritt 3

Zuletzt wird der Client EA_3 gestartet. Als Zugangspunkt findet er den NAP_2 und fordert von ihm die Liste angebotener Services an. Auch dieser NAP_2 hat bereits den angebotenen Service von NAP_1 registriert und beantwortet die Anfrage. Da der Client EA_3 nur das Format Y unterstützt, fordert er den Service "KISS FM" in den von ihm unterstützten Format Y von NAP_2 an. Dieser erkennt, daß der angeforderte Service nicht in dem gewünschten Format Y vorliegt und sendet eine Anfrage per Multicast in das KISS-Netz. Diese Anfrage erhalten alle NAPs sowie alle SAs, innerhalb eines begrenzten Bereiches. Sofern sie den angeforderten Service bedienen können, beantworten sie die Anforderung. In diesem Fall kann der SA_1 die Anfrage bedienen, da er den bekanntgegebenen Multimediastrom im Format X vom NAP_1 bereits registriert hat und einen Transcodierungdienst von Format X nach Format Y anbietet. Der NAP_2 fordert daraufhin den Multimediastrom von Transcoder SA_1 an, worauf dieser den Originalstrom beim NAP_1 anfordert. In diesem Fall bekommt der Transcoder SA_1 die Multicastadresse mitgeteilt auf dem der Service “KISS FM” bereits gesendet wird. Er bestätigt die Anfrage vom NAP_2 und dieser reicht die Bestätigung an den Client EA_3 durch.

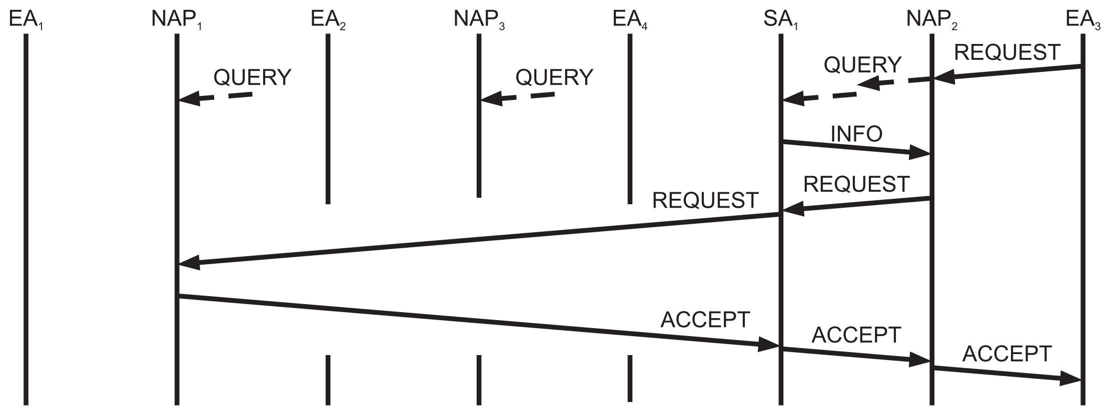

Abbildung 3.5: Beispiel-Szenario Schritt 4

Die nun erreichte Konfiguration der Multimediaströme entspricht der Abbildung 3.1. Aufbauend auf diesem Grundwissen wird im Folgenden das hinter KISS stehende Konzept dargestellt.

3.2 Die Architektur

KISS besteht aus einem losen Netz von NAPs, die über ein geeignetes Protokoll miteinander kommunizieren und den Zugangspunkt für Endapplikationen bilden. Dabei wird Wert darauf gelegt, daß möglichst viele Benutzer bedient werden können, ohne die Netzwerk-Resourcen unnötig zu belasten, um dadurch eine möglichst hohe Skalierbarkeit zu erreichen. Sowohl ein Client als auch ein Server im klassischen Sinne stellen eine Endapplikation dar. Ein NAP, der möglichst im lokalen Netz zu finden sein sollte, hat folgende Aufgaben:

- Kommunikation mit EAs und weiteren NAPs
- Bekanntgabe von vorhandenen Multimediaströmen
- Organisation und Überwachung von Multimediaströmen
- Service Agent Dienste anfordern

Typischerweise wird ein Streaming-Service in Form einer abstrakten Service-Beschreibung (service description, SD) von einem externen Service Anbieter (EA) bei einem NAP angemeldet und von diesem, gemäß der Vorgaben in der SD, den anderen NAPs mitgeteilt. Ein Client (EA) kann nun Informationen aller verfügbaren SDs bei einem NAP erfragen und die darin beschriebenen Multimediaströme anfordern. Das KISS-API stellt dabei den EAs Methoden zur automatischen Lokalisierung eines NAPs zur Verfügung, so daß es keiner Vorgabe des Benutzers bedarf. Des weiteren existieren Methoden zum Anbieten bzw. Anfordern von Multimediaströmen.

Die Verbindung zwischen EA und NAP besteht aus einer bidirektionalen Kontrollverbindung und beliebig vielen Datenverbindungen, die in der Regel unidirektional ausgeführt sind. Die Kontrollkommunikation wird im "client-network-inferface" (CNI, Abbildung 3.6) zusammengefaßt, in dem auch das Auffinden eines NAP beschrieben wird. Zu beachten ist, daß weder Server noch Client Informationen über die Organisation und Aktivitäten des Netzwerks benötigen. Hinter einem NAP wird das CNI zum "network-network-interface" (NNI, Abbildung 3.6) erweitert, welches Methoden wie das Erfragen und Bekanntgeben von Serviceinformationen hinzufügt. Diese Informationen werden vom NAP benötigt, wenn ein Client einen Multimediastrom anfordert. Grundsätzlich werden Multimediaströme per Multicast verschickt, sie sollten jedoch

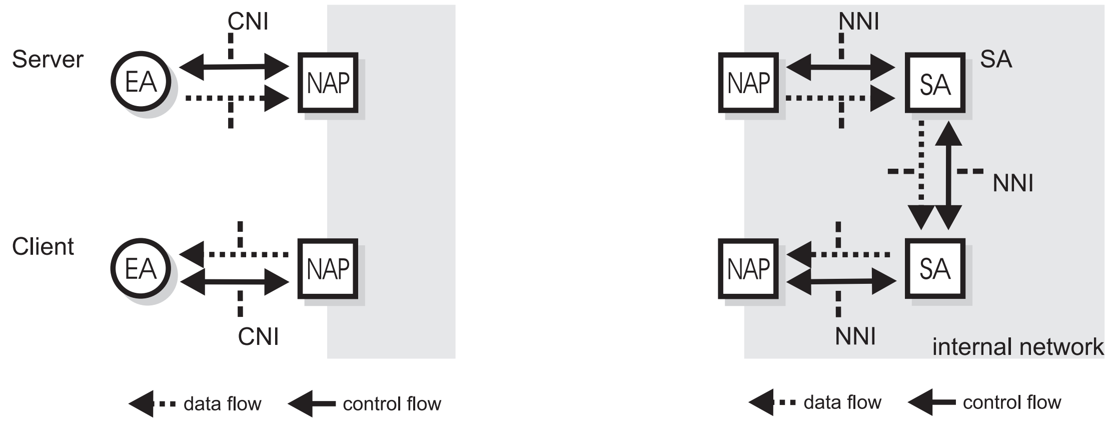

Abbildung 3.6: CNI und NNI

auch als Unicast-Strom bei einem NAP anforderbar sein, falls ein EA nicht an einem multicast-fähigen Netzwerk angeschlossen ist.

Als dritte Komponente von KISS stellen die SAs den NAPs Dienste zur Verfügung, die diese - für die EAs transparent - anfordern können. Diese Dienste bilden die eigentliche Erweiterung des Serviceangebotes von KISS. So kann ein SA z.B. die Datenrate eines Multimediastromes an die dem Client zur Verfügung stehende Bandbreite anpassen oder einen Multimediastrom von Coding-Format X in das Coding-Format Y umrechnen. Weitere Anwendungsmöglichkeiten wären das Tunneln von Firewalls, die Verbindung mehrerer KISS-"Wolken", die Synchronisation mehrerer Ströme, ein Zusammenmischen mehrerer Multimediaströme zu einem einzigen, Forward-Error-Correction (FEC), oder eine Unicast-Multicast-Umsetzung. SAs können beliebig verkettet werden, um komplizierte Szenarien zu realisieren zu. Hierzu zählt u.a. das Tunneln von Firewalls oder das Verbinden mehrerer KISS-Wolken.

Multimediaströme sollten nach Möglichkeit per Multicast verschickt werden, um die Netzwerk-Resourcen zu schonen. Innerhalb des KISS-Netzs (zwischen NAPs/SAs) wird ausschließlich Multicast verwendet. Sind die EAs an einem multicast-fähigen Netzwerk angeschlossen, sollten sie ebenfalls per Multicast bedient werden. Ansonsten versorgt der NAP sie mit einem dedizierten Unicast-Strom.

3.3 Beschreibung eines Streaming-Dienstes

Die Beschreibung eines Multimediastromes setzt sich aus zwei Bestandteilen zusammen. Zum einen aus den inhaltsbezogenen Informationen für den Benutzer und zum anderen aus der Definition der Übertragungs- und Darstellungsparameter (Präsentationsformat). Der Benutzer erwartet eine textuelle Beschreibung, die den Inhalt des Stromes erklärt. Das Streaming-System benötigt die Eigenschaften des Multimediastromes, wie das Coding-Format, die Datenrate und das zu verwendende Netzwerkprotokoll. Des weiteren können mehrere verschiedene Multimediaströme existieren, die zusammengehörende Inhalte übertragen (z.B. Audio und Video eines Fernsehsenders). Diese sollten bei der Beschreibung zusammengefaßt werden können. Somit ergeben sich mehrere Aspekte der Beschreibung:

- Eine allgemeine Beschreibung für einen Dienst (z.B. “KISS FM”) mit zusätzlichen Parametern, wie z.B. der Bekanntgabereichweite oder dem Bekanntgabezeitraum im KISS-Netz (im Folgenden *Service* genannt)
- Eine Eingrenzung des Inhalts (z.B. “AUDIO:englisch” oder “VIDEO:color”, im Folgenden *Content* genannt)
- Technische Daten (z.B. MPEG-II Layer 3 Audio, 56 kBits, RTP/UDP im Folgenden *Präsentationsformat* genannt)

Wird nun ein Service mit mehreren Multimediaströmen angeboten, so wird zunächst eine allgemeine Beschreibung benötigt. Hierzu wird eine sogenannte “*Service Description*” (SD) definiert, sie besteht aus einem *Service* und einer Liste von möglichen *Contents*. Die SD ist also eine abstrakte Beschreibung zusammengehörender Multimediaströme, die noch nicht auf das *Präsentationsformat* eingeht. Die Abstraktion geht soweit, daß ein Server eine SD anbieten kann, ohne einen dazugehörenden Multimediastrom anzubieten.

Um nun einen Multimediastrom mit seinen technischen Daten einem *Service* und einem *Content* zuzuordnen wird eine weitere Beschreibung benötigt, die sogenannte “*Data Description*” (DD). Eine DD enthält den *Service*, einen *Content* und das *Präsentationsformat*. Die Zuordnung zwischen DD und SD erfolgt über die Datenstrukturen *Service* und *Content*. Diese müssen zuerst in der SD definiert werden, damit ein NAP entsprechende DDs der jeweiligen SD zuordnen kann (Abbildung 3.7).

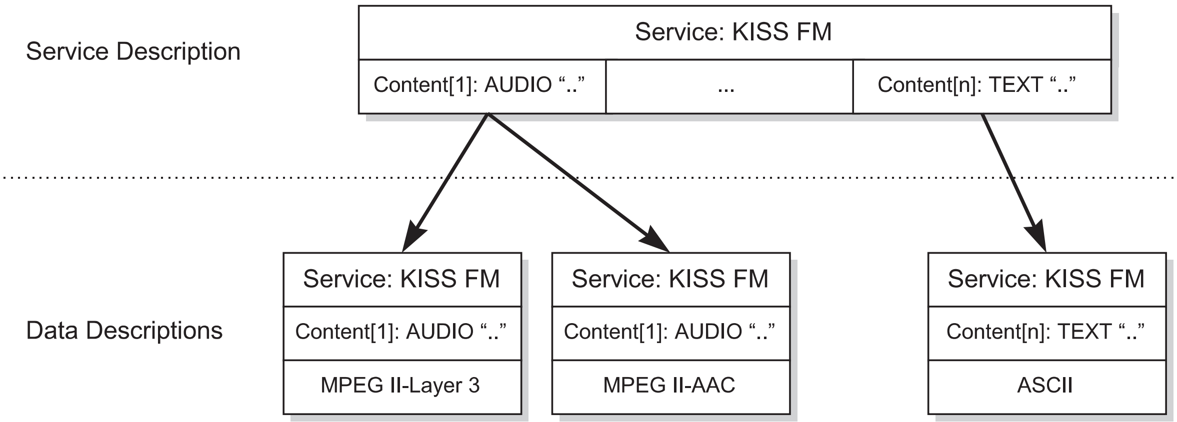

Abbildung 3.7: Zuordung von Data Descriptions zu Service Descriptions

3.4 Kommunikationsmechanismen

In den folgenden Unterpunkten soll ein Überblick über mögliche Kommunikationsabläufe zwischen Endapplikationen und NAP aufgezeigt werden, die in einem geeigneten Protokoll spezifiziert werden sollten.

3.4.1 Auffinden eines Netzwerkzugangs (NAP)

Das Auffinden eines Netzwerkzugangs für EAs erfolgt mit Hilfe von Multicast. Die EA versendet hierbei ein Anfrage-Paket per Multicast mit einer beliebigen TTL (Time To Live). Alle erreichten NAPs, die eine Anfrage bedienen können, antworten daraufhin per Unicast. Die EA kann dann anhand der Absenderadresse Kontakt zu einem NAP aufnehmen. Optional kann bereits bei dem Anfrage-Paket ein Servicename mitgeschickt werden, so daß nur die NAPs antworten, die diesen Service kennen.

3.4.2 Anfordern des Serviceverzeichnisses

Mit Hilfe eines speziellen "ANFRAGEN"-Paketes, kann eine Liste aller bekannten SDs (SD-QUERY) oder DDs (DD-QUERY) vom NAP angefordert werden. Optional kann bei einem SD-QUERY-Paket ein Servicename mitgeschickt werden, bei einem DD-QUERY-Paket eine DD. Dadurch können Anfragen eingegrenzt werden.

3.4.3 Anbieten von Servicebeschreibungen (SD)

Das Anbieten einer Servicebeschreibung geschieht mit einem "ANBIETEN"-Paket, das an einen NAP gesendet wird. Hierbei werden in der SD neben dem Servicenamen auch Mechanismen wie Bekanntgabezeitraum (z.B. täglich), Bekanntgabeintervall oder Bekanntgaberadius (z.B. LAN, MAN, WAN) festgelegt. Kann der NAP den Service nicht bedienen, wird dieser abgelehnt. Gründe für eine Ablehnung könnten eine Überlastung des NAP sein oder daß ein Service mit gleichlautendem Namen bereits existiert. Akzeptiert der NAP den Service, so bestätigt er dieses der EA und übernimmt nach den Vorgaben der EA (Server) die Bekanntgabe im KISS-Netz.

3.4.4 Anbieten und Anfordern von Multimediaströmen

Um einen Multimediastrom anbieten zu können, muß zunächst eine DD erzeugt werden, in der die Eigenschaften des Stromes eingetragen werden. Des weiteren müssen *Service* und *Content* aus einer existierenden SD übernommen werden. Diese DD wird dem NAP angeboten, und entsprechend bestätigt oder abgelehnt. Gründe für eine Ablehnung könnten sein, daß dem NAP keine passende SD bekannt ist, bereits eine gleichlautende DD installiert ist, oder daß der NAP überlastet ist. Akzeptiert der NAP den Multimediastrom, so wird eine Datenverbindung vereinbart und der NAP übernimmt die Bekanntgabe im KISS-Netz.

Soll ein Multimediastrom angefordert werden, so muß eine entsprechende DD als Anforderung an den NAP geschickt werden. Ist der dazugehörige Multimediastrom dem NAP bekannt, so bestätigt er die Anforderung und organisiert die Datenverbindung. Ist dieser Multimediastrom dem NAP zwar bekannt, liegt aber in einem anderen *Präsentationsformat* vor, so stellt der NAP eine Anfrage in das KISS-Netz und wartet, ob diese von einem SA, der das angeforderte *Präsentationsformat* generieren kann, beantwortet wird. Empfängt der NAP innerhalb eines Zeitrahmens keine Antwort, so wird die Anforderung der EA abgelehnt.

3.4.5 Verbindungsüberwachung/-abbau

Jede Komponente (EA, NAP und SA) sollte geeignete Mechanismen zur Überwachung der Datenverbindungen bereitstellen, da nicht auszuschließen ist, daß sich die eine oder andere Applikation "kommentarlos" beendet oder durch Netzwerkprobleme Verbindungen unterbrochen werden. In jedem Fall sollte sich eine Komponente bei ihren "Nachbarn" ordnungsgemäß abmelden.

3.4.6 Authentifizierung und Abrechnung

Im Zuge einer kommerziellen Nutzung sind die Aspekte Authentifizierung und Abrechnung von großer Bedeutung, wenn z.B. Live-Konzerte o.ä. übertragen werden sollen. Eine Spezifikation entsprechender Dienste würde allerdings über den Rahmen dieser Arbeit hinausgehen.

3.5 Zusammenfassung

Das in diesem Kapitel vorgestellte Konzept erweitert die Funktionalität der im Zusammenhang mit RTP angedachten Translator-/Mixerdienste um die Fähigkeit Multimediasessions dynamisch auf- und abzubauen, und unterstützt eine flexibele Konfiguration der Service Agents. Weiter ermöglicht es eine automatische Lokalisierung von Anbietern und damit eine einfachere Handhabung für Benutzer.

Nachdem nun allgemein die Funktionalität von Service Agents beschrieben wurde, wird im folgenden Kapitel ein spezieller Service Agent, der sogenannte Audio-Transcoder, vorgestellt. Dieser gestattet es, Audiostöme in Echtzeit vom Präsentationsformat X in das Präsentationsformat Y umzurechnen (transcodieren).

Kapitel 4

Der Audio-Transcoder als KISS Service Agent

In diesem Kapitel wird ein Audio-Transcoder für RTP-Ströme [4] vorgestellt, der einem KISS-Netz seine Dienste als *Service Agent* zur Verfügung stellt. Dabei stellt der Audio-Transcoder im RTP-Kontext [4] einen Translator dar.

4.1 Was ist ein Audio-Transcoder ?

Ein Audio-Transcoder ermöglicht es, ein Audiosignal von einer Kodierung in eine andere umzurechnen. Er wird in einem heterogenen Umfeld verwendet, wenn z.B. zwei Teilnehmer miteinander kommunizieren möchten, dies aufgrund unterschiedlicher Audiokodierungsverfahren aber nicht können. Man denke hier z.B. an ein Telefongespräch zwischen dem digitalen ISDN-Netz oder einem der Mobilfunknetze und dem analogen Telefonnetz. Eine weitere Einsatzmöglichkeit ist die Datenratenanpassung eines Audiostromes, wenn z.B. den Teilnehmern einer Telekonferenz unterschiedliche Bandbreiten zur Verfügung stehen. Der hier beschriebene Audio-Transcoder unterstützt sowohl die Datenraten- als auch der Kodierungsformat-Umsetzung für Realtime-Anwendungen in Datennetzen.

4.2 Einsatz-Szenarien unter KISS

Der Transcoder soll drei Szenarien in einem KISS-Netz ermöglichen. Erstens einen Bitratenumsetzer, der es ermöglicht, den selben RTP-Strom mit verschiedenen Bitraten in jeweils einer eigenen Multicastgruppe anzubieten und zweitens einen Kodierungsformatumsetzer um den selben Strom parallel in verschiedenen Kodierungen anbieten können. Diese beiden Szenarien zeigen die zwei Grundfunktionen des Audio-Transcoders. Es ist auch möglich, diese zu kombinieren, um sowohl eine Bitraten- als auch eine Kodierungsformat-Umsetzung in einem Transcodier-Vorgang vorzunehmen. Diese Möglichkeit wird für das dritte Szenario benötigt. Hier werden zwei verkettete Transcoder verwendet, um das Tunneln einer schmalbandigen Netzverbindung (wie etwa dem Internet) zu ermöglichen.

4.2.1 Bitratenanpassung mit einem Transcoder

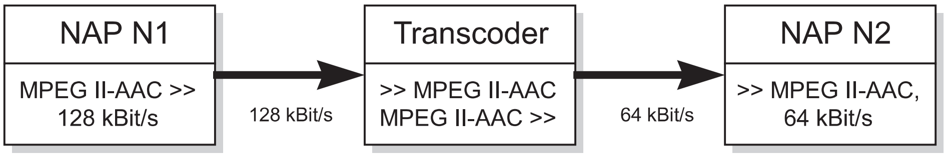

Abbildung 4.1: Bitratenanpassung mit einem Transcoder

Der Transcoder ist so konfiguriert, daß er RTP-Eingangsströme beliebiger Bitrate eines Coding-Formates (in Abbildung 4.1 MPEG II-AAC) in RTP-Ausgangsströme desselben Coding-Formates mit einer anderen[1] Bitrate umzurechnen (transcodieren) vermag. Beide Ströme werden parallel in verschiedenen Multicastgruppen angeboten, so daß ein Teilnehmer jeweils mit dem für ihn passenden Strom bedient werden kann.

4.2.2 Kodierungsformat-Umsetzung mit einem Transcoder

Der Transcoder ist so konfiguriert, daß er RTP-Eingangsströme eines Coding-Formates (in Abbildung 4.2 MPEG II-AAC) in RTP-Ausgangsströme eines anderen Coding-Formates (in Abbildung 4.2 MPEG II-Layer 3) mit derselben Bitrate umzurechnen (transcodieren) vermag. Beide Ströme werden ebenfalls parallel in verschiedenen Mul-

[1] in der Regel niedrigeren

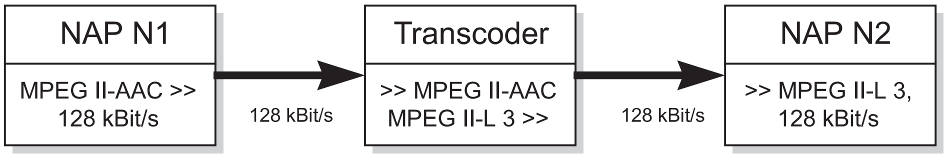

Abbildung 4.2: Kodierungsformat-Umsetzung mit einem Transcoder

ticastgruppen angeboten, so daß ein Teilnehmer auch hier mit dem für ihn passenden Strom bedient werden kann.

4.2.3 Der Transcoder als Internet-Tunnel

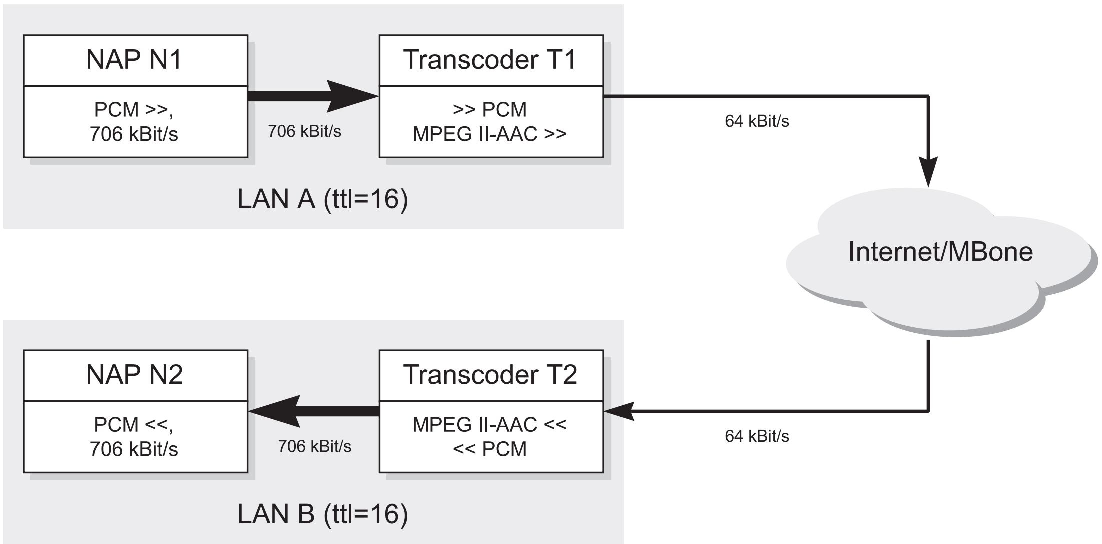

Abbildung 4.3: Der Transcoder als Internet-Tunnel

Die Transcoder sind wie in Abbildung 4.3 dargestellt, konfiguriert. Der Aufbau der Datenverbindungen für die Multimediaströme erfolgt "rückwärts". Der *Netwok Access Point* (NAP) N_2 fordert einen Datenstrom im PCM-Format mit 706 kBit/s an, der im lokalen Netz nicht in diesem Format verfügbar ist. Der Transcoder T_2 kann diesen Strom erzeugen, sofern ihm jemand den Strom im MPEG II-AAC Format anbietet und stellt daher eine entsprechende Anfrage ins KISS-Netz. Der Transcoder T_1 wiederum kann diesen Strom im MPEG II-AAC Format liefern und benötigt hierfür ein Eingangsstrom im PCM Format. Diesen kann der NAP N_1 anbieten. Nachdem die Datenverbindungen nun ausgehandelt und aufgebaut sind, wird der Multimediastrom, wie in Abbildung 4.3 dargestellt, transcodiert und wird vom NAP N_2 empfangen. Die Bitrate, die zwischen den Transcodern T_1 und T_2 verwendet wird, hängt von der "Entfernung" der Transcoder, gemessen in Multicast-TTLs (Time To Live), ab. Je größer die benötigte TTL ist, desto geringer ist die verwendete Bitrate.

4.3 RTP Konformität und die KISS-Erweiterung

Da der hier vorgestellte Audio-Transcoder auf dem Konzept des in Abschnitt 2.3.1 beschriebenen RTP-Translators aufbaut, unterstützt er die beiden Hauptanforderungen wie folgt: Die *Sychronisations Source* (SSRC) vom jeweiligen Eingangsstrom wird auch für den neuen Ausgangsstrom verwendet, so daß die ursprüngliche Quelle des Stromes auch weiterhin identifiziert werden kann, obwohl sich die Absender-Netzwerkadresse geändert hat, da der transcodierte Strom vom Transcoder und nicht mehr vom originalen Server gesendet wird. Gemäß der RTP Spezifikation könnte der *Timestamp*-Zähler (TS) des Ausgangsstromes mit dem des Eingangsstromes abgeglichen, oder mit einem Zufallswert neu initialisiert werden. Der in diesem Kapitel beschriebene Audio-Transcoder unterstützt die Synchronisation der Timestamp-Zähler, denn dadurch ist es möglich, daß ein Client beliebig zwischen beiden Audioströmen wechseln kann, ohne daß "Aussetzer" oder "Sprünge" auftreten.

Die Einbettung des Audio-Transcoders in das KISS-Szenario, und die damit verbundene Implementierung des Network-Network-Interface (NNI) erweitert den im RTP-Kontext[4] beschriebenen Translator um die Möglichkeit, ihn zur Laufzeit lokalisieren und konfigurieren zu können. Der Audio-Transcoder muß also nicht beim Aufsetzen mit fest vorgegebenen Parametern gestartet werden. Er reagiert vielmehr auf Transcodierungs-Anforderungen des KISS-Netzes und kann sich somit dynamisch den jeweiligen Anforderungen (Bitratenanpassung, Internet-Tunnel, etc.) anpassen. Es ist daher möglich, dem Benutzer - für diesen völlig transparent - einen Datenstrom anzubieten, der für seinen Netzzugang optimal angepaßt ist.

Kapitel 5

Entwurf des Audio-Transcoders

Der Audio-Transcoder besteht aus einem *Controller*, der über das *Network-Network-Interface* (NNI) mit dem KISS-Netz kommuniziert und die Datenverbindung für den Transcoder aushandelt, sowie einer Hintereinanderschaltung eines eingangsseitigen Audiodecoders und eines ausgangsseitigen Audioencoders, dem *Transcoder-Modul.* Als Transportprotokoll für die Audioströme wird das “Realtime Transport Protocol” (RTP) in Verbindung mit dem UDP/IP-Multicast-Protokoll verwendet. Der Audiodecoder empfängt jeweils einen RTP-Strom aus einer Multicastgruppe im Präsentationsformat X, decodiert diesen in einen PCM-Strom und übergibt dem Encoder die PCM-Parameter wie Anzahl der Bits pro Sample, Anzahl der Kanäle sowie die Abtastrate. Der Encoder encodiert diesen PCM-Strom dann jeweils in einen *Bitstream* des Präsentationsformats Y, erzeugt einen neuen RTP-Header und sendet diese Pakete in eine neue Multicastgruppe (Abbildung 5.1).

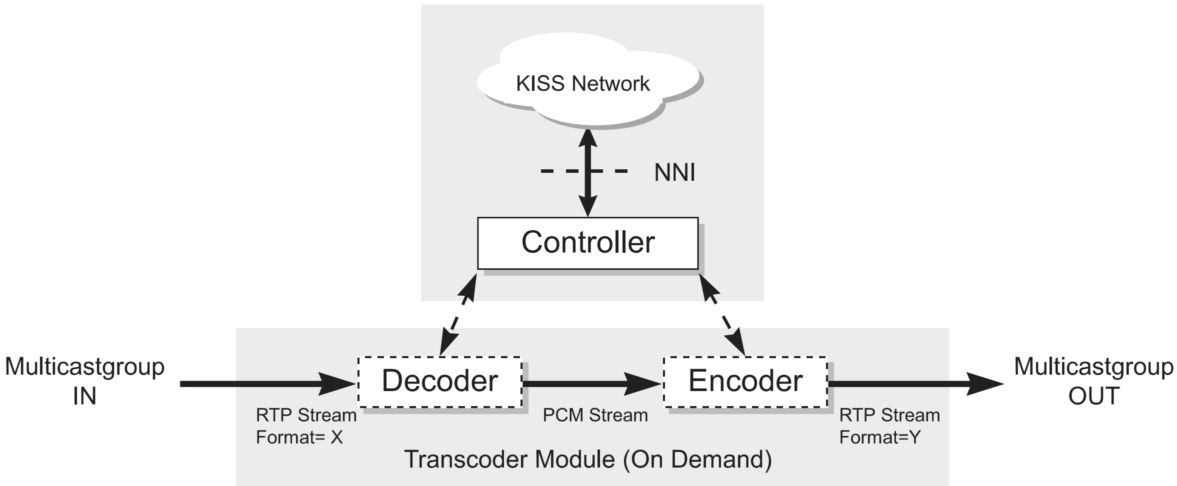

Abbildung 5.1: Aufbau des Transcoders

5.1 Der Controller

Der *Controller* kommuniziert über das NNI mit dem KISS-Netz. Seine Aufgabe ist die Bearbeitung von Transcodierungs-Anforderungen, der entsprechende Auf- und Abbau von Datenverbindungen (Datalink) sowie die Kontrolle (Erzeugen, Überwachen und Terminieren) der jeweiligen Transcoder-Module. Der Audio-Transcoder besteht immer aus einem *Controller*, der eine beliebige Anzahl von Transcoder-Modulen steuern kann. Als Kommunikationsprotokoll innerhalb des KISS-Netzes wird das von Jens Mödeker im Rahmen seiner Diplomarbeit [21] entwickelte "Service Management Protocol" (SMP) verwendet. Für eine detaillierte Beschreibung seiner Funktionalität wird daher auf diese Arbeit verwiesen.

5.1.1 Das Service Management Protocol (SMP)

Das "Service Management Protocol" (SMP) basiert auf UDP und IP-Multicast. Jeder NAP oder SA[1] empfängt von einem DATAGRAM-Socket SMP-Pakete, die an eine fest definierte IP-Multicastgruppe oder an ihn direkt adressiert sind. Ein SMP-Paket besteht aus einem Header in dem u.a. der Pakettyp vermerkt ist, sowie aus einer variablen Liste zusätzlicher Informationen, die je nach Pakettyp benötigt werden. Hierzu gehören z.B. Service Descriptions, Data Descriptions oder Adressinformationen für den Aufbau von Datenverbindungen. Für die Kommunikationsabläufe des Audio-Transcoders werden nur Data Descriptions und Adressinformationen benötigt. Im Folgenden werden die Pakettypen vorgestellt, die für den Audio-Transcoder relevant sind:

- `DL_ANNOUNCE`
  Ein NAP announciert eine Data Description. Aufgrund dieser Information kann eine Datenverbindung in dem bekanntgegebenen Format zu diesem NAP aufgebaut werden.

- `DL_QUERY`
  Ein NAP oder SA sucht eine Data Description um daraufhin eine Datenverbindung aufzubauen. Das Präsentationsformat kann genau spezifiziert sein, oder auch Wildcards (z.B für "Payload Type" bzw. Datenrate) enthalten.

- `DL_INFO`
  Ein NAP oder SA beantwortet eine `DL_QUERY`-Anforderung mit einem `DL_INFO`-Paket, sofern er die angeforderte Data Description anbieten kann. Dabei teilt er dem Empfänger mit, in welchem Präsentationsformat dieser die entsprechende Datenverbindung anfordern kann.

- `DL_CLIENT`
  Ein NAP oder SA fordert eine Datenverbindung von einem NAP oder SA an. Das Präsentationsformat muß eindeutig angegeben werden und darf keine Wildcards enthalten.

- `DL_ACCEPT`
  Ein NAP oder SA akzeptiert eine `DL_CLIENT`-Anforderung und sendet von nun an den Multimediastrom zum jeweiligen Empfänger.

[1] in diesem Abschnitt der Audio-Transcoder

- `DL_REJECT`
  Ein NAP oder SA lehnt eine `DL_CLIENT`-Anforderung ab, weil er z.B. überlastet ist, oder er den angeforderten Strom nicht (mehr) anbietet.

- `DL_RELEASE`
  Ein Empfänger (NAP oder SA) benötigt die Datenverbindung nicht mehr, und meldet sich beim jeweiligen Sender (NAP oder SA) ab. Auf die gleiche Weise kann ein Sender dem Empfänger mitteilen, das die Datenverbindung nicht länger existiert.

- `DL_PAUSE`
  Ein Empfänger (NAP oder SA) möchte kurzzeitig keinen Datenstrom empfangen, sich aber beim Sender (NAP oder SA) nicht vollkommen abmelden.

- `DL_CONTINUE`
  Ein mit `DL_PAUSE` angehaltener Datenstrom wird wieder fortgesetzt.

- `DL_PING, DL_ALIVE`
  Alle Sender einer Datenverbindung sollten in festen Zeitabständen mit einem `DL_PING`-Paket überprüfen, ob die jeweiligen Empfänger-Applikationen ihren Datenstrom noch empfangen. In diesem Fall muß der Client mit einem `DL_ALIVE`-Paket antworten. Ein Empfänger wiederum kann ebenfalls mit Hilfe eines `DL_PING`-Paketes feststellen, ob der Sender der Datenverbindung noch existiert (z.B. wenn sich die Datenverbindung im "PAUSE"-Modus befindet).

Typische Kommunikationsabläufe des Audio-Transcoders für das bereits in Zusammenhang mit Abbildung 4.1 erläuterte Szenario sind in Abbildung 5.2 dargestellt. Der Audiostrom fließt von NAP_1 über den Audio-Transcoder zu NAP_2. Der Initiator dieser Kommunikation ist der NAP_2, da der Aufbau der Datenverbindungen "rückwärts" erfolgt.

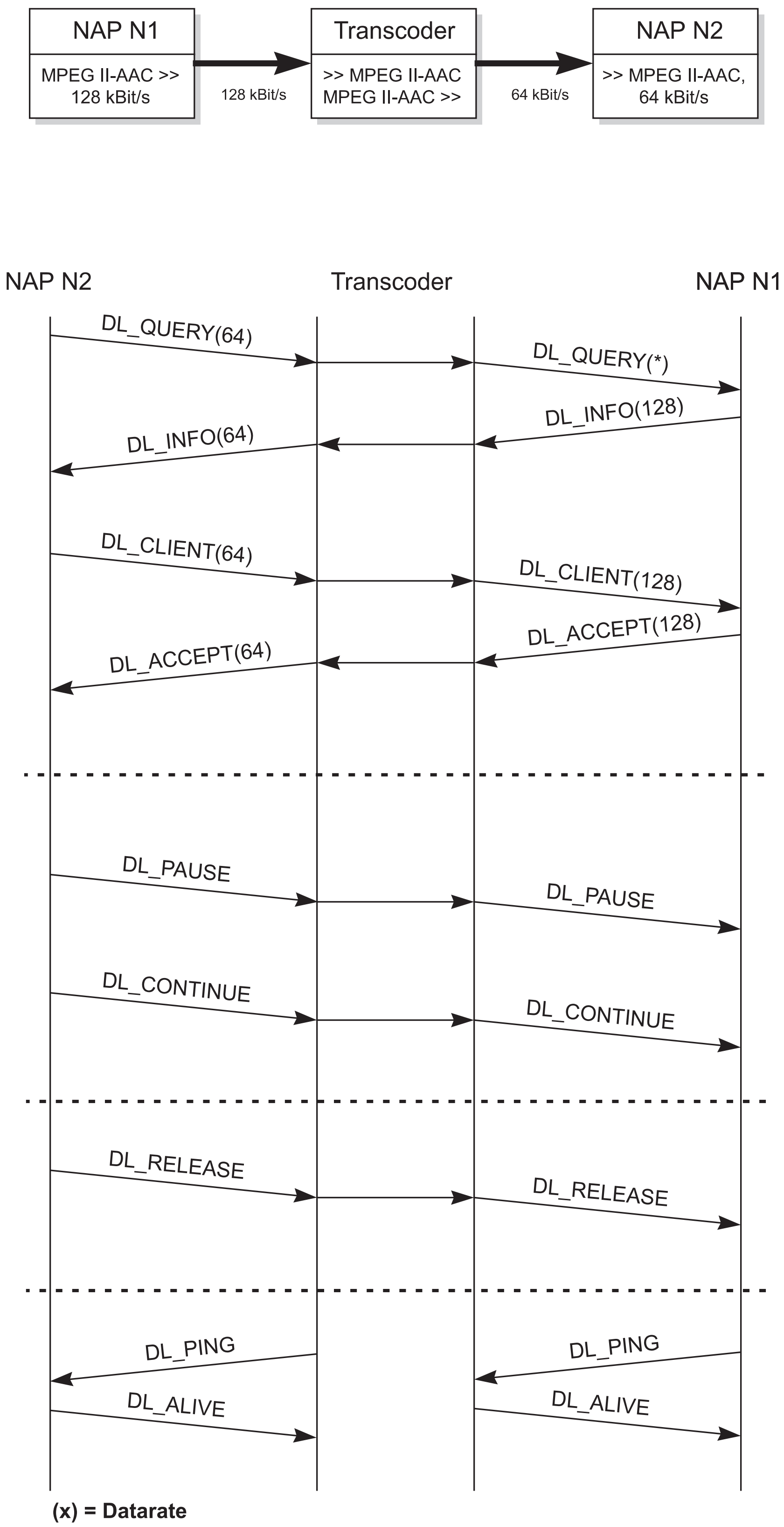

Abbildung 5.2: Typische Kommunikationsabläufe des Audio-Transcoders

5.1.2 Verkettung von Transcodern

Transcoder können beliebig verkettet werden. Dies ist z.B. nötig, wenn ein Transcoder nicht in der Lage ist, die angeforderte Transcodierung alleine vorzunehmen. Die interessantere Anwendung für eine Verkettung von Transcodern stellt allerdings die im vorherigen Kapitel beschriebene Tunnelung des Internets dar[2]. Hierzu wird vorausgesetzt, daß die Datenrate im LAN beliebig hoch sein kann, während sie im Internet mit steigender Entfernung der Kommunikationspartner mehr und mehr abnimmt. Um diesen Gegebenheiten Rechnung zu tragen, werden die maximalen Datenraten in Abhängigkeit von der IP-Multicast TTL, die benötigt wird, um den Kommunikationspartner zu erreichen, wie folgt festgelegt:

TTL	max. Bandbreite
16	keine Begrenzung
64	56 kBit/s
192	32 kBit/s

Der Transcoder ist bestrebt, einen möglichst hochwertigen (und damit auch breitbandigen) Eingangsstrom aus dem lokalen Netz zu verwenden um die Qualitätsverluste, die bei der Transcodierung entstehen, so gering wie möglich zu halten. Dies wird dadurch erreicht, daß der Transcoder der Reihe nach `DL_QUERY`-Pakete mit steigender TTL aber sinkender Maximalbandbreite ins Netz sendet (siehe Tabelle). Auf diese Weise ist sichergestellt, daß zuerst mögliche lokale NAPs oder SAs antworten. Nur wenn diese die Anfrage nicht bedienen können werden NAPs oder SAs im WAN gesucht.

In der in Abbildung 5.3 dargestellten Konstellation, gibt es keinen Anbieter(NAP oder SA) im LAN. Daher sendet der Transcoder T_1 ein `DL_QUERY`-Paket mit einer TTL von 64 und begrenzt die Datenrate damit auf 56 kBit/s. Kann der Transcoder T_2 den angeforderten Strom bedienen so beantwortet er die Anfrage. Wenn nicht sendet der Transcoder T_1 erneut ein `DL_QUERY`-Paket. Diesmal mit einer TTL von 192 und einer maximalen Bandbreite von 32 kBit/s. Sofern die Transcoder T_3 und/oder T_4 den angeforderten Strom erzeugen können, beantworten sie die Anfrage von T_1.

[2]vgl. Abbildung 4.3

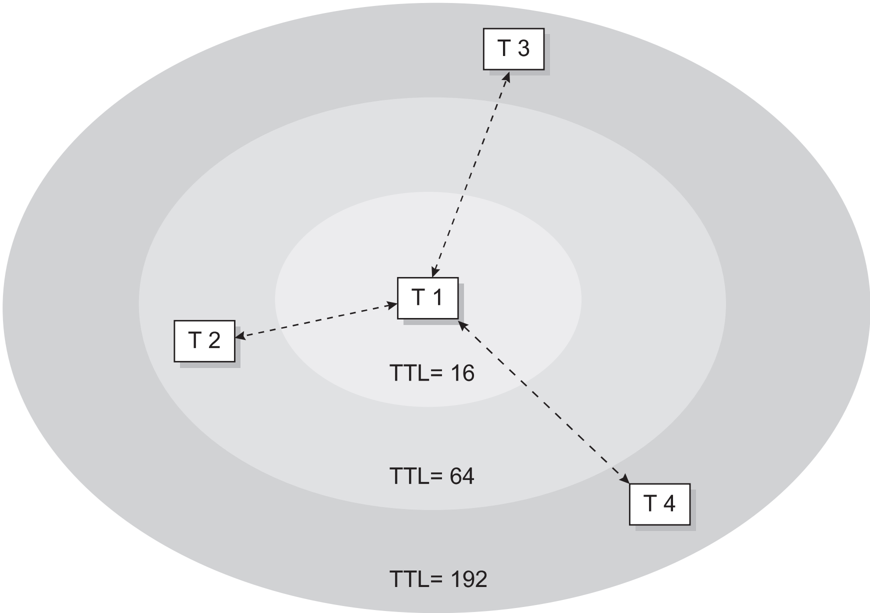

Abbildung 5.3: DL_QUERY mit steigender TTL

5.1.3 Kontrolle der Transcoder-Module

Es ist Aufgabe des *Controllers*, auf Anforderung durch ein `DL_CLIENT`-Paket die entsprechenden Datenverbindungen aufzubauen und ein Transcoder-Modul zu starten, sowie es aufgrund eines `DL_RELEASE`-Paketes zu terminieren. Treten während der Laufzeit eines Transcoder-Modules Fehler auf, so muß der *Controller* dieses Modul beenden und die anderen Kommunikationspartner mit einem `DL_RELASE`-Paket davon in Kenntnis setzen. Des weiteren obliegt es dem *Controller*, über den `DL_PING/DL_ALIVE`-Mechanismus die Überwachung der Datenverbindungen abzuwickeln.

5.2 Das Transcoder-Modul

Das Transcoder-Modul besteht im wesentlichen aus zwei Komponenten, die im Folgenden näher erläutert werden:

- einem streaming-fähigen Realtime Audiodecoder,
  der einen encodierten Audiostrom in Form von RTP-Paketen aus einer Multicastgruppe empfängt, diesen decodiert und als PCM-Strom ausgibt.
- einem streaming-fähigen Realtime Audioencoder,
  der einen PCM-Strom einliest, im entsprechenden Format encodiert und diesen in Form von RTP-Paketen in eine Multicastgruppe sendet.

Der Audio-Transcoder stellt somit eingangsseitig einen Client und ausgangsseitig einen Server dar.

5.2.1 Die generische E/A-Schnittstelle "IO Wrapper"

Sowohl der Decoder als auch der Encoder besitzen je eine Eingangs- und eine Ausgangsschnittstelle des Types *IO Wrapper*, von der sie einen Datenstrom einlesen bzw. in die sie einen Datenstrom ausgeben. Als Eingangs- bzw. Ausgangsmedium kommen z.B Dateien, Netzwerksockets (IP, ATM AAL 5, etc.), Pipes oder ein Audio Device in Frage. Zusätzlich sollte es möglich sein, einen Datenstrom auf mehrere Medien gleichzeitig auszugeben. Um dies zu ermöglichen und um den Encoder bzw. Decoder möglichst einfach an das jeweilige Medium anpassen zu können, wurde eine Abstraktionsschicht implementiert, die folgende Funktionen zur Verfügung stellt:

- `int read_pkt( char *buf, int size)`
  liest das nächste Paket der (maximalen) Länge *size* vom Eingangsmedium, speichert es im Puffer *buf* und liefert die Anzahl der gelesenen Bytes zurück.
- `int write_pkt( char *buf, int size)`
  schreibt den Puffer *buf* der Länge *size* in das Ausgabemedium und liefert die Anzahl der geschriebenen Bytes zurück.
- `bool setup_media( void *as)`
  initialisiert das Ausgabemedium. Bei einem Audio Device wird hier z.B. die Abtastrate oder die Anzahl der Kanäle, etc. eingestellt.

Für das zu verwendende Medium muß nun ein entsprechender “Treiber” implementiert werden, der die obigen drei Funktionen auf das API (Application Programming Interface) des jeweiligen Mediums abbildet (Abbildung 5.4).

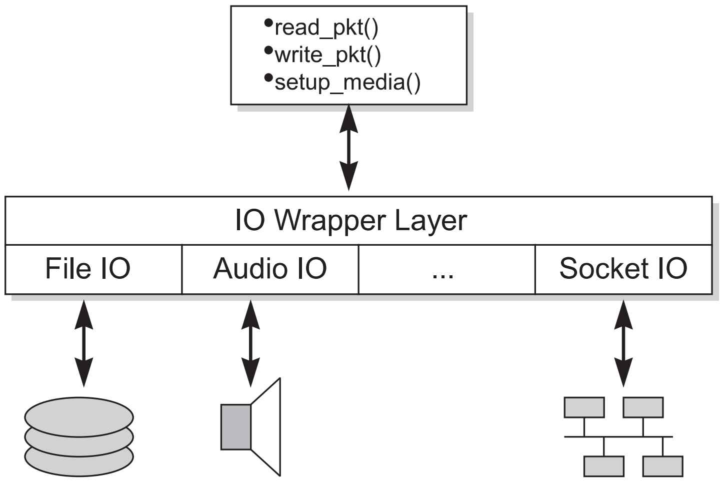

Abbildung 5.4: Die genersiche E/A Schnittstelle

5.2.2 Allgemeine Decoder-/Encoderstruktur

Decoder und Encoder sind modular und objektorientiert aufgebaut. Sie bestehen jeweils aus einem Inputmodul, einem modifizierten FIFO (MFIFO) und einem Outputmodul (Abbildung 5.5).

Abbildung 5.5: Struktur von Decoder und Encoder

Die Input- und Outputmodule werden durch die Verwendung von *Multithreading* völlig entkoppelt voneinander ausgeführt. Der für Decoder und Encoder unterschiedlich ausgeführte MFIFO bildet die einzige Kommunikationsverbindung zwischen den Input- und Outputmodulen. Diese Modularität ermöglicht eine einfache Implementierung verschiedener Audio-Codecs und Transportprotokolle, da diese in getrennten Modulen implementiert werden. Ein weiterer Vorteil ist, daß die entsprechenden Module erst zur Laufzeit ausgewählt werden müssen und sie sich beliebig kombinieren lassen.

5.2.3 Der Decoder

Das Input-Modul des Decoders enthält die Funktionalität des Transportprotokolls (z.B. RTP). Es liest RTP-Pakete über den *IO Wrapper* ein und wertet den RTP-Header aus. Dabei wird überprüft, ob das Paket von der richtigen Quelle (SSRC) kommt und ob es das erwartete Coding-Format (Payload Type) enthält. Schlägt einer dieser Tests fehl, wird das Paket verworfen, ansonsten wird das Paket inklusive seiner Sequenznummer und seines Timestamps (TS) im MFIFO gespeichert (Abbildung 5.6).

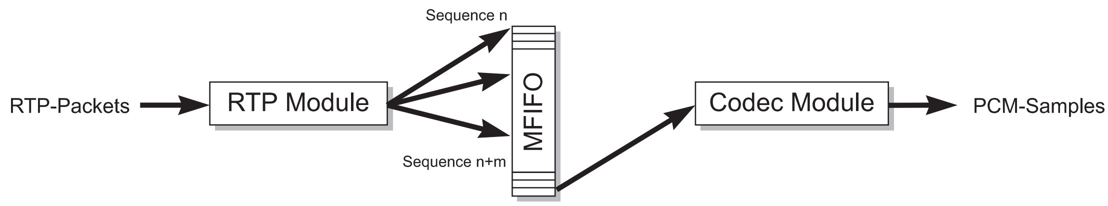

Abbildung 5.6: Der Aufbau des Decoders

Der MFIFO des Decoders funktioniert im fehlerfreien Falle wie ein normaler FIFO-Ringpuffer nach dem *First-In-First-Out*-Prinzip und seine Hauptaufgabe ist das Ausgleichen von Jitter-Effekten oder kurzzeitigen Bandbreiteschwankungen. Anhand der Sequenznummer kann das InputModul feststellen, ob ein Paket verloren gegangen, oder in der falschen Reihenfolge eingetroffen ist. In diesem Fall wird es nicht am Ende des MFIFOs eingekettet, sondern entsprechend seiner Sequenznummer einsortiert (Packet Reordering). Damit das *Packet Reordering* funktioniert, muß sichergestellt sein, daß immer eine Mindestanzahl von vorgepufferten Paketen im MFIFO vorhanden ist, um zu spät eintreffende Pakete wieder an ihrer ursprünglichen Position einsortieren zu können. Reißt der Paketstrom ab, weil in einem bestimmten Zeitraum keine oder nur ungültige Pakete eintreffen, muß der Decoder neu initialisiert werden. Abbildung 5.7 verdeutlicht die Funktion des modifizierten Ringpuffers graphisch.

Durch die Vorpufferung erhöht sich die Zeitspanne zwischen der Echtzeit und der Abspielzeit (Playout Time). Daher sollte die Anzahl der vorgepufferten Pakete an die Verbindungsqualität und damit an das Risiko, daß *Packet Disordering* auftritt, angepaßt werden. Größere Schwankungen der zur Verfügung stehenden Bandbreite bzw. *Packet Disordering* beobachtet man in der Regel nur in Weitverkehrsnetzen. Im LAN bzw. innerhalb eines IP-Subnetzes treten diese Probleme sehr selten auf. Der Decoder ermittelt während der Initialisierungsphase, ob sich der Sender des Datenstromes im

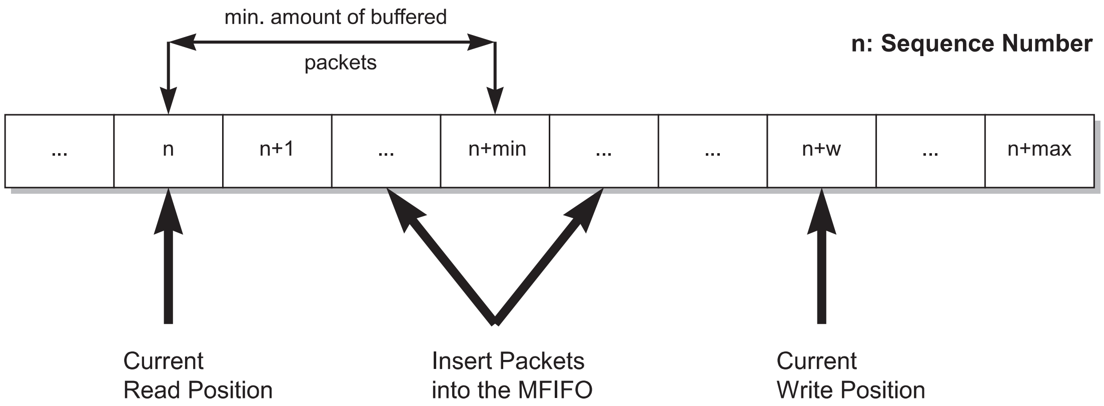

Abbildung 5.7: Funktion des modifizierten FIFO-Ringpuffers

selben IP-Subnetz befindet oder nicht und legt daraufhin die Anzahl der vorzupuffernden Pakete fest. Im Falle eines Paketverlustes liefert der MFIFO an dieser Stelle ein Paket der Länge null.

Das Output Modul enthält die Funktionalität des jeweiligen Audiodecoders. Es liest die Pakete aus dem MFIFO, decodiert diese und gibt sie über den *IO Wrapper* aus. Nachdem das Output Modul das erste Paket decodiert hat, initialisiert es zuerst das Output-Medium mit Hilfe der *IO Wrapper*-Methode `setup_media()`. Hier wird das Output-Medium (z.B. Pipe, Audio Device) konfiguriert. Wird beim Auslesen des MFIFO ein Paketverlust festgestellt, bedeutet dies, daß das entsprechende Paket und die darin enthaltene Audio-Spielzeit ersetzt werden muß, da Sender und Empfänger sonst nicht mehr synchron zueinander arbeiten. Einige moderne Audiodecoder-Implementierungen bieten die Möglichkeit ein fehlendes Paket durch "Interpolation" weitgehend zu rekonstruieren. Wenn ein Audiodecoder diese Möglichkeit nicht bietet, muß ein "leerer" PCM-Block entsprechender Länge eingefügt werden.

5.2.4 Der Encoder

Der Encoder besteht aus einem Input-Modul, das PCM-Samples in festen vom jeweiligen Audio-Codec vorgegebenen Blockgrößen über den *IO Wrapper* einliest und in einem modifizierten FIFO (MFIFO) zwischenspeichert. Der MFIFO wurde dahingehend modifiziert, daß er sich im Falle eines Überlaufs neu initialisiert und dabei die gepufferten und veralteten Daten verwirft. Echtzeit-Audioencodierung ist ein sehr rechenintensives Verfahren, das heutige Prozessoren wie z.B. einen UltraSPARC oder einen Intel Pentium im Mittel bis zu 100 % auslasten kann. In der Regel hängt die Prozessorauslastung von der Art des Eingangssignales ab. Sprache z.B. ist schwieriger zu encodieren als Musik und benötigt mehr Rechenzeit. Es kann daher vorkommen, daß ein Audioencoder einige Sekunden lang nicht mehr in Echtzeit arbeitet, also länger zum Encodieren eines PCM-Blockes benötigt als dieser Audiospielzeit enthält. Der MFIFO puffert während dieser Zeit die PCM-Samples zwischen, damit das Input-Medium (z.B. Socket, Audio Device) des Rechners nicht überläuft. Sobald der Encoder wieder weniger Rechenzeit benötigt werden die im MFIFO aufgestauten Samples abgearbeitet.

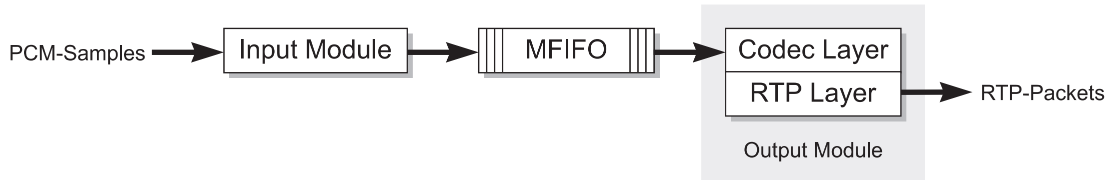

Abbildung 5.8: Der Aufbau des Encoders

Die Basisklasse des Outputmoduls enthält die Funtionalität des zu verwendenden Transportprotokolls (z.B. RTP). Sie initialisiert den RTP-Header (Setzen der SSRC und des "Payload Types", etc.) und aktualisiert für jedes gesendete Paket die Sequenznummer sowie den Timestamp-Zähler. Für jeden zu verwendenden Encoder-Algorithmus muß nun eine spezielle Klasse implementiert werden, die von der Basisklasse abgeleitet ist. Diese codec-spezifische Klasse liest die PCM-Blöcke aus dem MFIFO, encodiert sie und gibt sie mit Hilfe einer geerbten Methode der Basisklasse über den *IO Wrapper* aus (Abbildung 5.8).

5.3 Zusammenfassung

In diesem Kapitel wurde der Entwurf des Audio-Transcoders dargestellt. Er besteht aus einem *Controller*, der für die Kommunikation mit dem KISS-Netz zuständig ist. Außerdem initialisiert, startet, überwacht und beendet er die *Transcoder-Module*, welche die eigentliche Audio-Kodierung vornehmen. Ein Transcoder-Modul besteht aus einem streaming-fähigen Echtzeit-Audiodecoder sowie einem streaming-fähigen Echtzeit-Audioencoder. Als Transportprotokoll für die Audioströme wird RTP verwendet.

Das folgende Kapitel beschreibt die Implementierung des hier vorgestellten Audio-Transcoders unter Linux 2.0/2.1 und Solaris 2.6.

Kapitel 6

Implementierung

Dieses Kapitel beschreibt die Implementierung des Transcoders. Im ersten Abschnitt wird erläutert, auf welche bereits vorhandenen Softwarekomponenten bei der Implementierung des Audio-Transcoders zurückgegriffen wurde. Die weiteren Abschnitte zeigen zunächst, aus welchen Funktionseinheiten der Audio-Transcoder besteht und wie diese untereinander kommunizieren. Abschließend werden die einzelnen Funktionseinheiten beschrieben.

6.1 Verwendete Softwarekomponenten

Der Audio-Transcoder wurde objektorientiert konzipiert und in C++ unter Ausnutzung der Möglichkeiten der *multithreaded* Programmierung implementiert. Hierzu wurde das unter UNIX-Betriebssystemen übliche pthread-API (Posix 1003.1c [20]) verwendet. Da sich das Betriebssystem nun um das *Scheduling* kümmert, vereinfacht sich die Implementierung parallel ablaufender Programmteile, und man ist nicht mehr gezwungen, `select(2)`-basierte *Scheduling*-Routinen selbst zu implementieren. Allerdings entstehen zusätzliche Probleme bei der Synchronisation der einzelnen Threads, da diese nun parallel auf die Resourcen des Prozesses zugreifen und somit Überschneidungen entstehen können. Um diese Zugriffe zu kontrollieren, wurden Synchronisationsmechanismen wie *Semphore* und *Mutexe* verwendet. Das, in der vom Autor entwickelten Klassenbibliothek *LibStream* [3] enthaltene Thread-Objekt `c_thread` enthält Methoden zum Erzeugen und Terminieren eines Threads sowie zur Abfrage des Thread-Status. Des weiteren erbt es die Funktionalität der *Message Port*-Klasse `c_msg_port` über die die einzelnen Thread-Objekte mit Hilfe von *Messages*, die von der Basisklasse `c_msg` abgeleitet sind, *multithread safe* (MT-safe) miteinander kommunizieren können. Somit steht ein einheitlicher Kommunikationsmechanismus für alle Threads eines Prozesses zur Verfügung, über den globale Messages wie `EXIT_PROCESS` oder `TIMER_EVENT` verschickt, sowie die gesamte Inter-Thread-Kommunikation abgewickelt werden kann. Ein Programm, wie z.B. der Audio-Transcoder, kann daher aus autonom arbeitenden Objekten zusammengesetzt werden, die über diesen Kommunikationsmechanismus Informationen austauschen.

Die Klassenbibliothek *LibStream* ist für Linux-2.0/2.1 in Verbindung mit der *GNU C Library Version 2* [22], Solaris 2.6 sowie Windows 95 bzw. Windows NT 4.0 verfügbar. Eine detaillierte Beschreibung befindet sich in der Dokumentation zur Klassenbibliothek *LibStream*, die dieser Arbeit beigefügt ist.

Der Transcoder kann nur unter Linux/Solaris kompiliert werden, da unter Windows der `fork(2)-dup2(2)-exec(2)`-Mechanismus nicht unterstützt wird. Dieser ermöglicht es z.B., daß der Parent-Prozeß IP-Sockets bereits initialisieren kann, um sie anschließend in Form einer fest definierten Filedescriptor-Nummer an einen Child-Prozeß zu übergeben, so daß dieser die Sockets direkt verwenden kann, ohne sich um deren Konfiguration kümmern zu müssen.

Für die Kommunikation mit dem KISS-Netz wird die von Jens Mödeker entwickelte Klassenbibliothek *NAPhelp*, die das "Service Management Protocol (SMP)" enthält, verwendet. In diesem Zusammenhang wird für Socket-Operationen größtenteils auch die von Karl Jonas entwickelte Klasse *IPLink* benutzt.

Als Encoder- bzw. Decoder-Bibliotheken wurde die MPEG Layer 3 Implementierung der Fraunhofer-Gesellschaft (FhG-IIS) sowie die MPEG II-AAC Implementierung von AT&T verwendet. Da das jeweilige API einer NDA (Non Disclosure Agreement) unterliegt, dürfen die Sourcen der entsprechenden Encoder- bzw. Decodermodule nicht veröffentlicht werden, und sind somit nicht im Anhang abgedruckt.

6.2 Die Komponenten des Audio-Transcoders

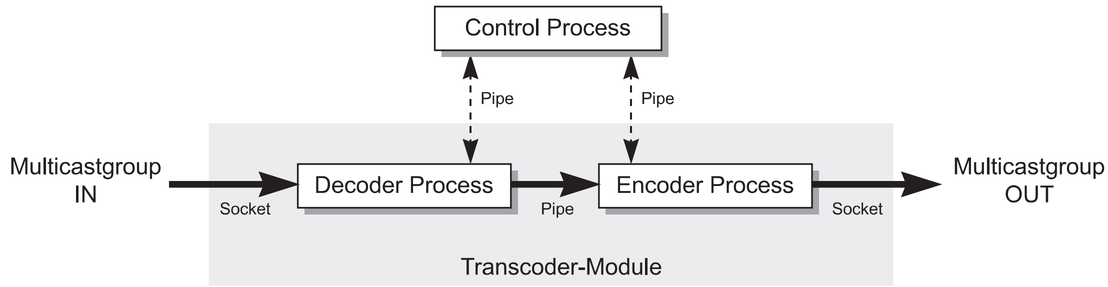

Abbildung 6.1: Die Transcoder-Prozesse

Der Audio-Transcoder besteht aus drei unterschiedlichen Prozessen:

- **Control Process**
  Dieser Prozeß ist für die Steuerung des Transcoders zuständig. Er kommuniziert mit dem KISS-Netz, baut die jeweiligen Daten- bzw. Control-Verbindungen auf und überwacht die Encoder- bzw. Decoder-Prozesse.

- **Decoder Process**
  Dieser Prozeß empfängt die encodierten Daten über einen UDP-Socket aus einer Multicastgruppe, decodiert diese, und schreibt die PCM-Daten in eine Pipe.

- **Encoder Process**
  Dieser Prozeß liest die PCM-Daten aus der Pipe, encodiert sie im neuen Format und sendet sie über einen UDP-Socket in eine neue Multicastgruppe.

Der Audio-Transcoder besteht immer aus einem Control-Prozeß, der eine beliebige Anzahl von Transcoder-Modulen steuern kann. Dabei besteht ein Transcoder-Modul aus je einem Decoder- und einem Encoderprozeß, die durch eine Pipe miteinander verbunden sind. Diese Trennung ist nötig, da einige der verwendenten Encoder- bzw. Decoder-*Libraries* gleichnamige globale Symbole benutzen und es daher nicht möglich ist, sie zu einem ausführbaren Programm (Executeable) zu linken. Der Audio-Transcoder ist somit in der Lage, beliebig viele Audioströme parallel zu transcodieren. Die einzige Beschränkung stellt die Rechenleistung des verwendeten Hostrechners dar.

Um die Encoder- bzw. Decoderprozesse möglichst einfach zu halten, kommunizieren diese nur über fest definierte Filedescriptoren (UDP-Socket, Pipe), die vom Control-Prozeß initialisiert und mit Hilfe der `dup2(2)`-Funktion unter der jeweiligen Filedeskriptor-Nummer installiert werden. Im Falle des Audio-Transcoders bedeutet das, daß *stdin* und *stdout* umgelenkt werden. Dies sorgt für eine klare Aufgabentrennung zwischen den Prozessen, da nur der Control-Prozeß, der bereits die gesamte Kommunikation mit dem KISS-Netz abwickelt, Informationen über die Konfiguration der UDP-Sockets verarbeiten muss. Die Encoder- und Decoderprozesse benutzen einfach die bereits initialisierten Filedescriptoren und benötigen daher keine weiteren Angaben über Einstellungen der UPD-Sockets.

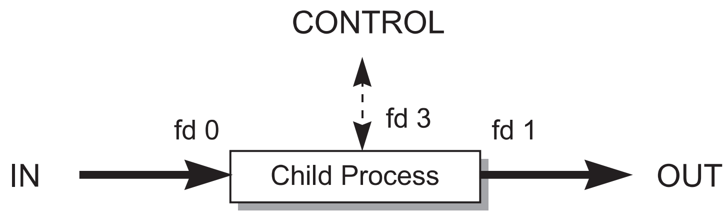

Abbildung 6.2: Umlenken der Filedescriptoren

Beide Prozesse lesen ihren Eingangsstrom von Filedescriptor null (stdin) und geben den Ausgangsstrom über Filedescriptor eins (stdout) wieder aus. Die beiden Filedescriptoren werden also, wie weiter oben beschrieben, in eine Pipe bzw. Socket umgelenkt. Filedesciptor zwei (stderr) ist für Fehlerausgaben in die Konsole reserviert. Über Filedescriptor drei sind die Child-Prozesse jeweils durch eine Pipe mit dem Control-Prozeß verbunden (Abbildung 6.2). Dieser teilt den Child-Prozessen durch Schließen (`close(2)`) der Pipe mit, daß sie sich beenden sollen. Ein Child-Prozeß kann seinerseits dem Control-Prozeß durch Schließen der Pipe anzeigen, daß ein nicht korrigierbarer Fehler aufgetreten ist, aufgrund dessen dieses Transcoder-Modul beendet werden muß.

Der Audio-Transcoder ist prinzipiell in der Lage, Audioströme jedes implementierten Audio-Codecs zu decodieren bzw. einen Audiostrom mit diesen Codecs zu encodieren. Um die, in den vorherigen Kapiteln vorgestellten Szenarien demonstrieren zu können, kann beim Start des Audio-Transcoders über die Kommandozeile festgelegt werden, welche Eingangsströme er decodieren und welche Ausgangsströme er erzeugen soll. Diese Einstellungen gelten für alle Transcoder-Module dieses Audio-Transcoders. Somit kann z.B. ein Audio-Transcoder gestartet werden, der nur MPEG Layer 3 nach MPEG II-AAC transcodieren kann.

6.2.1 Der Control-Prozeß

Der Control-Prozeß bildet die Netzwerk-Schnittstelle und die Verwaltungsinstanz des Transcoders. Auf Anforderung des KISS-Netzes baut er Datenverbindungen auf und startet bzw. beendet die dazugehörigen Transcoder-Module.

Der Control-Prozeß besteht aus mehreren Kern-Thread-Objekten, die jeweils bestimmte Teilaufgaben erfüllen. Durch die Kommunikation untereinander entsteht hieraus die Funktionalität des Control-Prozesses. Drei weitere Hilfs-Threads stellen allgemeine Dienste zur Verfügung, die von allen anderen Threads angefordert werden können.

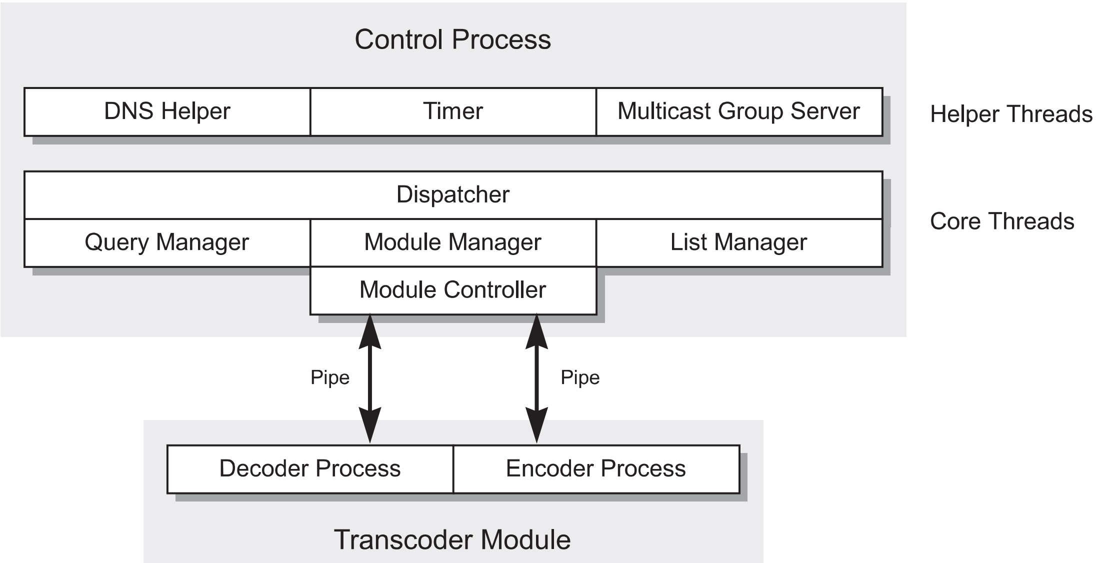

Abbildung 6.3: Schematischer Aufbau des Control-Prozesses

- **Timer (Hilfs-Thread)**
  Der *Timer* stellt allen anderen Modulen *One Shot*- sowie *Intervall*-Timer-Dienste zur Verfügung. Die Kommunikation erfolgt über das *Message System*.

- **Multicast Group Server (Hilfs-Thread)**
  Der *Multicast Group Server* generiert mit Hilfe der "C"-Funktion `rand(3)` zufällige IP-Multicastgruppen-Adressen und stellt sicher, daß die ermittelte Gruppe nicht bereits benutzt wird.

- **DNS Helper (Hilfs-Thread)**
  Der *Domain Name Service*-Helper entkoppelt *Reverse DNS*-Abfragen und speichert eine Liste bereits aufgelöster (resolved) Anfragen, so daß die Antwort bei einer erneuten Anfrage direkt zur Verfügung steht.

- **Dispatcher**
  Der *Dispatcher* bildet die Netzwerkschnittstelle. Er empfängt SMP-Pakete von seinem Netzwerk-Socket, überprüft deren Gültigkeit und verteilt sie an den jeweils zuständigen Thread. Des weiteren empfängt er über seinen *Message Port* SMP-Pakete von den übrigen Threads und sendet diese ins Netz.

- **List Manager**
  Der *List Manager* speichert und aktualisiert eine Liste aller im KISS-Netz verfügbaren *Data Descriptions* (DD). Hierzu wertet er die `DL_ANNOUNCE` und `DL_INFO` Pakete aus. Dies wird z.B. vom *Query Manager* verwendet.

- **Query Manager**
  Der *Query Manager* bearbeitet `DL_QUERY`-Anfragen aus dem KISS-Netz, indem er einen passenden Eingangsstrom sucht, um den angefragten Ausgangsstrom erzeugen zu können. Hierzu gibt es mehrere Möglichkeiten, die vom konkreten Fall abhängig sind.

- **Module Controller**
  Pro Transcoder-Modul wird ein *Module Controller* erzeugt, der das jeweilige Transcoder-Modul steuert. Er baut die benötigten Datenverbindungen auf, erzeugt das Transcoder-Modul, überwacht über den `DL_PING-DL_ALIVE`-Mechanismus die Datenverbindungen und terminiert das Transcoder-Modul.

- **Module Manager**
  Der *Module Manager* verwaltet eine Liste aller aktiven *Module-Controller*, die wiederum jeweils ein Transcoder-Modul steuern. Er ist für das Erzeugen und Terminieren der *Module-Controller* zuständig. Außerdem verteilt er die vom Dispatcher kommenden SMP-Pakete an die entsprechenden *Module-Controller*.

Abbildung 6.4 zeigt, welcher Thread welche SMP-Pakete bearbeitet. Aus Gründen der Übersichtlichkeit, werden die Rückverbindungen der einzelnen Threads zum Dispatcher nicht dargestellt:

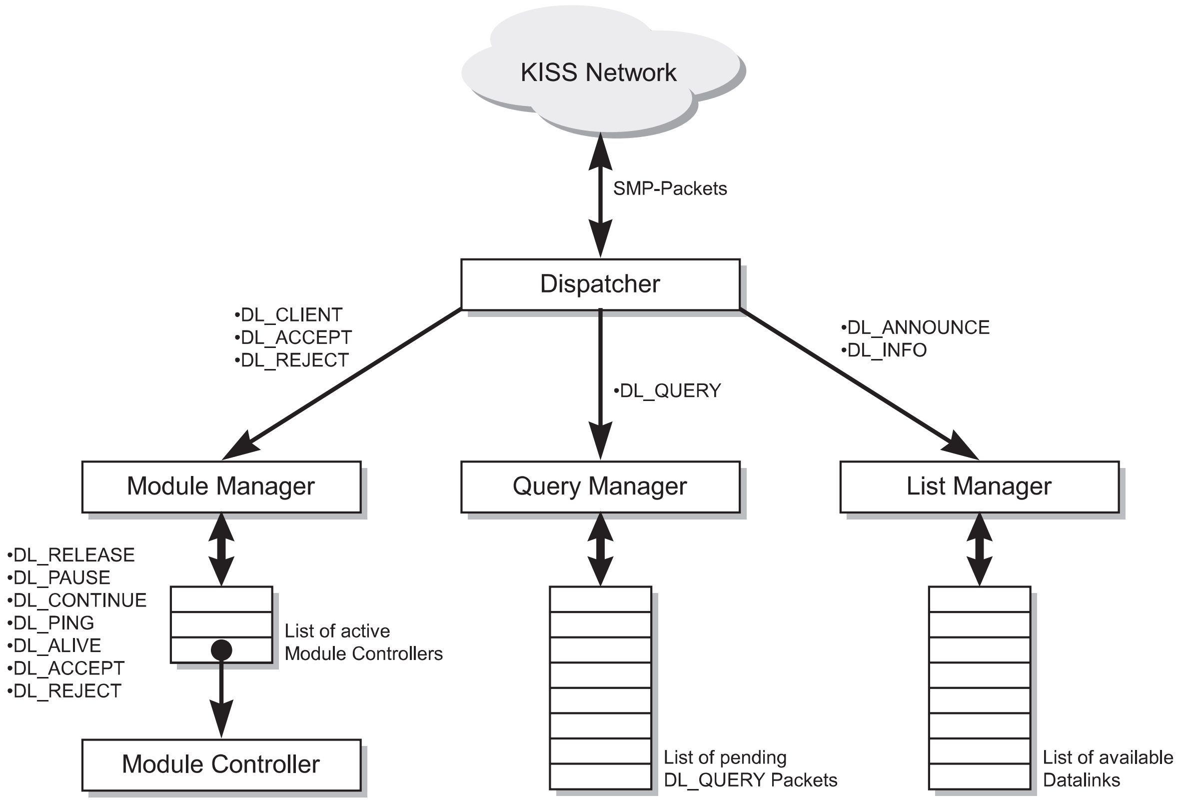

Abbildung 6.4: Die Kommunikation der Kern-Threads im Control-Prozeß

6.2.2 Der Decoder-Prozeß

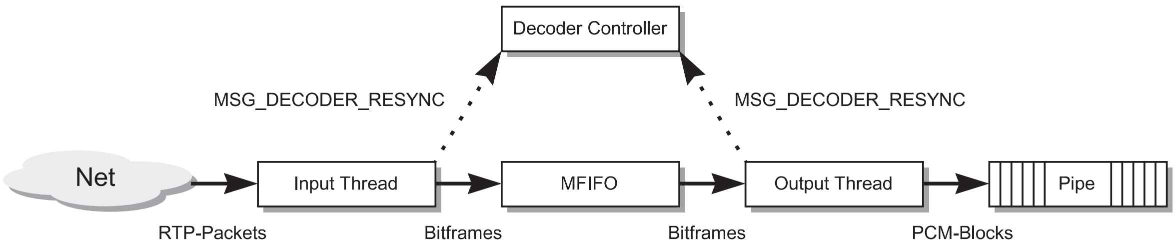

Abbildung 6.5: Threads im Decoder Prozeß

Der Decoder-Prozeß wird von einem *Module-Controller* des Control-Prozesses gestartet. Der *Module-Controller* teilt ihm über die Kommandozeile die benötigten Parameter, den Payload-Type und die SSRC des zu empfangenden Audiostromes mit. Daraufhin wird der Audio-Decoder entsprechend initialisiert und anschließend gestartet. Über den Filedescriptor null (stdin) liest dieser nun die RTP-Pakete ein, decodiert sie in einen PCM-Strom und gibt sie über den Filedescriptor eins (stdout) in die Pipe aus. Bevor der Decoder die ersten PCM-Daten ausgibt, sendet er zunächst einen Datenblock in die Pipe, der die Parameter der PCM-Daten (Abtastrate, Anzahl der Kanäle, Bits pro Sample), sowie den RTP Timestamp (TS) des ersten empfangenen Paketes enthält (Abbildung 6.6). Diese Parameter werden vom Encoder benötigt. Der Decoder-Prozeß beendet sich selbst, wenn ein nicht korrigierbarer Fehler aufgetreten ist. Ansonsten beendet er sich, sobald der *Module Controller* die *Control Pipe* schließt.

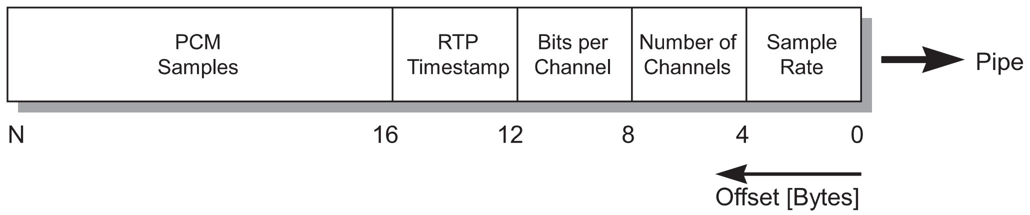

Abbildung 6.6: Übertragung des Parameter-Blocks über die Pipe

Der *Decoder Controller*-Thread wird benötigt, um den Decoder neu zu initialisieren bzw. neu mit dem zu empfangenden Audiostrom zu synchronisieren. Diese Resynchronisation kann vom Input- oder Output-Thread über eine *Message* des Types `MSG_DECODER_RESYNC` ausgelöst werden, wenn diese einen Übertragungsfehler feststellen, der durch den MFIFO nicht korrigiert werden kann.

6.2.3 Der Encoder-Prozeß

Abbildung 6.7: Threads im Encoder Prozeß

Der Encoder-Prozeß wird von einem *Module-Controller* des Control-Prozesses gestartet. Der *Module-Controller* teilt ihm über die Kommandozeile die benötigten Parameter, den Payload-Type, die SSRC, und die Datenrate des zu erzeugenden Audiostromes mit. Daraufhin wird der Audio-Encoder entsprechend initialisiert und anschließend gestartet. Über den Filedescriptor null (stdin) liest dieser nun die PCM-Daten aus der Pipe ein, encodiert sie, und gibt sie in Form von RTP-Paketen über Filedescriptor eins (stdout) aus. Der erste Datenblock, den der Encoder einliest, enthält die Parameter des PCM-Stromes und den RTP-Timestamp (Abbildung 6.8). Dieser wird benötigt, um die Timestamps des vom Encoder erzeugten RTP-Ausgangsstromes mit den Timestamps des vom Decoder empfangenen RTP-Eingangsstromes abzugleichen. Der Encoder-Prozeß wird beendet, wenn ein nicht korrigierbarer Fehler auftritt, oder wenn die *Control Pipe* vom *Modul Controller* geschlossen wird.

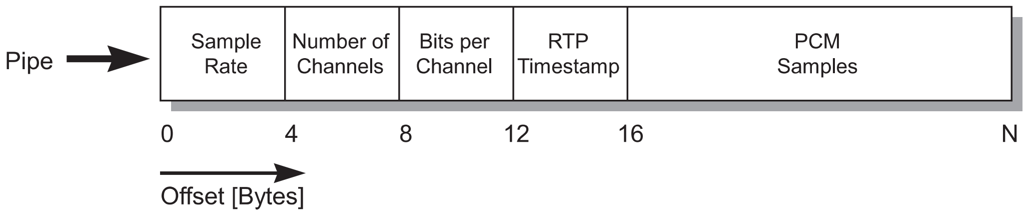

Abbildung 6.8: Übertragung des Parameter-Blocks über die Pipe

6.3 Programmablauf der einzelnen Objekte

In diesem Abschnitt wird die Funktion und der Programmablauf innerhalb der einzelnen Thread-Objekte erklärt.

Bei den Hilfs-Threads werden die API-Methoden vorgestellt, mit deren Hilfe der Dienst des jeweiligen Objekts in Anspruch genommen werden kann. Diese Methoden vereinfachen die teilweise komplizierte Kommunikation über Messages, da sie für den Benutzer als einfache Funktionsaufrufe erscheinen.

Der Ablauf der Kern-Objekte des Audio-Transcoders wird schematisch mittels folgender vereinfachten "C"-ähnlichen Syntax dargestellt:

```
thread test( list of parameters) {  //*** we may pass the thread a list of parameters
  IF count = max OR error occured { //*** if
    reset the timer                 //*** do this
  }
  ELSE {                            //*** else
    increment timer value           //*** do that
  }
  SWITCH type {                     //*** C-like switch()
    CASE TYPE_ABC:                  //*** if type = TYPE_ABE
      do something                  //*** do this
    CASE TYPE_XYZ:                  //*** if type = TYP_XYZ
      do something different        //*** do that
  }
  WHILE transcoder active {         //*** while the transcoder is active
    do what you like                //*** do what follows
    BREAK;                          //*** continue execution after the 'while' loop
  }
}
```

6.3.1 Timer

Das *Timer*-Objekt ist ein Hilfsobjekt, das von Jens Mödeker implementiert wurde. Es empfängt über seinen *Message Port* Anforderungen für *One Shot*- bzw. *Intervall-Events* anderer Threads. Tritt eines dieser *Events* ein, sendet der *Timer* an den *Message Port* des jeweiligen Threads eine entsprechende *Message*, um diesen über den Ablauf des Timers zu informieren. Um mehrere *Timer Events* in einem Thread-Objekt verarbeiten zu können, ist es möglich, jedem *Timer Event* eine Referenz auf ein beliebiges Objekt zuzuordnen, um die *Events* unterscheiden zu können.

Das Timer-Objekt enthält folgende API-Methoden:

```
void insert_timer( void *item, int timeout);     // new one shot timer
void update_timer( void *item, int timeout);     // reset one shot timer
void remove_timer( void *item);                  // remove one shot timer

void insert_interval( void *item, int interval); // new interval timer
void remove_interval( void *item);               // remove interval timer

void remove_all( void *item);                    // remove all timers of <item>
void remove_thread();                            // remove all timers of the thread
```

6.3.2 Multicast Group Server

Das *Multicast Group Server*-Objekt ist ein Hilfsobjekt. Über seinen *Message Port* können zufällige Adressen von unbenutzten Multicastgruppen angefordert werden. Diese Adressen werden zunächst mit Hilfe der "C"-Zufallsfunktion `rand(3)` erzeugt. Anschließend wird ein entsprechender Socket geöffnet um zu prüfen, ob die Multicastgruppe bereits verwendet wird. In diesem Falle wird eine neue Adresse erzeugt. Wird die generierte Multicastgruppe nicht verwendet, so wird ihre Adresse in Form einer *Message* an den anfordernden Thread geschickt.

Das Multicast-Group-Server-Objekt enthält die folgende API-Methode:

```
c_sockaddr_in *get_mcast_group(); // requests a new multicast group
```

6.3.3 DNS Helper

Das *DNS Helper*-Objekt ist ein Hilfsobjekt, das von Jens Mödeker implementiert wurde. Es wird verwendet, um *Reverse-DNS*-Anfragen, die mit Hilfe von der `gethostbyaddr(3)`-Funktion ausgeführt werden, zu entkoppeln und bereits aufgelöste Anfragen zu "cachen" damit sie bei der nächsten Anfrage direkt beantwortet werden können. Hierzu empfängt das Objekt eine entsprechende Anfrage über seinen *Message Port* und prüft zunächst, ob sich die jeweilige Adresse bereits im *Cache* befindet. Wenn nicht, wird eine entsprechende DNS-Anfrage mittels `gethostbyaddr(3)` gestartet. Der anfragende Thread kann durch einen Timeout festlegen, wie lange er auf das Ergebnis warten möchte. Läuft dieser Timeout ab, und die `gethostbyaddr(3)`-Funktion ist noch nicht beendet, erhält der Thread eine Antwort-Message (Reply), die die IP-Adresse als ASCII-String enthält (z.B. "129.26.120.57"). Versucht er es einige Sekunden später erneut, befindet sich die Antwort bereits im Cache, sofern `gethostbyaddr(3)` die Adresse auflösen konnte (z.B. "mats.gmd.de").

Das DNS-Helper-Objekt enthält die folgende API-Methode:

```
char *get_dnsname( u_int32 ip, int mtimeout=0); // returns the DNS name of <ip>
```

6.3.4 Dispatcher

Das *Dispatcher*-Objekt stellt die Netzwerkschnittstelle des Audio-Transcoders dar. Es empfängt SMP-Pakete aus dem KISS-Netz oder sendet SMP-Pakete in das KISS-Netz. Er besteht aus einer Basisklasse, die von Jens Mödeker für den NAP implementiert wurde. Da sowohl der NAP als auch der Audio-Transcoder das Network-Network-Interface (NNI) verwenden, kann der Dispatcher für beide ähnlich ausgeführt sein. Lediglich die Methode `recv( c_smp *)`, die empfangene SMP-Pakete verarbeitet und an die jeweiligen *Manager Threads* verteilt, muß speziell für den Audio-Transcoder angepaßt werden.

Der schematische Ablauf sieht wie folgt aus:

```
thread dispatcher( <our input/output payload type>) {
  WHILE transcoder is running {
    IF we´ve received a message on our message port {
      send the attached SMP packet into the KISS network
    }
    IF we´ve received a SMP packet on our KISS network socket {
      SWITCH SMP.type {
        CASE DL_INFO:
        CASE DL_ANNOUNCE:
          IF SMP.ptype = <our input payload type>
            forward packet to the list manager
        CASE DL_QUERY:
          IF SMP.ptype = <our output payload type>
            forward packet to the query manager
        CASE DL_ACCEPT:
        CASE DL_REJECT:
          IF SMP.ptype = <our input payload type>
            forward packet to the module manager
        CASE DL_CLIENT:
        CASE DL_CONTINUE:
        CASE DL_PAUSE:
          IF SMP.ptype = <our output payload type>
            forward packet to the module manager
        CASE DL_ALIVE:
        CASE DL_PING:
        CASE DL_RELEASE:
          IF SMP.ptype = <our input/output payload type>
            forward packet to the module manager
      }
    }
  }
}
```

6.3.5 List Manager

Der *List Manager* verwaltet eine doppelt-verkettete Liste aller im KISS-Netz verfügbaren *Data Descriptions*. Hierzu wertet er die SMP-Pakettypen `DL_INFO` und `DL_ANNOUNCE` aus. Er fügt neue DDs in die Liste ein und löscht DDs, wenn innerhalb des angegebenen Intervalls kein erneutes Announce-Paket eintrifft. Der *Query Manager* sowie der *Controller Manager* können von ihm eine Liste aller verfügbaren DDs anfordern, die auf eine bestimmte *Wildcard-Query*-DD passen. Zusätzlich informiert der *List Manager* den *Query Manager* mit einer *Message* des Types `MSG_NEW_DATA_LINK` über jede neu registrierte DD, da der *List Manager* auch die `DL_INFO`-Pakete verarbeitet, die als Antwort auf eine `DL_QUERY`-Anfrage des *Query Managers* eintreffen.

Der schematische Ablauf sieht wie folgt aus:

```
thread list_manager(){
  WHILE transcoder is running {
    get next message from message port
    SWITCH msg.type {
      CASE DL_INFO:           //*** answer to a preceding DL_QUERY
      CASE DL_ANNOUNCE:       //*** a NAP announces a DD
        IF a new DD is announced {
          add this DD to our list
          send a MSG_NEW_DATA_LINK to the query manager
        }
        ELSE {
          reset the timer for this DD
        }
      CASE MSG_TIMER:         //*** the timer for a DD has expired
        remove the timeouted DD from the list
      CASE MSG_DL_LOOKUP:     //*** the query manager looks for a suitable input DD
        return a list of all matching DDs
    }
  }
}
```

6.3.6 Query Manager

Der *Query Manager* bearbeitet ausschließlich SMP-Pakte des Types `DL_QUERY`, indem er versucht einen passenden Eingangsstrom zu finden, aus dem der angefragte Ausgangsstrom erzeugt werden kann. Zuerst versucht er entsprechende DD in der Liste des *List Managers* zu finden. Ist dort keine passende DD registriert, stellt er selbst eine `DL_QUERY`-Anfrage nach einer entsprechenden DD ins KISS-Netz, die von einem NAP oder einem weiteren Audio-Transcoder beantwortet werden kann. Erhält er hierauf keine Antwort, versucht er es mehrmals mit sinkender limitierter Bandbreite und steigender TTL.

Der schematische Ablauf sieht wie folgt aus:

```
thread query_manager() {
  WHILE transcoder is running {
    get next message from message port
    SWITCH msg.type {
      CASE DL_QUERY:          //*** a NAP/SA queries for a DD
        IF this DL_QUERY is already in our list of pending DL_QUERY packets {
          drop it
        }
        ELSE {
          send a MSG_DL_LOOKUP to the list manager
          IF have a matching input stream {
            answer with a DL_INFO packet
          }
          ELSE {
            add this DL_QUERY to our list of pending DL_QUERY packets
            send a new DL_QUERY packet because we are looking for an input DD
            we start with TTL=16, now bandwith limitation
            start the timer for out DL_QUERY
          }
        }
      CASE MSG_NEW_DATA_LINK: //*** the list manager has registered a new DD
        check our list, if we can use the new DD to answer a pending DL_QUERY
      CASE MSG_TIMER:         //*** the timer for a pending DL_QUERY has expired
        IF TTL < 192 {
          reset timer
          send new DL_QUERY, decrease bandwidth limit, increase TTL
        }
        ELSE { //*** not matching input DD found
          remove the DL_QUERY packet from our list
      }
    }
  }
}
```

6.3.7 Module Manager

Der *Module Manager* ist für das Erzeugen und Beenden der *Module Controller* verantwortlich. Alle aktiven *Module Controller* sind in einer Liste gespeichert, so daß z.B. der *Query Manager* jederzeit feststellen kann, welche Eingangsströme bereits verwendet werden. Außerdem kann der *Module Manager* anhand dieser Liste feststellen, zu welchem *Module Controller* ein bestimmtes SMP-Paket durchgereicht werden muß.

Der schematische Ablauf sieht wie folgt aus:

```
thread module_manager() {
  WHILE transcoder running {
    get next message from message port
    SWITCH msg.type {
      CASE MSG_REMOVE_MODULE: //*** a module controller reuqests its cancelation
        terminate and remove the corresponding module controller
      CASE DL_CLIENT:         //*** someone requests a data link
         IF requested stream is still running {
           forward SMP packet to corresponding module controller
         }
         ELSE {
           IF required input stream is available
             create a new module controller for the new stream
           ELSE
             send a DL_REJECT
         }
      CASE DL_ACCEPT:         //*** these pakets are handled by the
      CASE DL_REJECT:         //*** modules controllers
      CASE DL_RELEASE:        //***
      CASE DL_ALIVE:          //*** we simply forward them
      CASE DL_PAUSE:          //***
      CASE DL_CONTINUE:       //***
        forward SMP packet to corresponding module controller
    }
  }
}
```

6.3.8 Module Controller

Der *Module Controller* steuert jeweils ein Transcoder Modul. Er baut die benötigten Datenverbindung auf, erzeugt die Child-Prozesse, installiert mittels `dup2(2)` die Socket- bzw. Pipe-Filedescriptoren auf *stdin* und *stdout* und startet sie. Anschließend überwacht er die Datenverbindungen sowie die Child-Prozesse und reagiert auf etwaige Fehler, indem er zuerst die Child-Prozesse terminiert und dann eine *Message* des Types `MSG_REMOVE_MODULE` an den *Module Manager* schickt.

Der schematische Ablauf sieht wie folgt aus:

```
thread module_controller( DD for input data link, we provide this output data link) {
  request input data link with DL_CLIENT from our 'server'
  IF timeout OR error {
    send DL_RELEASE to client
    BREAK;
  }
  request an unused multicast group from multicast group server
  send DL_ACCEPT to client
  start child processes
  WHILE transcoder module running {
    check the child processes control pipes
    IF error occured
      BREAK;
    check if the msg port has pendimg messages
    SWITCH msg.type {
      CASE RELEASE:              //*** one client releases our stream
        IF the last client releases us
          BREAK;
      CASE DL_CLIENT:            //*** we have a new listener
        put the new listener into the listener list
      CASE DL_PAUSE:             //*** one client paused our stream
        IF all clients pause our stream
          send DL_PAUSE to our 'server'
      CASE DL_CONTINUE:          //*** at least one client continues the stream
        send DL_CONTINUE to our 'server'
      CASE DL_ALIVE:             //*** DL_ALIVE from our ``server''
        reply with DL_PING
    }
    check if our clients are still listening, send them a DL_PING
    IF we don´t receive a DL_ALIVE after some retries {
      remove the client from the listener list
      IF last client removed
        BREAK;
    }
  }
  send a MSG_REMOVE_MODULE to the module manager
  wait for module manager to terminate us
}
```

6.3.9 Decoder Input-Thread

Der *Decoder Input-Thread* empfängt RTP-Pakete über den *IO Wrapper*, überprüft deren Gültigkeit und sortiert sie in den MFIFO ein. Kann er mehrmals hintereinander ein Paket aufgrund einer falschen Sequenznummer nicht in den MFIFO einsortieren, erzeugt er eine Message des Types `MSG_DECODER_RESYNC` und veranlaßt damit den Decoder-Controller den Decoder neu zu initialisieren. Nach dem Empfang des ersten Paketes wird die Art der Netzverbindung ermittelt (LAN oder WAN), und die Größe des MFIFOs darauf angepaßt.

Der schematische Ablauf sieht wie folgt aus:

```
thread decoder_input( expected payload type, expected SSRC) {
  WHILE decoder running
    read RTP packet via IO Wrapper
    ...
    IF rtp.version = 2 AND rtp.ssrc = expected SSRC {
      IF rtp.pt = expected payload type {
        IF first_packet_received {
          determine the connection type (LAN or WAN)
          ...
          SWITCH connection type
          CASE LAN:
            no extra_buffers
          CASE WAN:
            use extra_buffers
          }
          initialize the MFIFO according to the connection type
          ...
        }
        put the packet and its SEQ/TS into the MFIFO
        ...
        IF put() fails {
          send a MSG_DECODER_RESYNC to the decoder controller
          BREAK;
        }
      }
    }
  }
}
```

6.3.10 Decoder Output-Thread

Der *Decoder Output-Thread* liest RTP-Pakete aus seinem MFIFO, decodiert bzw. rekonstruiert diese und gibt sie über den *IO Wrapper* aus. Im Falle eines Netzwerkfehlers (kein gültiges Paket innerhalb eines längerern Zeitraumes empfangen) erzeugt er eine Message des Types `MSG_DECODER_RESYNC` und veranlaßt damit den Decoder-Controller den Decoder neu zu initialisieren. Bevor er den ersten PCM-Block ausgibt, initialisiert er das Ausgabe-Medium (Setzen der PCM Parameter).

Der schematische Ablauf sieht wie folgt aus:

```
thread decoder_output() {
  WHILE decoder running
    get packet from MFIFO
    ...
    IF network error {
       send a MSG_DECODER_RESYNC to the decoder controller
       BREAK;
    }
    IF packet is lost {
      reconstruct packet into a PCM block
      ...
    }
    ELSE{
      decode packet into a PCM block
      ...
    }
    IF playing first PCM block {
      initialize output media, write PCM parameter block into the pipe
      ...
    }
    write PCM block via IO Wrapper
    ...
    delete packet
  }
}
```

6.3.11 Encoder Input-Thread

Der *Encoder Input-Thread* liest PCM-Blöcke, in vom jeweiligen Audio-Codec vorgegebenen Größen, über den *IO Wrapper* ein und speichert sie im MFIFO. Im Falle eines Überlaufs des MFIFO wird dieser neu initialisiert und die bereits gespeicherten PCM-Blöcke werden verworfen.

Der schematische Ablauf sieht wie folgt aus:

```
thread encoder_input( max MFIFO size) {
  WHILE encoder running {
    alloc next MFIFO buffer for writing
    ...
    IF MFIFO overflow {
      empty MFIFO
      lock next MFIFO buffer block for writing
      ...
    }
    read PCM block via IO wrapper into MFIFO buffer
    ...
    IF read error {
      BREAK;
    }
    unlock locked MFIFO buffer block
    ...
  }
}
```

6.3.12 Encoder Output-Thread

Der *Encoder Output-Thread* liest PCM-Blöcke aus dem MFIFO, encodiert sie mit Hilfe des jeweiligen Audio-Codecs, erzeugt einen RTP-Header und sendet die Pakete über den *IO Wrapper* ins Netz. Je nach verwendetem Audio-Codec und den daraus resultierenden Größen der encodierten Pakete (Bitframes) werden mehrere Bitframes in ein RTP-Paket gepackt, damit der RTP/UDP/IP-Overhead nicht unverhältnismäßig groß wird.

Der schematische Ablauf sieht wie folgt aus:

```
thread encoder_output( payload type, bitrate, SSRC) {
  read PCM Parameter block from the pipe
  initialize the encoder
  IF error
    BREAK;
  WHILE encoder running {
    lock next MFIFO buffer for reading
    ...
    get pointer to PCM block in MFIFO
    ...
    encode PCM block
    ...
    generate RTP header
    ...
    send new RTP packet via the IO Wrapper
    ...
    IF send error {
      BREAK;
    }
    unlock locked MFIFO Buffer
    ...
  }
}
```

Kapitel 7

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein Audio-Transcoder zur Datenraten- und Kodierungsformatumsetzung entworfen und implementiert. Durch die Einbettung in das KISS-Konzept kann sich der Audio-Transcoder dynamisch auf die jeweiligen Anforderungen einstellen und seine Dienste dem Endbenutzer auf transparente Weise zur Verfügung stellen.

7.1 Zusammenfassung

Der für den Einsatz im Internet (MBone) interessante Aspekt ist sicherlich die Datenratenumsetzung. Im Vergleich zu den zur Zeit gebräuchlichen A/V-MBone-Tools wie *vic* oder *vat* ist es möglich, bei Bedarf einen Datenstrom zeitgleich in verschiedenen Datenraten anzubieten. Momentan könnte ein solcher Dienst nur durch einen starr aufgesetzten RTP-Translator angeboten werden. Dieser würde den Dienst während seiner gesammten Laufzeit anbieten, egal ob es gerade "Zuhörer" gibt oder nicht. Die NAP/Audio-Transcoder-Kombination ermöglicht es, die RTP-Translator-Dienste des Audio-Transcoder nur bei Bedarf und dann auch möglichst für den jeweiligen Benutzer optimal angepaßt, zu nutzen. Durch die Multicast-Unicast-Umsetzungsfähigkeit des NAPs wäre es weiter möglich, auch Endbenutzer mit Point-to-Point-Zugängen (Modem, ISDN), die in der Praxis nicht IP Multicast-fähig sind, zu bedienen. Somit könnte ein Service wie z.B. "FAU TV" weiterhin breitbandig (ca. 550 kBit/s) im WiN-MBone angeboten werden, während für nicht WiN-Teilnehmer oder Ponit-to-Point-Benutzer dieser Service zeitgleich mit einer geringeren Datenrate und sogar über Unicast-Verbindung verteilt werden könnte.

Das zur Zeit implementierte KISS-Szenario (NAP, Audio-Transcoder, Audio-Client und Audio-Server) läuft im lokalen Netz sehr stabil. Einige Tests zwischen der GMD, der Uni Köln sowie der TU Berlin verliefen im Großen und Ganzen vielversprechend. Ein wenig störend wirkte sich der teilweise ziemlich langsame "Verbindungsaufbau" im MBone zwischen den einzelnen KISS-Kompomenten aus. Größere Testreihen mit möglichst vielen Benutzern (teilweise außerhalb des WiN und vorzugsweise im Ausland) waren wegen des gesetzten Zeitrahmens bis jetzt nicht möglich. Einige Versuche sind geplant und können hoffentlich innerhalb der nächsten Wochen durchgeführt werden.

7.2 Betrachtung der Skalierbarkeit

Der Overhead durch die SMP-Protokollinformationen, die der Audio-Transcoder zusätzlich zu den eigentlichen Multimediaströmen erzeugt, kann praktisch vernachlässig werden, da sich der Audio-Transcoder auf der SMP-Ebene vollkommen passiv verhält. Solange kein Dienst von ihm angefordert wird, empfängt er nur *Announcement*-Pakete, sendet aber keine eigenen Pakete ins Netz. Im Falle einer Dienstanforderung ensteht nur bei der eher selteneren Konstellation einer Transcoder-Verkettung ein zusätzlicher Paket-Austausch in Form von `DL_QUERY`- und entsprechenden `DL_INFO`-Paketen. Jens Mödeker hat in seiner Diplomarbeit dargelegt, daß der Verbindungskontroll-Mechanismus über `die DL_PING/DL_ALIVE`-Pakete bei einem 8 kBit/s-Datenstrom einen Overhead ca. 1.1...1.8% erzeugt. Bei steigender Datenrate und damit zunehmender Paketgröße nimmt dieser Overhead weiter ab.

Durch die Verwendung von IP-Multicast kann die Anzahl der Clients pro Transcoder-Modul beliebig groß werden. Der Audio-Transcoder schränkt somit die Skalierbarkeit eines KISS-Szenarios nicht ein. Die Kapazität des Audio-Transcoders läßt sich beliebig skalieren, da es ohne weiteres möglich ist, mehrere identische Audio-Transcoder parallel im Netz zu starten, um andere Transcoder zu entlasten. Dies ist zum Beispiel nötig, wenn ein Datenstrom in mehreren verschiedenen Bitraten angeboten werden soll, die Rechenleistung des Host-Rechners aber nicht für mehrere Transcoder-Module ausreicht.

7.3 Integration skalierbarer Videocodecs

Um MBone-Tools wie *vic* oder *vat* zu ersetzen, und um damit Dienste wie “FAU TV” bzw. deren Dienstqualität für den einzelnen Benutzer zu verbessern, ist es unerläßlich, auch Videosignale übertragen und ihre Bandbreite den Gegebenheiten anpassen zu können. Da die Transcodierung von Videosignalen viel rechenintensiver ist, als bei Audiosignalen, bietet sich hier die Verwendung anderer Verfahren wie z.B. der skalierbaren Videokodierung an. Ein Videostrom besteht hier aus mehreren Einzelströmen: einem (schmalbandigen) Basisstrom sowie mehreren Strömen, die Extra-Informationen enthalten, mit deren Hilfe qualitativ hochwertigere Videosignale reproduziert werden können. Ein Benutzer würde zunächst den Basisstrom anfordern und schließlich so viele Extraströme, bis die ihm zur Verfügung stehende Bandbreite erschöpft wäre (Abbildung 7.1).

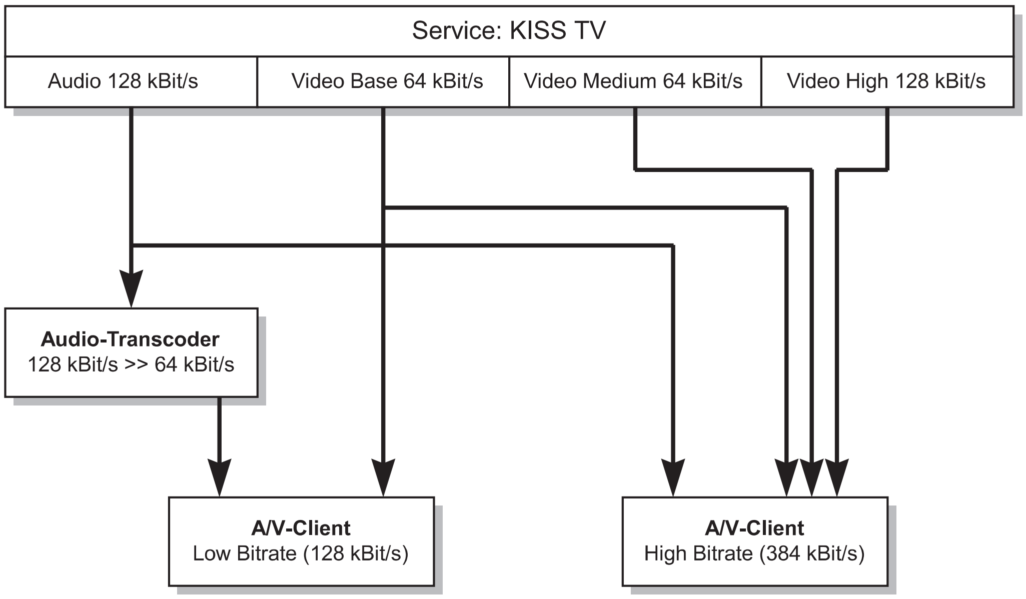

Abbildung 7.1: Der Audio-Transcoder und die skalierbare Videokodierung

Ein solches skalierbares Videokodierverfahren [23] ist seit kurzem an der GMD verfügbar. Zur Zeit wird geprüft, wieviel Aufwand die Implementierung eines entsprechenden Video-Encoders, sowie der dazugehörigen Client-Funktionalität erfordern würde.

Literaturverzeichnis

[1] K.Jonas, 1997, “Forget The Net”, Proceedings of IEEE DMS’97, Seite 103

[2] K.Jonas, P.Kanzow, M.Kretschmer, 1997, “Audiostreaming On The Internet”, Proceedings of IEEE ISIE’97 Vol. I, Seite SS71

[3] M.Kretschmer, GMD.IMK, 1997, “Libstream V0.8”

[4] H.Schulzrinne, Casner, Frederick, Jacobson, 1996, “RTP: A Transport protocol for Real-Time Applications”, RFC 1889

[5] Marina del Rey, 1981, “Internet Protocol - ARPA Internet Program Protocol Spezifikation”, RFC 791

[6] Marina del Rey, 1980, “DoD Standard Transmission Control Protocol”, RFC 761

[7] J.Postel, 1980, “DoD Standard User Datagram Protocol”, RFC 768

[8] S.Deering, 1989, “Host Extensions for IP Multicasting”, RFC 1112

[9] H.Schulzrinne, 1996, “RTP Profiles for Audio and Video Conferences with minimal Control”, RFC 1890

[10] D.Waitzman, C.Partridge, S.Deering, 1988, “Distance Vector Multicast Routing Protocol”, RFC 1075

[11] USC, CISCO, UMICH, XEROX, LBL, 1997, “Protocol Independed Multicast-Sparse Mode (PIM-SM)”, RFC 2117

[12] The ATM Forum, 1996, “Traffic Management Specification Version 4.0”, af-tm-0056.000

[13] CISCO, ArrowPoint, 1997, “RSVP Management Information Base”, IETF-Draft draft-ietf-rsvp-v2-mib-00.txt

[14] ISO/IEC 11172.3, MPEG I - Audio

[15] ISO/IEC 13718.3, MPEG II - Audio

[16] ISO/IEC 13718.7, Advanced Audio Coding, Part 7 (AAC)

[17] Dolby Laboratories, http://www.dolby.com/tech/parametr.html, 19.11.1997

[18] ITU-T G.723.1

[19] ITU-T G.711

[20] IEEE/ANSI Std 1003.1, 1996, "Information Technology - Portable Operating System Interface (POSIX(R)) - Part 1: System Application: Program Interface (API) [C Language]"

[21] J.Mödeker, Diplomarbeit an der FH Köln: "Transparenter Netzwerkzugang für multimediale Streaming-Dienste - Spezifikation, Implementierung und Bewertung am Beispiel eines Echtzeit-Streaming-Dienstes im Internet"

[22] GNU Project - Free Software Foundation (FSF), "GNU C Library Version 2", http://www.gnu.org/software/libc/libc.html, 15.01.1998

[23] U.Horn, B.Girod, 1997, "Scalable video transmission for the Internet"

Anhang A

Quellcodes

Der Anhang enthält die Quellcodes des Audio-Transcoders. Die folgende Auflistung zeigt, welche Objekte in welchen Quellcode-Dateien zu finden sind:

- Globale Funktionen, Objekte und Variablen
  `Makefile`
  `version.h, main.cc`
  `help.h, help.cc`
  `appsighandler.h, appsighandler.cc`
  `datalink.h, datalink.cc`
  `process.h, process.cc`
  `tcprocess.h, tcprocess.cc`
- Timer
  `timer.h, timer.cc`
- Multicast Group Server
  `mcipserver.h, mcipserver.cc`
- DNS Helper
  `dns.h, dns.cc`
- Dispatcher
  `dispatch.h, dispatch.cc`
  `tcdispatch.h, tcdispatch.cc`
- List Manager
  `listmgr.h, listmgr.cc`

- Query Manager
  `querymgr.h`, `querymgr.cc`
- Module Controller
  `module.h`, `module.cc`
- Module Manager
  `controller.h`, `controller.cc`
- Decoder
  `dec.cc`
  `decoder.h`, `decoder.cc`
  `decfifo.h`, `decfifo.cc`
  `decpcm.h`, `decpcm.cc`
  `iowrapper.h`
- Encoder
  `enc.cc`
  `encoder.h`, `encoder.cc`
  `encfifo.h`, `encfifo.cc`
  `encpcm.h`, `encpcm.cc`
  `iowrapper.h`