Mischen und Multiplexen von Audioströmen in computergestützten Konferenzsystemen

Jürgen Nickelsen

Diplomarbeit

Technische Universität Berlin
Fachbereich Informatik
Fachgebiet Systemanalyse und EDV
Forschungsschwerpunkt Prozeßdatenverarbeitung im Prozeßrechnerverbund

Aufgabensteller: Professor Dr. H. Krallmann
Mitbetreuer: Sigrid Gleser, Gerrit Kalkbrenner

10. Januar 1996

Kenner

Gut! Brav, mein Herr! Allein
Die linke Seite
Nicht ganz gleich der rechten;
Hier erscheint es mir zu lang,
Und hier zu breit;
Hier zuckts ein wenig,
Und die Lippe
Nicht ganz Natur,
So tot noch alles!

Künstler

O ratet! helft mir,
Daß ich mich vollende!
Wo ist der Urquell der Natur,
Daraus ich schöpfend
Himmel fühl und Leben
In die Fingerspitzen hervor?
Daß ich mit Göttersinn
Und Menschenhand
Vermöge zu bilden,
Was bei meinem Weib
Ich animalisch kann und muß!

Kenner

Da sehen sie zu.

Künstler

So!

Johann Wolfgang v. Goethe

KAPITEL 1 Vorwort

In dieser Arbeit beschreibe ich die Entwicklung des Programms „Audio Multiplexer Engine“ (AMEngine), das ich im Rahmen des Projekts „Multimediale Teledienste“ an der TU Berlin konzipiert und implementiert habe. Die AMEngine ist in erster Linie als Audio-Subsystem der Videokonferenz-Applikation „Multimedia Collaboration“ (MMC) vorgesehen, kann aber auch für andere Anwendungen benutzt werden.

Nach der Definition der verwendeten Terminologie und einem kurzen Abschnitt über die Technologie der Digitalisierung von Audio-Signalen führe ich in das Umfeld der Arbeit ein.

Die Beschreibung der Anforderungen an die AMEngine, der Entstehungsgeschichte und des endgültigen Entwurfs bilden den Hauptteil dieser Arbeit. Ein Bericht über die Erfahrungen mit der vorliegenden Implementierung und ein Ausblick auf zukünftige Entwicklungen bilden den Abschluß.

Unix Reference Manual Pages

Häufig benutze ich die Namen von Funktionen des Unix-Systems, in der üblichen Weise verbunden mit einem Verweis auf die Reference Manual Pages. Steht im Text „`select`(2)“, so ist damit die Funktion select() gemeint, die in Abschnitt 2 der Reference Manual Pages zum verwendeten Betriebssystem HP-UX beschrieben ist. [HP 1992]

Inhalt

Vorwort 3

Begriffe 9

KAPITEL 1 Digitale Verarbeitung von Audiodaten 13

KAPITEL 2 Einleitung 17

2.1 Das BERKOM-Projekt Multimedia Collaboration (MMC) 17
2.1.1 Anforderungen an MMC 18
2.1.2 Vergleich mit anderen Systemen 20
2.2 Architektur von MMC 21
2.3 Die AV-Komponente (AVC) von MMC 23
2.4 Entwurf und Architektur der AVC 27
2.5 Zusammenfassung 29

KAPITEL 3 Anforderungen an das Audio-Subsystem von MMC 31

3.1 Grundlegende Anforderungen 31
3.2 Nutzung innerhalb von MMC 32
3.2.1 Endpoint Control Protocol 32
3.2.2 AV-Protokoll 32

3.2.3 Transportprotokolle 32
3.2.4 Echtzeiteigenschaften bei Live-Audio 33

3.3 Verwendung durch externe Audioquellen 34

3.3.1 Anwendungsschnittstelle 34
3.3.2 Echtzeitanforderungen 34
3.3.3 Flußkontrolle 34
3.3.4 AV-Übertragung ohne AVXP 35

KAPITEL 4 Vorgehensmodell 37

4.1 Anforderungen des Projekts an das Vorgehensmodell 37

4.2 Evolutionäres Prototyping 38

KAPITEL 5 Plattformen und Werkzeuge 41

5.1 Hardware-Plattform 41

5.2 Software-Plattform 41

5.2.1 Betriebs- und Windowsystem 41
5.2.2 Schnittstelle zur Audio-Hardware 42
5.2.3 Tcl-Interpreter 45

5.3 Benutzte Werkzeuge 46

5.3.1 Programmiersprache 46
5.3.2 Compiler 46
5.3.3 Debugger 47
5.3.4 Versions- und Konfigurationsverwaltung 47
5.3.5 Texteditor 47

5.4 Zusammenfassung 49

KAPITEL 6 Frühe Prototypen 51

6.1 IFA-Prototyp 51

6.1.1 Architektur / Implementierung 51
6.1.2 Nützlichkeit und Einschränkungen 52

6.2 Der zweite Prototyp: AMServer 52

6.2.1 Anforderungen 53
6.2.2 Redesign der AVC 53
6.2.3 Hardware-Schnittstelle: AIO 53
6.2.4 Architektur 54
6.2.5 Erfahrungen mit dem AMServer 57
6.2.6 Zusammenfassung 60

KAPITEL 7 Entwurf der AMEngine 61

7.1 Erweiterte Anforderungen an die AMEngine 61

7.2 Entwurf 61

7.2.1 Entwurfsentscheidungen 61
7.2.2 Architektur 65
7.2.3 Schnittstellen der Module 67
7.2.4 Aufrufoptionen 90

7.3 Testumgebung 91

7.3.1 Tests der Audio-Schnittstelle AIO 91
7.3.2 Standalone-Tests der AMEngine 92
7.3.3 Integrationstests mit der AVC und MMC 93
7.4 Zusammenfassung **93**

KAPITEL 8 Erfahrungen und Ausblick **95**

8.1 Allgemeine Erfahrungen **95**
8.2 Erfüllen der Basisfunktionalität **95**
8.3 Eigenschaften der AMEngine **96**
8.3.1 Erweiterbarkeit um andere Transportprotokolle 96
8.3.2 Vermeidung von Aussetzern 96
8.3.3 Beibehalten der Effizienz und des guten Delay-Verhaltens 98
8.3.4 Verbesserung der Stabilität 98
8.3.5 Verbesserung der Debugging- und Monitoring-Möglichkeiten 98
8.3.6 Verbesserung der Wartbarkeit und Erweiterbarkeit sowie der Wiederverwendbarkeit einzelner Module. 99
8.3.7 Anwendungsmöglichkeiten außerhalb von MMC 100
8.4 Weiterentwicklung der AMEngine **100**
8.4.1 Spatial Mixing 100
8.4.2 Audioformate 100
8.4.3 Gemeinsame Datenübertragung von Audio- und Videodaten 101
8.4.4 Zusammenfassung 102

KAPITEL 9 Literaturverzeichnis **103**

ANHANG A Reference Manual Page **107**

ANHANG B Nutzung der AMEngine durch externe Anwendungen **119**

ANHANG C Audio/Video Exchange Protocol **127**

ANHANG D Endpoint Control Protocol **129**

Begriffe

In diesem Kapitel definiere ich einige der in dieser Arbeit verwendeten Begriffe. Die meisten davon werden im Text zwar eingeführt, werden aber darüber hinaus hier schon einmal zusammengefaßt vorgestellt.

Application Sharing, ASC

Die *Application Sharing Component* (ASC) ermöglicht den Konferenzteilnehmern das gemeinsame Benutzen von Programmen (*Application Sharing*) und damit das gemeinsame Bearbeiten von Dokumenten. Dabei startet ein Konferenzteilnehmer eine Applikation, die er dann den anderen auf deren Bildschirmen mit Hilfe der ASC sichtbar macht. Das Recht, Eingaben in die Applikation zu machen (*floor control* [Malm 1994]), kann an alle oder einzelne Teilnehmer vergeben werden.

ATM

Der *Asynchronous Transfer Mode* (ATM) ist die Vermittlungstechnologie und das Übertragungsverfahren der B-ISDN-Hochgeschwindigkeitsnetze. ATM verwendet eine verbindungsorientierte Übertragungstechnik und ermöglicht Bandbreitenreservierung. Für die Datenübertragung über ATM-Netze werden spezielle Programmierschnittstellen (ATM Adaptation Layer, AAL) benutzt. Zur Integration von ATM-Netzen in bestehende Netzwerke werden auch IP-Pakete über ATM übertragen.

AVC

Die Audio-/Video-Komponente (*Audio Video Component*, auch AV-Komponente) ist der Teil der MMC-Applikation, der für die Bearbeitung der Audio- und Video-Datenströme auf dem lokalen Rechner zuständig ist. Die AVC wird vom *AV-Manager* (AVM) der Videokonferenz gesteuert und baut auf dessen Anweisung Audio- und Video-

Datenverbindungen zu den anderen Konferenzteilnehmern auf beziehungsweise ab.

AVM

Der *Audio-/Video-Manager* ist die Instanz in einer MMC-Konferenz, die die AV-Komponenten der einzelnen Konferenzteilnehmer Audio- und Video-Datenverbindungen untereinander aufbauen läßt. Beim Aufbau werden die Präferenzen und Fähigkeiten der einzelnen AV-Komponenten bezüglich der Art dieser Verbindungen im Hinblick auf die Datenformate und AV-Qualitäten berücksichtigt.

BSD

Das ursprünglich von AT&T entwickelte Betriebssystem UNIX™ wurde seit Ende der siebziger Jahre an der University of California at Berkeley im Rahmen der Forschung der Computer Science Research Group (CSRG) weiterentwickelt. Im Auftrag des amerikanischen Verteidigungsministeriums wurden die im Internet verwendeten Protokolle in das System integriert. Die daraus entstandene „Berkeley Software Distribution“ (BSD) hatte wegen der Netzwerk-Integration und wegen anderer Erweiterungen einen erheblichen Einfluß auf die Weiterentwicklung von Unix-artigen Betriebssystemen. [Leffler et al. 1989]

CCITT

Comite Consultatif International Telegraphique & Telephonique – Vorläufer der ITU.

Endpoints

Der Begriff der Endpoints spielt eine wichtige Rolle Im Modell der audiovisuellen Kommunikation des Konferenzsystems *Multimedia Collaboration* (MMC). Es gibt *Source Endpoints* (Quellen) und *Sink Endpoints* (Senken) für audiovisuelle Datenströme. Jeder Source Endpoint kann mehrere Datenströme aussenden, jeder Sink Endpoint einen Datenstrom empfangen. Die Datenströme können Audiodaten oder Videodaten oder gemischte Daten (zur Zeit noch nicht benutzt, zum Beispiel H.320-Ströme) enthalten.

G.711

G.711 ist ein Standard zur digitalen Übertragung von Audiodaten. Er wird vor allem im Telefonbereich eingesetzt. [ITU-T G.711]

GSM

GSM ist ein Standard zur digitalen Übertragung von Audiodaten. Er wird vor allem in Mobiltelefonnetzen eingesetzt. [ETSI GSM 6.10]

IFA

Die Internationale Funkausstellung (IFA) findet alle zwei Jahre in Berlin statt. Auf ihr werden Exponate aus den Bereichen Unterhaltungselektronik und Telekommunikation gezeigt. In den letzten Jahren hat unter dem Stichwort „Multimedia“ die Computertechnik in beiden Bereichen sehr stark an Bedeutung gewonnen.

IP

Das Internet Protocol (IP) ist das Netzwerkprotokoll der Internet-Protokoll-Suite. Es ist in Schicht drei (Network Layer) des ISO-OSI-Referenzmodells einzuordnen. Darauf aufsetzend gibt es eine Reihe von Transportprotokollen; besondere Bedeutung haben TCP und UDP. [Comer 1988, Tanenbaum 1989]

ISDN

Integrated Services Digital Network (ISDN) ist der Begriff für die von den Telekommunikations-Organisationen angebotenen digitalen Netze im Schmalband-Bereich, das heißt, bis zu einer übertragungsleistung von 2 MBit/s. Der Standard-ISDN-Anschluß (S0) bietet zur Datenübertragung zwei sogenannte B-Kanäle mit je 64 KBit/s.

ISO-OSI-Referenzmodell

Zur Unterstützung ihrer Standardisierungsaktivitäten im Bereich der Netzwerkprotokolle hat die International Organization for Standardization (ISO) das „ISO OSI (*Open Systems Interconnection*) Reference Model" entwickelt. Dieses Modell unterteilt Kommunikation und ihre Anwendungen in sieben aufeinander aufbauende Schichten. Es wird daher auch oft „Sieben-Schichten-Modell" genannt. [Tanenbaum 1989]

ITU, ITU-T

Die *International Telecommunications Union* ist der weltweite Zusammenschluß der Telekommunikations-Organisationen. ITU-T, früher CCITT genannt, ist das Normungsgremium der ITU.

MMC

Multimedia Collaboration (MMC) ist das im Projekt „Multimediale Teledienste" von den einzelnen Projektpartnern entwickelte multimediale Konferenzsystem. MMC stellt den Konferenzteilnehmern audiovisuelle Kommunikation und das gemeinsame Benutzen von Programmen (*Application Sharing*) zur Verfügung.

MMT

Multimedia Transport (MMT) ist ein Projekt im Rahmen „Multimediale Teledienste", in dem Implementierungen für Multimedia-Datenübertragung besonders geeigneter Protokolle entwickelt und erprobt werden. Wichtige Ziele sind hier

- gute Ausnutzung der verfügbaren physikalischen Netzbandbreite,
- geringer Protokoll-Overhead,
- gesicherte Übertragung in der richtigen Reihenfolge ohne fehlende oder duplizierte Daten und
- Zusicherung von Bandbreiten und Übertragungsqualitäten (*Quality Of Service*-Parameter).

POSIX, POSIX.1

POSIX.1 ist als IEEE Standard 1003.1-1990 als internationaler Standard ISO/IEC 9945-1:1990 verabschiedet worden. [Lewine 1994, ISO 1990]
„POSIX is a family of standards developed by the IEEE (Institute of

Electrical and Electronics Engineers). POSIX stands for Portable Operating System Interface for Computing Environments. It originally referred to the IEEE Standard 1003.1-1988 (the operating system interface), but the IEEE is currently working on other related standards in the POSIX family.“ [Stevens 1992]

Tcl

Die „Tool Command Language“ wurde als Skript- und Erweiterungssprache für Applikationen im Sprite-Projekt an der University of California at Berkeley entwickelt. Der Tcl-Interpreter ist zur Einbettung in andere Programme vorgesehen. [Ousterhout 1990, 1994]

TCP

Das *Transmission Control Protocol* (TCP) ist ein Transportprotokoll aus der Internet-Protokoll-Suite. Es ist in Schicht vier (*Transport Layer*) des ISO-OSI-Referenzmodells einzuordnen. TCP bietet eine gesicherte Datenübertragung eines Byte-Stroms in der richtigen Reihenfolge ohne fehlende oder duplizierte Daten. Paketgrenzen werden dabei nicht erhalten. Zusicherungen bezüglich der verfügbaren Bandbreite oder der Übertragungsdauer werden nicht gemacht. [Comer 1988, Tanenbaum 1989]

UDP

Das *User Datagram Protocol* (UDP) ist ein Transportprotokoll aus der Internet-Protokoll-Suite. Es ist in Schicht vier (*Transport Layer*) des ISO-OSI-Referenzmodells einzuordnen. UDP bietet keine gesicherte Datenübertragung; die gesendeten Pakete können verloren gehen, dupliziert werden oder in falscher Reihenfolge ankommen. [Comer 1988, Tanenbaum 1989]

Unix

Wenn ich in dieser Arbeit den Begriff „Unix“ verwende, meine ich damit die Vielfalt von Betriebssystemen, die aus dem ursprünglich bei AT&T entwickelten System UNIX™ hervorgegangen oder solchen Systemen nachempfunden sind. UNIX ist eine Handelsmarke der X/Open Ltd.

X11

Das am MIT unter Beteiligung von Industriepartnern entwickelte Window-System „X“ bildet in seiner Version 11 als technische Plattform den Industriestandard für Window-Systeme auf Unix-Workstations. Seine wesentlichen Merkmale sind Hardware-Unabhängigkeit (X11 ist für nahezu jede Workstation-Hardware verfügbar) und Netzwerktransparenz: Ein Programm, das auf einem Rechner aktiv ist, kann auf einem anderen seine Graphikausgaben ausführen.

KAPITEL 1

Digitale Verarbeitung von Audiodaten

Das in dieser Arbeit beschriebene System verarbeitet digitalisierte Klänge. Die digitale Verarbeitung von Klängen verläuft grundsätzlich in fünf Schritten:

1. Klänge sind zunächst einmal Schalldruckschwankungen. Diese Schalldruckschwankungen werden durch ein Mikrophon in elektrische Schwingungen (analoge Signale) umgewandelt (siehe Bild).
2. Ein Analog-Digital-Wandler wandelt die analogen elektrischen Schwingungen in digitale Audiodaten um.
3. Die Daten können nun von einem Digitalrechner verarbeitet werden.
4. Ein Digital-Analog-Wandler wandelt die digitalen Audiodaten in elektrische Schwingungen zurück.
5. Ein Lautsprecher wandelt die elektrischen Schwingungen wieder in Schalldruckschwankungen und damit in Klänge um.

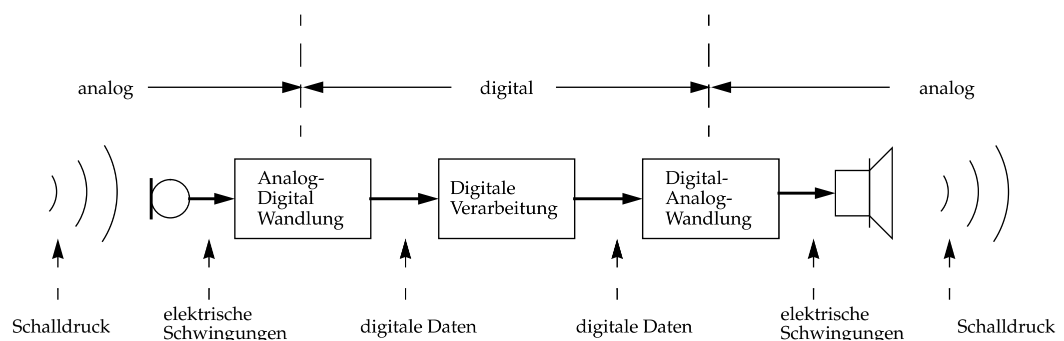

ABBILDUNG 1 Schritte der digitalen Audio-Datenverarbeitung

Die Im Projekt eingesetzten Rechner haben eingebaute Analog-Digital-Wandler (auch AD-Wandler genannt) und Digital-Analog-Wandler (DA-Wandler), die die Schritte 2 und 4 übernehmen.

Das Analogsignal ist sowohl wert- als auch zeit*kontinuierlich*, die digitale Darstellung des Signals nach der AD-Wandlung dagegen sowohl wert- als auch zeit*diskret*. Das digitalisierte Signal ist somit immer nur eine Annäherung an das ursprüngliche Signal. Die Qualität dieser Annäherung hängt von der Auflösung der Abtastung im Werte- und im Zeitbereich ab, also von dem Wertevorrat der Abtastwerte (*Samples*), der durch das verwendete Datenformat vorgegeben ist, und von der Abtastrate.

Die Abtastrate, die Größe der Samples in Bit und die Anzahl der übertragenen Kanäle bestimmen die Datenrate bei der Übertragung digitaler Audiodaten:

Datenrate [Bit/s] = ***Abtastrate*** [1/s] * ***Wortbreite*** [Bit] * ***Anzahl Kanäle***

In [Rosen, Howell 1991] findet sich eine Einführung in die digitale Verarbeitung von Audiodaten.

Datenformate

TABELLE 1 Verbreitete Formate digitaler Audiodaten

Standard	Abtastrate	Wortbreite	Kanäle	Datenrate
CD	44100 Hz	16 Bit	2	1411200 Bit/s
DAT	48000 Hz	16 Bit	2	1536000 Bit/s
G.711	8000 Hz	8 Bit	1	64000 Bit/s
GSM	8000 Hz	n/a	1	13000 Bit/s

- **CD, DAT**

  Das heute bekannteste digitale Audiomedium ist die Compact Disc (CD), die als Distributionsmedium für Musik die traditionelle Schallplatte mittlerweile fast verdrängt hat. Auf der CD liegen die Daten in zwei Kanälen (linker und rechter Stereokanal) als linear codierte 16-Bit-Samples vor; die Abtastrate beträgt 44,1 kHz.

  Diese Parameter sind für eine subjektiv sehr gute Wiedergabe jeder Art von Musik ausreichend. Die DAT-Norm (Digital Audio

Tape) für digitale Tonaufzeichnung auf Magnetbandkassetten geht mit einer Abtastrate von 48 kHz sogar noch weiter.

- **G.711**

  Bei der reinen Sprachwiedergabe werden an die Wiedergabequalität sehr viel geringere Anforderungen gestellt als bei Musik. Die Erfahrungen mit dem analogen Telefonnetz zeigen zum Beispiel, daß menschliche Sprache schon dann gut verständlich ist, wenn nur der Frequenzbereich von 300 bis 3000 Hz übertragen wird.

  Oft ist die Datenrate sehr viel wichtiger als eine hohe Wiedergabequalität, gerade wenn digitale Audiodaten über schmalbandige Kanäle übertragen werden oder wenn die Übertragungskanäle gleichzeitig für andere Datenübertragungen genutzt werden sollen. Das ist bei der Übertragung der Daten über Weitverkehrsnetze üblicherweise der Fall.

  Ein Beispiel dafür ist das Telefonnetz. Das Telefonnetz der Deutschen Telekom ist mittlerweile weitgehend digital ausgerüstet, auch wenn die Endgeräte (Telefone, Faxgeräte, Modems) noch zum größten Teil mit analogen Signalen arbeiten. Das Telefonnetz ist dafür ausgelegt, eine gute Sprachverständlichkeit bei gleichzeitig nicht zu hohem Bandbreitenbedarf zu gewährleisten.

  Bei der digitalen Übertragung von Sprache im Telefonnetz wird eine Abtastrate von 8 kHz mit µlaw-codierten 8-Bit-Samples benutzt. Dieses Datenformat ist als G.711 von der International Telecommunications Union (ITU) standardisiert. [ITU-T G.711] Bei der ulaw-Kodierung, werden 13 Bit breite linear codierte Samples in logarithmisch codierte 8-Bit-Samples gewandelt. Das ist nicht ohne Qualitätsverlust möglich, bietet aber für eine Sprachverständlichkeit ausreichende Qualität.

- **GSM**

  Für Funktelefonnetze, in denen die verfügbare Bandbreite besonders gering ist, wurden verlustbehaftete Kompressionsverfahren entwickelt, die die zu übertragende Datenmenge stark reduzieren, aber die Sprachverständlichkeit bewahren. Hier ist vor allem der Standard GSM 6.10 zu nennen, der in vielen Funktelefonnetzen zum Einsatz kommt [Degener 1994, 1996; ETSI GSM 6.10].

  Im allgemeinen Fall von Datenkompression werden Algorithmen benutzt, die unter Ausnutzung der Entropie der Daten zum Beispiel durch Huffman- oder LZW-Kodierung im Schnitt Kompressionsraten von 40 - 60% erreichen.

  Bei Audiodaten erreichen diese Algorithmen nur schlechte Resultate. Hier ist es jedoch nicht unbedingt wichtig, die Daten unverändert wiederherstellen zu können. Gerade bei Sprachübertragung kommt es hauptsächlich auf gute Sprachverständlichkeit an, nicht unbedingt auf eine besonders hohe Wiedergabequalität.

KAPITEL 2 Einleitung

2.1 Das BERKOM-Projekt Multimedia Collaboration (MMC)

Einordnung

Das Projekt *BERKOM* (BERliner KOMmunikationssystem) ist ein von der Deutschen Telekom initiiertes Projekt zur Erprobung von Kommunikationstechniken in Breitband- und ISDN-Netzen. In der zweiten Phase des Projekts, die Ende 1994 nach fünf Jahren auslief, konzentrierten sich die Aktivitäten auf eine Kommunikations-Infrastruktur für Multimedia-Applikationen, genannt „Multimediale Teledienste“. Diese Phase beinhaltet drei Arbeitsgebiete:

- Der *BERKOM Multimedia Transport Service* (MMT) stellt das Transportsystem für die audiovisuelle Kommunikation dar. Er basiert auf ST-II, dem *Internet Stream Protocol*, das Verbindungen zwischen mehreren Endpunkten mit Durchsatz- und Laufzeit-Zusicherungen erlaubt. Das Transportsystem kann eine Vielzahl von Netzwerktechnologien nutzen, darunter unter anderem Ethernet, FDDI und ATM.
- Der *BERKOM Multimedia Mail Service* (MMM) ermöglicht das Verschicken von Multimedia-Dokumenten mit Audio- und Video-Annotationen. Er ist geplant als eine standardkonforme Erweiterung zu den Funktionalitäten von X.400 und ODA und kann Dokumente mit MIME, dem Multimedia-Mail-Format des Internet, austauschen.
- Der *BERKOM Multimedia Collaboration Service* (MMC) unterstützt gemeinsames Arbeiten in einer verteilten Umgebung. Er erlaubt den Benutzern, ihre Applikationen auf verschiedenen Arbeits-

plätzen gemeinsam zu bedienen und miteinander audiovisuelle Konferenzen zu führen.

[Altenhofen et al. 1993]

In der anschließenden dritten Phase des BERKOM-Projekts werden die Aktivitäten zu MMC fortgesetzt und zielen darauf ab, die Ergebnisse in Produkte der beteiligten Partner zu überführen.

An der Technischen Universität Berlin nimmt der Forschungsschwerpunkt „Prozeßdatenverarbeitung im Prozeßrechnerverbund / Prozeßrechnerverbundzentrale“ (FSP-PV / PRZ) als Subauftragnehmer von Hewlett-Packard am Projekt Multimedia Collaboration teil. In Kooperation mit der Firma Hewlett-Packard (HP) wurde eine Implementierung auf HP-Workstations entwickelt. Dazu entwickelte ich das in dieser Arbeit beschriebene Audio-Subsystem.

Wenn ich im folgenden von „wir“ beziehungsweise „uns“ spreche, meine ich damit die MMC-Arbeitsgruppe in der PRZ der TU Berlin.

2.1.1 Anforderungen an MMC

Unter der Projektleitung der DeTeBerkom, einer Tochter der Deutschen Telekom, arbeiten mehrere Firmen und Forschungseinrichtungen an der Implementierung von MMC für ihre jeweilige Plattform. Das Ziel der DeTeBerkom und ihrer Partner ist die Entwicklung eines Service für die Büroumgebung von großen, geographisch verteilten Organisationen, wie multinationalen Firmen, oder Behörden mit verschiedenen Standorten, wie zum Beispiel der deutschen Bundesregierung, die in Zukunft auf Standorte in Bonn und Berlin verteilt sein wird. In einem solchen Szenario wird der Einsatz von Breitband-WANs und heterogenen LANs nötig, um innerhalb der Behörden eine große Zahl von Arbeitsplätzen zu vernetzen.

Besonders wichtig ist dabei die Spezifikation und Realisierung einer offenen homogenen Umgebung für kooperative Arbeit in einem Verbund aus verschiedenen Hardware-Plattformen. Um dies zu erreichen, folgt der Entwurf zwei wesentlichen Kriterien:

- Benutzung internationaler Standards (ISO, ITU-T), wenn möglich
- Aufteilung des Systems in Komponenten mit definierter Funktionalität und Schnittstelle.

MMC ist eine Applikation aus dem Bereich Computer Supported Collaborative Work (CSCW), die Videokonferenzen zwischen meh-

reren Benutzern ermöglicht. Kollaboratives Arbeiten wird von MMC durch drei wesentliche Funktionalitäten unterstützt:

1. *Application Sharing*: Durch das gemeinsame Benutzen von Programmen können von den Konferenzteilnehmern Dokumente gemeinsam bearbeitet werden. Das Recht, Eingaben an ein gemeinsam benutztes Programm zu schicken, wird einzeln zugeteilt oder allen Teilnehmern gegeben. Teilnehmer können Markierungen auf die Fenster des verteilten Programms setzen (Telemarker) und die Bewegungen ihres Mauszeigers für andere sichtbar machen (Telepointer).
2. *Audiovisuelle Kommunikation*: Die Konferenzteilnehmer sehen und hören sich gegenseitig. Dabei können je nach Hard- und Software-Ausstattung der Teilnehmer in einer Konferenz auch Verbindungen mit unterschiedlichen Audio- und Video-Qualitäten gleichzeitig existieren.
3. *Konferenz- und Benutzermanagement*: Zu einer Konferenz werden einzelne Teilnehmer oder vorher definierte Gruppen von Teilnehmern eingeladen. Verschiedene Konferenz-Modi ermöglichen die Einschränkung der Konferenz auf bestimmte Teilnehmergruppen sowie verschiedene Strategien zum Umgang mit Teilnehmern, die ohne vorhergehende Einladung an der Konferenz teilnehmen wollen (*Latecomer*). Verschiedene Rollen geben den Teilnehmern je nach Modus der Konferenz unterschiedliche Rechte. In einem Verzeichnis werden Informationen über Benutzer, Gruppen von Benutzern und Konferenzen verwaltet.

[Altenhofen et al. 1993; Gleser, Kückes, Nickelsen 1994]

Diese Funktionalität soll auf einer möglichst großen Spanne von Netzwerken genutzt werden können. Dabei soll die *Quality of Service* dem genutzten Medium sinnvoll angepaßt werden. Auf breitbandigen Übertragungsmedien, wie zum Beispiel FDDI- und ATM-Leitungen mit 100 MBit/s und mehr, sollen die vorhandenen Kapazitäten zur Übertragung von hochqualitativem Video und Audio genutzt werden (Bildwiederholrate, Bildgröße, Bildqualität; Frequenzgang, Dynamikumfang). MMC soll aber auch auf Leitungen geringer Kapazität bis hin zu ISDN einsetzbar sein. Die Video- und Audioqualität ist dabei entsprechend anzupassen.

MMC ist als plattformübergreifende Anwendung konzipiert, wobei jeder Projektpartner eine eigene Implementierung für seine Plattform entwickelt:

- IBM RS/6000 Workstation (IBM, Heidelberg)
- Digital Alpha Workstation (Digital Equipment, Karlsruhe)
- Siemens RM Workstation (Sietec, Berlin)

- Sun SPARCstation (GMD FOKUS, Berlin)
- HP 9000/700 Workstation (TU Berlin; Hewlett-Packard, Böblingen)
- Intel-PC mit MS-Windows (Liebing & Ullfors, Berlin)
- Apple Power Macintosh (MacConsult, Berlin)

Die verschiedenen Implementierungen sind dabei interoperabel. Bestandteil des Projektplans für 1996 ist die Verfügbarkeit von MMC als kommerzielles Produkt.

Für 1996/97 ist die Interoperabilität mit nicht-MMC-fähigen Systemen wie H.320-Bildtelefonen und ISDN-Telefonen angestrebt. Dabei wird entsprechend den Fähigkeiten der Endgeräte nicht der volle Funktionsumfang von MMC abgedeckt. [ITU-T H.320]

2.1.2 Vergleich mit anderen Systemen

JVTOS

Im Rahmen des RACE-Projekts CIO (RACE 2060) wurde in Zusammenarbeit mit anderen Forschungsinstituten an der TU Berlin der Joint Viewing and Teleoperation Service (JVTOS) entwickelt. Konzeptionell ist JVTOS MMC in vielen Punkten sehr ähnlich. Die Hauptunterschiede zu MMC sind:

- JVTOS legt einen Schwerpunkt auf die Übersetzung von Datenströmen beim Übergang auf andere Plattformen zur Interoperabilität zwischen verschiedenen Window-Systemen (X11, Macintosh, MS-Windows) und Audio/Video-Formaten. Bei MMC wird dagegen von einem zentralen Protokoll für das Window-System (X11) ausgegangen. Um in MMC eine AV-Verbindung zwischen zwei Teilnehmern aufbauen zu können, muß dagegen ein beiden Teilnehmern gemeinsames Audio- bzw. Video-Format gefunden werden.
- Der Auf- und Abbau von Konferenzen erfolgt bei JVTOS dezentral; die dazu benötigten Komponenten laufen auf jedem Rechner.
- JVTOS ist auf die Nutzung breitbandiger Übertragungsmedien (zum Beispiel FDDI, ATM) ausgelegt. MMC kann dagegen auch Medien geringer Übertragungskapazität bis hin zu Schmalband-ISDN nutzen.

[Dermler, Froitzheim 1992; Dermler et al. 1993, 1994; Gehring 1996]

H.320

H.320 ist der ITU-Standard für Bildtelefonkommunikation über ISDN-Leitungen. Hier werden typischerweise wenige B-Kanäle (meistens zwei) zu je 64 KBit/s für die Übertragung von Audio und Video und eines zusätzlichen, nicht näher spezifizierten Datenkanals benutzt. H.320-Endgeräte sind vorwiegend spezielle Bildtelefone oder Personal Computer. Hauptunterschiede zu MMC:

- H.320 ist auf spezielle Audio- und Videoformate festgelegt (G.711, G.722, G.728; H.261) und kann deshalb nur begrenzt höhere Übertragungskapazitäten nutzen.
- Andere Übertragungsmedien als ISDN sind bei H.320 nicht vorgesehen.
- Mit H.320-Endgeräten sind direkt nur Konferenzen zwischen *zwei* Teilnehmern möglich. Für Konferenzen mit mehreren Teilnehmern verbinden sich die jeweiligen Endgeräte mit sogenannten *Multipoint Control Units* (MCUs), die für jeden Teilnehmer durch Mischen oder Umschalten das Signal der jeweils anderen Teilnehmer aufbereiten.
- Eine Möglichkeit zum Application Sharing ist bei H.320 nicht vorgesehen (und bei reinen Bildtelefon-Endgeräten auch nicht denkbar). Der Datenkanal kann jedoch für nicht standardisierte Erweiterungen benutzt werden.
- Funktionen zum Konferenzmanagement sind bei H.320 nicht vorgesehen. Konferenzen zwischen mehreren Teilnehmern werden durch manuelles Schalten der Konferenz in der MCU und Einwählen der Teilnehmer in die MCU eröffnet.

[ITU-T H.320]

2.2 Architektur von MMC

MMC ist als verteiltes System angelegt. Komponentenverschiedener Herstellern sind dabei kombinierbar, da die Schnittstellen zwischen den Komponenten spezifiziert sind [Altenhofen et al. 1993].

- Der *Conference Manager* (CM) ist das zentrale Modul zur Initiierung und Verwaltung von Konferenzen. Es gibt mindestens einen CM im System; typischerweise existiert ein CM pro Konferenz.
- Das *Conference Directory* (CD) ist die zentrale Datenbank für alle Informationen über Benutzer, Gruppen und Konferenzen. Es gibt mindestens ein CD im System.

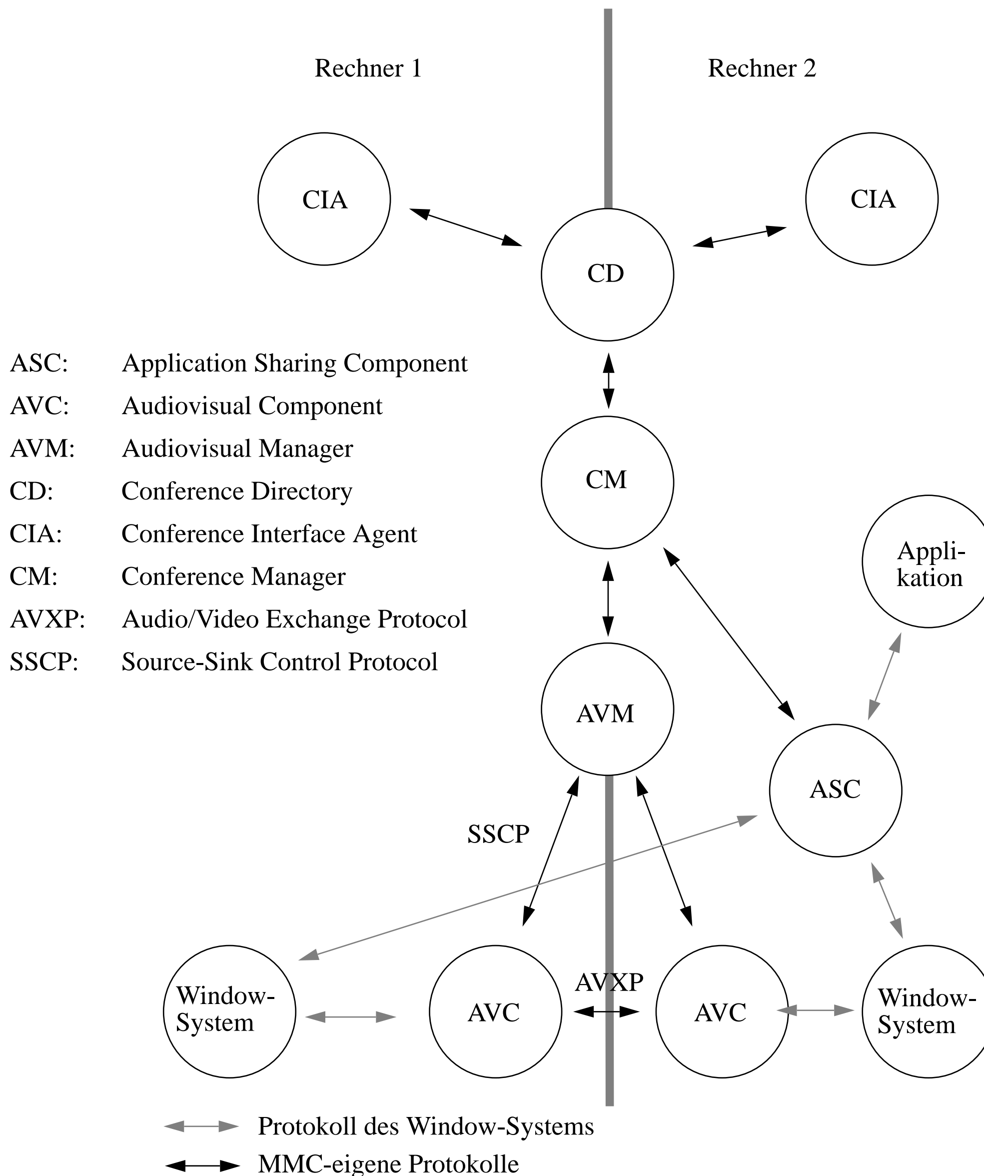

ABBILDUNG 2 Übersicht über die Architektur von MMC

- Der *Audiovisual Manager* (AVM) koordiniert die Einrichtung von Kommunikationsverbindungen zwischen den Konferenzteilnehmern. Es gibt pro Konferenz einen AVM.
- Die *Application Sharing Component* (ASC) ermöglicht das gleichzeitige Darstellen und Benutzen von Applikationen für mehrere Teilnehmer. Es gibt pro Konferenzteilnehmer eine ASC.

- Der *Invitation Broker* (IB, im Bild nicht dargestellt) meldet dem Benutzer das Eintreffen einer Einladung. Es gibt für jeden Konferenzteilnehmer einen IB.
- Der *Conference Interface Agent* (CIA) dient als Benutzungsschnittstelle zu MMC. Es gibt für jeden Konferenzteilnehmer einen CIA.
- Die *Audiovisual Component* (AVC) stellt Audio- und Videoverbindungen zwischen den einzelnen Konferenzteilnehmern her; dazu benutzt sie die Audio- und Video-Hardware der lokalen Workstation. Es gibt für jeden Konferenzteilnehmer eine AVC.

Diese Komponenten kommunizieren miteinander über MMC-eigene Protokolle. Die meisten dieser Protokolle basieren auf *Remote Procedure Calls* (RPCs) und benutzen die ISO/ROSE-Schnittstelle des ISODE-Pakets. Zusätzlich benutzt die ASC für das Application Sharing das Protokoll des Window-Systems (X11).

Die einzelnen Instanzen der AVC kommunizieren miteinander über das Audio/Video Exchange Protocol (AVXP). Dieses benutzt aus Effizienzgründen nicht die ISODE-Schnittstelle, sondern direkt die Transportschicht des darunter liegenden Netzwerkprotokolls.

2.3 Die AV-Komponente (AVC) von MMC

Die AVC ist die Komponente des MMC-Systems, die dafür sorgt, daß sich die Teilnehmer einer MMC-Konferenz gegenseitig hören und sehen können. Dazu wird das Bild eines Teilnehmers mit einer Videokamera aufgenommen, seine Sprache mit einem Mikrophon. Das Bild und der Ton werden digitalisiert und als audiovisuelle Datenströme an die Rechner der anderen Konferenzteilnehmer geschickt. Hier werden die Video- und Audiodaten als Bild und Ton wiedergegeben.

Endpoints

In einer abstrakten Betrachtung besteht das audiovisuelle Kommunikationssystem von MMC aus einer Anzahl von *Endpoints*, die durch gerichtete Datenströme miteinander verbunden sind; die Richtung der Datenströme verläuft von den Quellen (*Source Endpoints*) zu den Senken (*Sink Endpoints*). Ein Source Endpoint kann mit mehreren Sink Endpoints verbunden sein, ein Sink Endpoint dagegen nur mit einem Source Endpoint (1:n-Relation).

Endpoints können Videodaten oder Audiodaten oder beides senden bzw. empfangen.

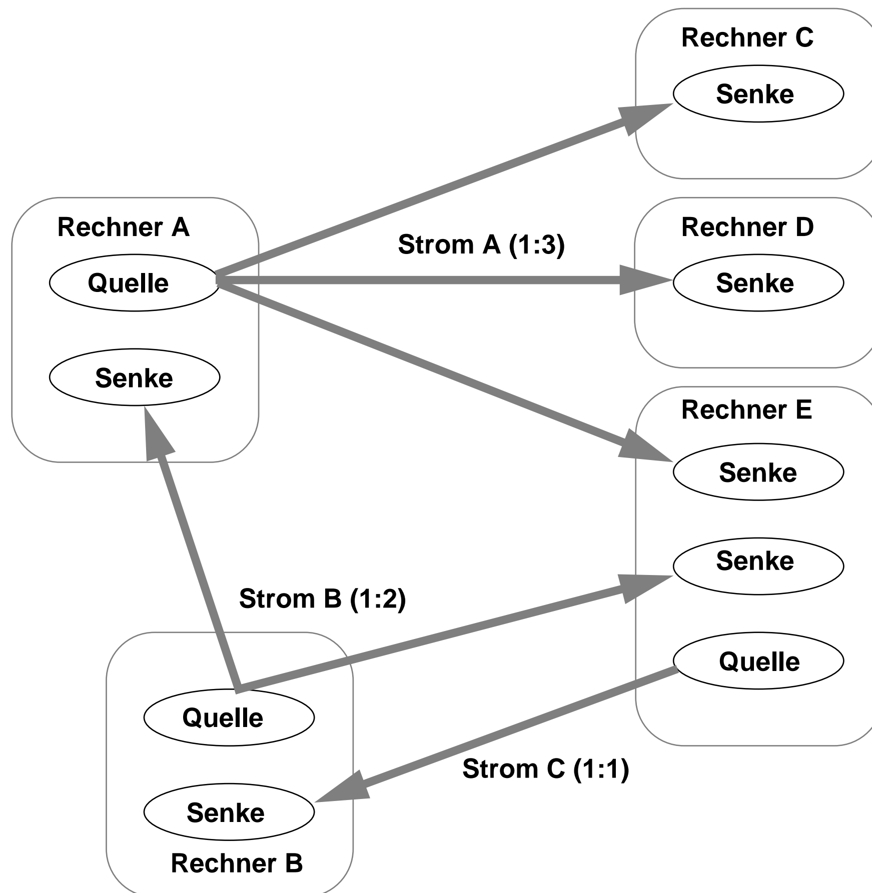

ABBILDUNG 3 Kommunikationsströme als Relationen zwischen Endpunkten

Jeder Endpoint hat bestimmte Eigenschaften und Verhaltensweisen. Kommunikation ist nur dann möglich, wenn in diesen Eigenschaften zueinander passende Endpoints miteinander verbunden sind. In MMC stehen die Datenströme unter der Kontrolle des Audiovisual Managers (AVM), der das System durch Auf- und Abbau der Verbindungen steuert.

Endpoints existieren nur zeitweilig. Das gleiche gilt für die Relationen zwischen den Quellen und Senken des Kommunikationssystems; sie werden nach Bedarf erzeugt und wieder gelöscht. Zusätzlich kann eine Relation während ihrer Lebensdauer um Senken erweitert beziehungsweise verringert werden. Die Lebensdauer

einer solchen Relation ist dabei begrenzt durch die Lebensdauer der beteiligten Quelle. Solange Senken in der Relation verbleiben, wird die Lebensdauer der Relation von Senken nicht beeinflußt; die Entfernung der letzten verbliebenen Senke bewirkt jedoch die Löschung der Relation. Die Lebensdauer der an der Relation beteiligten Endpoints wird durch die Lebensdauer der Relation allerdings nicht beeinflußt.

Im Gegensatz zu den nur zeitweilig existenten Endpoints und ihren Relationen gibt es in diesem System auf jedem Rechner eine Instanz, die während der ganzen Lebensdauer des Kommunikationssystems existiert. Diese Instanz wird vom AVM angesprochen und ist verantwortlich für die Erzeugung, Manipulation und Löschung von Endpoints. Diese Instanz wird Audiovisuelle Komponente (*Audiovisual Component*, AVC) genannt. Auf jedem Rechner gibt es genau eine AVC, die alle Endpoints dieses Systems verwaltet.

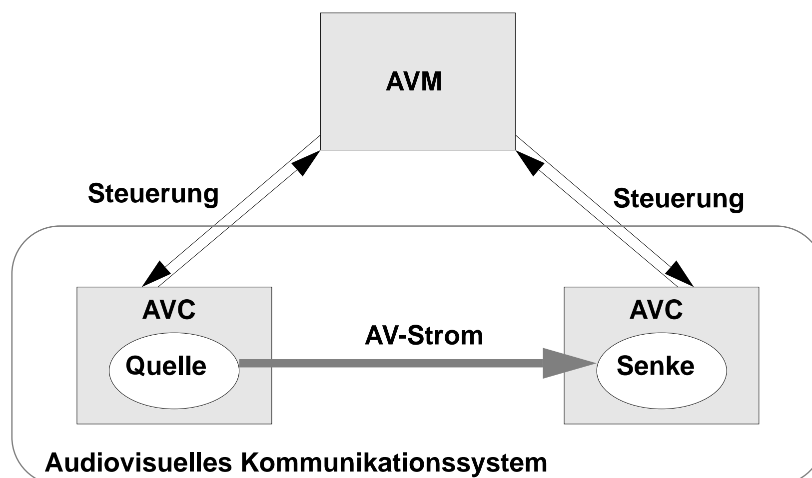

ABBILDUNG 4 Rolle des Audiovisual Managers

Schnittstelle zwischen AVC und AVM

Der AVM spricht die AVC über das *Source/Sink Control Protocol* (SSCP) an, das im MMC-Projekt für diesen Zweck spezifiziert wurde [Altenhofen et al. 1993]. Das SSCP enthält folgende Requests:

SSCP_Bind

Der AVM autorisiert sich gegenüber der AVC. Durch ein übergebenes Paßwort weist der AVM seine Berechtigung nach, die AVC zu steuern.

SSCP_Unbind
Der AVM meldet sich bei der AVC ab. Die AVC schickt daraufhin alle noch ausstehenden Events an den AVM. Die Autorisierung des AVM wird daraufhin als ungültig angesehen.

SSCP_ListEndpointAVTypes
Die AVC listet die von ihr unterstützten Audio- und Video-Datentypen auf. Der AVM benötigt diese Informationen, um beim Verbindungsaufbau zwischen verschiedenen AVCs zueinander passende Datentypen auszuwählen.

SSCP_OpenSource
Die AVC richtet einen Source Endpoint ein. Dabei gibt der AVM die Parameter der Datenverbindung an und ob Audiodaten oder Videodaten oder beides übertragen werden sollen.

SSCP_OpenSink
Die AVC richtet einen Sink Endpoint ein. Die vom AVM übergebenen Parameter entsprechen denen bei SSCP_OpenSource.

SSSP_CloseEndpoint
Die AVC schließt den angegebenen Endpoint.

SSCP_AddSinks
Zu jedem Source Endpoint verwaltet die AVC eine Liste von Sink Endpoints anderer AVCs, zu denen der Source Endpoint seine AV-Daten schickt. SSCP_AddSinks fügt dieser Liste Sink Endpoints hinzu, damit der Source Endpoint auch an diese Sink Endpoints seine Daten schickt.

SSCP_RemoveSinks
Die angegebenen Sink Endpoints werden von der Liste des lokalen Source Endpoints entfernt. Es werden dann an diese Sink Endpoints keine Daten mehr geschickt.

SSCP_SetTransmission
Der „Übertragungsstatus“ (*Transmission State*) des angegebenen Endpoints wird auf „on“ oder „off“ gesetzt. Nach dem Einrichten eines Endpoints ist dieser Status initial „off“. Das Umschalten des Status auf „on“ bedeutet für einen Source Endpoint, daß er anfängt, Daten zu senden. Ein Sink Endpoint empfängt zwar auch schon Daten, wenn sein Übertragungsstatus „off“ ist, stellt diese aber nicht auf der lokalen AV-Hardware dar, spielt also empfangene Audiodaten nicht ab und öffnet für Videodaten kein Fenster auf dem Window-System. Erst wenn der Status auf „on“ gesetzt wird, werden die empfangenen Daten dem Benutzer präsentiert.

Das SSCP definiert zwei Events, die die AVC an den AVM schicken kann:

SSCP_EndpointClosed: Der angegebene Endpoint wurde unvorhergesehenerweise geschlossen. Ein zusätzlicher Parameter gibt den Grund dafür an.

SSCP_SinkDropped: Die Verbindung von einem Source Endpoint der AVC zu einem Sink Endpoint auf einem anderen Rechner wurde geschlossen. Ein zusätzlicher Parameter gibt den Grund dafür an.

Übertragung von Audiodaten zwischen verschiedenen Plattformen

Für den Austausch von Audiodaten zwischen den AVCs der Konferenzteilnehmer ist zunächst der ITU-Standard G.711 vorgesehen, das heißt, Audiodaten mit einer Abtastrate von 8 kHz in µlaw-Kodierung, mono. Eine Erweiterung um andere Datenformate ist geplant.

2.4 Entwurf und Architektur der AVC

Da Audio- und Videodaten getrennt voneinander übertragen werden, kann die AVC die Endpoints auf getrennte Audio- und Video-Instanzen abbilden. Da Audio- und Video-Bearbeitung sich in wesentlichen Punkten unterscheiden, erschien es uns für die Verwirklichung der AVC sinnvoll, die Funktionalität in drei Subsysteme zu unterteilen:

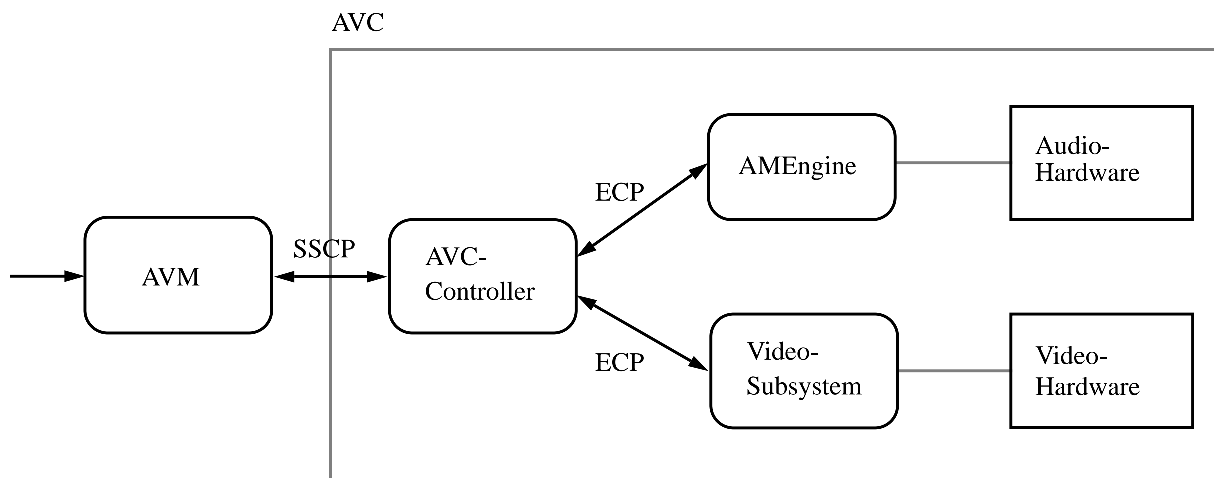

ABBILDUNG 5 Subsysteme der AVC

1. Der *AVC-Controller*, der SSCP-Requests vom AVM entgegennimmt. Je nach Art des Requests wird dieser entweder von ihm selbst bearbeitet oder zur Bearbeitung an eins der anderen Subsysteme weitergeleitet.
2. Das *Video-Subsystem*, das vom AVC-Controller Requests entgegennimmt und bearbeitet, die Video-Verbindungen betreffen.
3. Das *Audio-Subsystem*, genannt *Audio Multiplexer* Engine (AME, auch *AMEngine*), das vom AVC-Controller Requests entgegennimmt und bearbeitet, die Audio-Verbindungen betreffen.

Diese Subsysteme sind als eigenständige Programme ausgeführt. Der AVC-Controller aktiviert Audio- und Video-Subsystem und kommuniziert mit ihnen über das *Endpoint Control Protocol* (ECP), das zu diesem Zweck entwickelt wurde. Nachrichten im ECP bestehen aus Textzeilen, die als Klartext lesbar sind. (Das ECP ist im Anhang „Endpoint Control Protocol“ beschrieben.)

Audio- und Video-Subsystem greifen über das Betriebssystem auf die Audio- beziehungsweise Video-Hardware der Workstation zu.

Diese Aufteilung bietet erhebliche Vorteile:

- Durch die Aufteilung der AVC in einzelne Prozesse mit gut definierten Schnittstellen können diese Teile besser von verschiedenen Personen bearbeitet werden. Der Kommunikationsaufwand bei der Entwicklung der Komponenten der AVC reduziert sich auf die Festlegung von Syntax und Semantik des zwischen den Komponenten verwendeten Protokolls (ECP). Da es im Projekt tatsächlich nötig war, die Arbeit auf mehrere Personen aufzuteilen, kam uns dieser Vorteil sehr gelegen. (Ansgar Kückes implementierte den AVC-Controller und das Video-Subsystem.)
- Die ansonsten sehr große Komplexität des Systems wird so in leichter handhabbare Teile aufgeteilt. Die Wartbarkeit des Systems wird dadurch verbessert.
- Das Verteilen der von außen ankommenden Events auf die einzelnen Komponenten der AVC wird vom Betriebsystem übernommen. Durch die Nutzung der Betriebssystemfunktionalität wird Komplexität in der AVC eingespart und damit die Wartbarkeit verbessert.
- Da die Komponenten als eigenständige Programme entworfen wurden, können Audio- und Video-Subsystem einzeln getestet werden. Das Programm kann unabhängig von MMC gestartet werden, liest dann von der Standardeingabe ECP-Requests, die von Hand oder von einer Datei eingegeben werden können, und

schreibt die Rückgabewerte auf die Standardausgabe. Diese Möglichkeit des Testens erwies sich als außerordentlich wertvoll.
- Die Wiederverwendbarkeit der Komponenten wird durch die Aufteilung erhöht, da Anpassungen an geänderte Gegebenheiten an den einzelnen Komponenten leichter vorgenommen werden können. Eine Verwendung von Audio- und Video-Subsystem außerhalb des MMC-Kontexts ist möglich.

Die Aufteilung in eigenständige Prozesse hat jedoch auch mehrere potentiell nachteilige Konsequenzen:

- Die Verteilung der Rechenzeit zwischen den Subsystemen ist abhängig vom Scheduling des Betriebssystems. Eine explizite Verteilung der Rechenzeit auf die verschiedenen Komponenten der AVC ist so nicht möglich. Die Erfahrung zeigt aber, daß dieser Punkt keine negativen Auswirkungen auf das Funktionsverhalten der AVC hat.
- Das nötige Scheduling zwischen den verschiedenen Prozessen erzeugt einen gewissen Performance-Overhead. Es hat sich jedoch gezeigt, daß dieser Effekt auf heutigen Maschinen dank ihrer Rechenleistung praktisch keine Rolle spielt.
- Durch den indirekteren Zugriff des AVC-Controllers auf Audio- und Video-Subsystem (Kommunikation statt eines einfachen Prozeduraufrufs) ist die Ausführung der Operationen langsamer. Da aber die SSCP-Requests nur verhältnismäßig selten erfolgen (beim Initiieren und Beenden der Konferenz; beim Verbindungsauf- und -abbau), ist dieser Effekt nicht schwerwiegend.
- In einer Videokonferenz ist Lippensynchronität des Tons sehr wünschenswert. Die Synchronisierung zwischen Audio- und Video-Subsystem ist durch die Aufteilung erschwert; sie muß über den AVC-Controller stattfinden. Da Synchronisationsmechanismen zwischen Ton und Bild bisher noch nicht implementiert sind, liegen zu diesem Punkt noch keine Erfahrungen vor.

Wir sehen, daß diese Nachteile bis auf den letzten Punkt zu vernachlässigen sind. Den erhöhten Kommunikationsaufwand zwischen Audio- und Video-Subsystem nehmen wir hin, weil die Vorteile der Aufteilung diesen Nachteil bei weitem überwiegen.

2.5 Zusammenfassung

Das Teledienste-Projekt „Multimedia Collaboration“ (MMC) entwickelt eine multimediale Konferenzumgebung. Innerhalb von MMC ist die Audio/Video-Komponente (AVC) für den Austausch der audiovisuellen Informationen zwischen den Konferenzteilnehmern

zuständig. Das Audio-Subsystem innerhalb der AVC ist als eigenständiger Prozeß realisiert.

Im weiteren wird das Audio-Subsystem der AV-Komponente näher betrachtet.

KAPITEL 3

Anforderungen an das Audio-Subsystem von MMC

3.1 Grundlegende Anforderungen

Das Audio-Subsystem stellt einen Zugang zur Audio-Hardware der Workstation zur Verfügung. Im Vordergrund steht dabei die Benutzung durch MMC. Es soll aber auch möglich sein, während der Laufzeit von MMC von anderen Applikationen, zum Beispiel Multimedia Mail, die Audio-Fähigkeiten der Workstation zu nutzen.

Eine Verwendungsmöglichkeit des Audio-Subsystems auch unabhängig von MMC ist darüber hinaus wünschenswert.

Für beide Verwendungsarten ist es erforderlich, daß mehrere verschiedene eingehende und ausgehende Datenströme gleichzeitig bearbeitet werden können, auch wenn sie Audiodaten verschiedener Kodierung transportieren. Das erfordert die Fähigkeit, eingehende Datenströme zu verschiedener Kodierung zu mischen und wiederzugeben, sowie Datenströme verschiedener Kodierung auszusenden.

Endpoints als eigentliche Sender und Empfänger von Audiodaten können zur Laufzeit des Programms erzeugt und gelöscht werden. Einzelne Endpoints können während ihrer Lebensdauer Verbindungen zu anderen Endpoints aufnehmen und beenden.

3.2 Nutzung innerhalb von MMC

3.2.1 Endpoint Control Protocol

Für die Verwendung mit MMC ist das Endpoint Control Protocol als Protokoll zwischen dem AVC-Controller und dem Audio-Subsystem spezifiziert (siehe Anhang). Mit Hilfe dieses Protokolls schickt der AVC-Controller die Kommandos, die sich aus den an ihn gerichteten SSCP-Requests ergeben, an das Audio-Subsystem (siehe Einleitung).

3.2.2 AV-Protokoll

Für die Übertragung der Nutzdaten zwischen den Endpoints spezifiziert MMC das Audio/Video Exchange Protocol (AVXP). Eine Beschreibung des AVXP befindet sich im Anhang. Im AVXP besteht der Datenstrom aus einer Reihe von Paketen, die jeweils einen Header mit zusätzlichen Informationen enthalten. Für das Audio-Subsystem relevante Felder beinhalten eine Sequenznummer und einen Zeitstempel zur Synchronisation mit einem Videodatenstrom. Die Menge der Audio-Nutzdaten in einem AVXP-Paket ist bis auf eine Obergrenze, die sich aus der Verwendung des Transportprotokolls UDP ergibt, nicht festgelegt.

3.2.3 Transportprotokolle

MMC benutzt als Transportprotokoll für die AV-Daten zur Zeit noch UDP. Langfristig sollen aber auch die im Projekt Multimedia Transport (MMT) entwickelten Protokollimplementierungen genutzt werden. Als zwischenzeitliche Lösung für Anwendungen, die Flußkontrolle im Audio-Strom benötigen, wird TCP als Transportprotokoll genutzt. Eine Übertragung der AV-Daten über ATM unter direkter Benutzung der ATM Adaptation Layer ist für die Zukunft vorgesehen.

Es ist somit erforderlich, das Audio-Subsystem möglichst unabhängig vom verwendeten Transportprotokoll zu halten beziehungsweise die Erweiterung auf weitere Transportprotokolle vorzusehen.

3.2.4 Echtzeiteigenschaften bei Live-Audio

Vermeidung von Aussetzern

In Videokonferenzen ist die Kontinuität des Tonsignals besonders wichtig. Bei gemeinsam benutzten Applikationen, aber auch beim Videobild werden kurze Aussetzer und Verzögerungen toleriert, die sich durch die Belastung der beteiligten Maschinen und Engpässe im Netzwerk ergeben. Beim Tonsignal verschlechtern dagegen schon relativ wenige Aussetzer die Qualität deutlich; die Sprachverständlichkeit leidet stark.

Interessant in diesem Zusammenhang sind Studien, die das European Telecommunication Evaluation Center an der TU Berlin zu Quality-of-Service-Aspekten bei Videokonferenzen durchgeführt hat. Unter anderem wurden in einer Studie die Auswirkungen von Datenverlusten bei Videokonferenzen untersucht. Bei Videoframes wurde von den Versuchspersonen ein Verlust von bis zu 19% noch als „gut“ bewertet, ein Verlust von bis zu 51% als „noch akzeptabel“. Bei Audiopaketen wurde dagegen nur ein Verlust von bis zu 4% als „gut“ bewertet, als nur „noch akzeptabel“ ein Verlust von bis zu 6%. [Kawalek 1994, 1995]

Übertragungszeit des Signals

Ebenfalls von besonderer Bedeutung für Konferenz-Anwendungen ist eine möglichst geringe Verzögerung (Delay) in der Audio-Übertragung. Wie Experimente mit variierten Verzögerungszeiten bei MMC-Konferenzen gezeigt haben, ist eine Verzögerung von etwa einer halben Sekunde noch erträglich. Bei größeren Verzögerungen sind die Reaktionen der Gesprächspartner zeitlich zu weit auseinandergerückt und stören damit den Gesprächsfluß deutlich. Eine geringere Verzögerung in der Audio-Übertragung verbessert die Gesprächssituation.

Im Interesse einer geringen Verzögerung kann die Vollständigkeit des Datenstroms in gewissem Maße vernachlässigt werden. Gehen einzelne Pakete des Audio-Datenstroms verloren, ist es nicht sinnvoll, in diesem Fall den Source Endpoint zu einer wiederholten Sendung des Pakets zu veranlassen, da das eine zusätzliche Verzögerung des Datenstroms verursachen würde. Wie die oben genannten Untersuchungsergebnisse zeigen, kann der Verlust einzelner Pakete durchaus toleriert werden.

Geht dagegen über längere Zeit eine größere Menge von Paketen in einem Datenstrom verloren, zeigt das, daß entweder eine der beteiligten Maschinen oder die Netzwerkverbindung überlastet ist. Zur

Vermeidung des ersten Falls ist es wichtig, daß die Verarbeitung der Audiodaten im Audio-Subsystem möglichst effizient erfolgt. Im zweiten Fall ist es erforderlich, entweder die Netzwerkverbindung zu entlasten oder einen Audio-Datentyp zu verwenden, der eine geringere Bandbreite beansprucht.

Da die Audio-Daten im gleichen Zeittakt abgespielt werden, wie sie aufgenommen werden, und dementsprechend (von geringen Schwankungen abgesehen) getaktet (isochron) eintreffen, ist eine Flußkontrolle bei Live-Audiodaten nicht notwendig.

In dem seltenen Fall, daß Audio-Pakete schneller eintreffen, als sie abgespielt werden können, ist es zur Wahrung einer geringen Verzögerung sinnvoll, ältere Pakete zu verwerfen, um sofort neuere Pakete abspielen zu können.

3.3 Verwendung durch externe Audioquellen

3.3.1 Anwendungsschnittstelle

Als Zugang für externe Audioquellen soll das Audio-Subsystem eine wohldefinierte Anwendungsschnittstelle bereitstellen, die der Quelle die vom Audio-Subsystem angebotenen Dienste zur Verfügung stellt. In welcher Form diese Schnittstelle bereitgestellt wird, ist offen, da es bisher keine standardisierte Schnittstelle für derartige Anwendungen gibt.

Um gleichzeitig von MMC und einer externen Audioquelle benutzt zu werden, muß das Audio-Subsystem von mehreren Prozessen gleichzeitig gesteuert werden können; das gleiche gilt für eine Verwendung durch mehrere externe Audioquellen ohne MMC.

3.3.2 Echtzeitanforderungen

Für Live-Audiodaten gelten die gleichen Echtzeitanforderungen wie bei der Verwendung mit MMC.

3.3.3 Flußkontrolle

Beim Abspielen von gespeicherten Audiodaten ist die Situation genau umgekehrt: Hier kommt es darauf an, daß alle Daten genau so wiedergegeben werden, wie sie angeliefert werden; es darf nichts verlorengehen. Da es bei gespeicherten Audiodaten schwierig ist,

die Daten getaktet anzuliefern, ist es erforderlich, daß das Audio-Subsystem durch Flußkontrolle die Übertragung der Audiodaten steuert. Um Schwankungen bei der Anlieferung der Daten auszugleichen zu können, ist ein begrenztes Maß an Pufferung notwendig.

Da das AVXP keine Maßnahmen zur Flußkontrolle vorsieht, ist die Verwendung eines Transportprotokolls erforderlich, das diese Möglichkeit zur Verfügung stellt. Dafür kommen TCP und (zukünftig) MMT in Frage, UDP mangels eigener Möglichkeiten zur Flußkontrolle dagegen nicht.

3.3.4 AV-Übertragung ohne AVXP

Das AVXP ist auf die Übertragung von Live-Audiodaten im MMC-Kontext zugeschnitten. Für externe Audioquellen ist es nur begrenzt sinnvoll. Es ist daher wünschenswert, für externe Quellen eine Übertragung von Audiodaten ohne AVXP (Rohdaten) zu ermöglichen.

Zusammenfassung

Das zu entwickelnde System soll als Audio-Subsystem von MMC eingesetzt werden und die für diese Anwendung spezifizierten Anforderungen erfüllen. Insbesondere sind das:

- Kommunikation mit dem AVC-Controller über das ECP
- Einrichten und Löschen von Source und Sink Endpoints
- Übertragung von Audiodaten über das AVXP
- Wiedergabe (Mischen) von mehreren Audio-Datenströmen auch verschiedener Kodierung
- Aussenden von mehreren Audio-Datenströmen auch verschiedener Kodierung
- Verwendung der Transportprotokolle UDP und TCP; Erweiterbarkeit um andere Transportprotokolle
- Vermeidung von Aussetzern
- geringe Übertragungszeit

Für die Verwendung durch externe Audioquellen ergeben sich folgende Anforderungen:

- Schnittstelle für externe Quellen
- Steuerbarkeit durch mehrere Prozesse gleichzeitig
- Flußkontrolle für gespeicherte Audiodaten
- Datenübertragung ohne AVXP

In den folgenden Kapiteln beschreibe ich, wie diese Anforderungen in den Entwurf der AMEngine umgesetzt wurden.

KAPITEL 4 Vorgehensmodell

Zu Beginn eines Projekts ist es zunächst notwendig, sich für die anzuwendende Vorgehensweise zu entscheiden. Die Softwaretechnik bietet eine Reihe von Vorgehensmodellen für Software-Projekte, aus denen das für die Anforderungen des jeweiligen Projekts am besten geeignete auszuwählen ist.

4.1 Anforderungen des Projekts an das Vorgehensmodell

Wie allgemein in Entwicklungsprojekten und besonders in Forschungsprojekten üblich, zeichnete sich die Situation im MMC-Projekt durch häufige Änderungen der Anforderungen aus. Da bezüglich herstellerübergreifender Video-Konferenzsysteme zu Projektbeginn wenige Erfahrungen existierten, konnten die Anforderungen erst während der Projektlaufzeit aus den Erfahrungen mit den jeweils vorliegenden MMC-Implementierungen präzisiert werden. Auch der heutige Stand ist noch nicht abgeschlossen.

Um die Implementierungen der Projektpartner auf Interoperabilität zu überprüfen, waren im Projektplan regelmäßige Testtermine vorgesehen. In anschließenden Meilensteinabnahmen war die Fertigstellung der im Projektplan verlangten Ausbaustufe nachzuweisen.

Zusätzlich war es der Projektleitung bei der DeTeBerkom wichtig, die Fortschritte der Projektarbeit regelmäßig auf Messen (IFA, CeBIT u.a.) und anderen Veranstaltungen zu demonstrieren.

Während des Jahres 1995 fanden bereits Tests der MMC-Prototypen bei verschiedenen Anwendern statt, deren Erfahrungen zur Weiterentwicklung des Systems genutzt wurden.

4.2 Evolutionäres Prototyping

In dieser Projektsituation bot sich ein evolutionäres Vorgehen an. Beim evolutionären Prototyping handelt es sich um eine inkrementelle Systementwicklung. „Der Prototyp wird laufend den neu hinzugekommenden und aktualisierten Anforderungen angepaßt. Die Entwicklung ist somit kein abgeschlossenes Projekt, sondern verläuft parallel zur Nutzung der Anwendung. Es gibt keine Trennung zwischen Prototyp und Produkt. [...] Die rasche Validierung der Anforderungen durch die Anwender ist sofort am ausführbaren System möglich.“ [Wallmüller 1990]

Sage und Palmer schreiben über evolutionäres Prototyping:

> This model may work exceptionally well when the user is unsure of the system requirements and benefits from an iterative process in which successive models are refined an presented as prototypes if the proposed software product. [...] The use of prototyping becomes beneficial for situations in which the requirements may be stated vaguely or inadequately, [...] or perhaps where it is desirable to build critical portions of the system prior to making a commitment to construct a system in its entirety.
>
> Prototyping may be employed as a stand-alone process useful in viewing alternative solutions or checking algorithms, or it may be used to provide candidate solutions to particular problems or to produce a final product. [Sage, Palmer 1990]

Als eine Gefahr des evolutionären Prototypings nennt Wallmüller: „Die Entwurfsstruktur kann mangelhaft sein (»Flickwerk«). Das Problem des Systementwurfs ist durch diesen Ansatz nicht gelöst.“ [Wallmüller 1990]

Um diese Gefahr zu umgehen, wurde ein Redesign des Audio-Subsystems vorgenommen, sobald die Entwurfsstruktur für eine Weiterentwicklung nicht mehr tragbar war.

Dadurch ergaben sich drei Phasen der Arbeit am Audio-Subsystem:

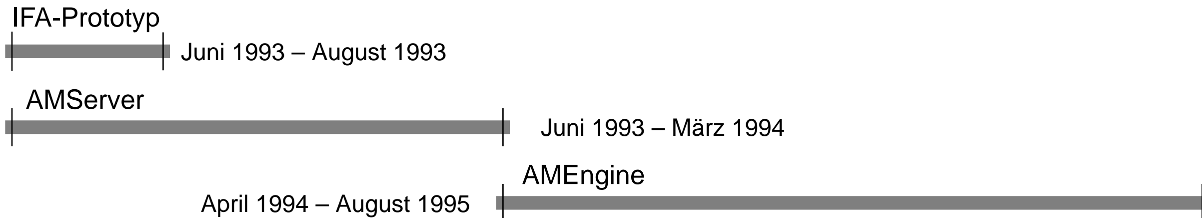

ABBILDUNG 6 Phasen der Arbeit am Audio-Subsystem

1. Nach dem Beginn der Projektarbeit im Juni 1993 wurde sehr schnell ein Prototyp für die Internationale Funkausstellung (IFA) benötigt, auf der MMC vorgestellt werden sollte. Die Anforderungen an diesen Prototypen waren gering. Diesen Prototyp entwickelte Ansgar Kückes, während ich gleichzeitig bereits damit begann, am zweiten Prototypen zu arbeiten.
2. In Konzept und Realisierung dieses zweiten Prototyps gingen die mit dem IFA-Prototypen gewonnenen Erfahrungen schon mit ein. Dieser zweite Prototyp, „AMServer“ genannt, erfüllte die Anforderungen für den Einsatz im MMC-Kontext schon zum großen Teil. Die Erfahrungen damit zeigten jedoch, daß schließlich doch die Wartbarkeit und die Flexibilität dieses Prototyps noch zu wünschen übrig ließen.
3. Um weitere Funktionalität in das Audio-Subsystem zu integrieren, war es unumgänglich, das System neu zu entwerfen. Die aus diesem Entwurf hervorgegangene Realisierung „AMEngine“ ist nun das Audio-Subsystem für die Implementierung der MMC-Konferenzapplikation der TU Berlin.

Die Arbeit an der AMEngine profitierte stark von der Arbeit am zweiten Prototypen, bei der etliche Design- und Implementierungsideen auf Verwirklichungs- und Einsatztauglichkeit überprüft werden konnten. Viele davon wurden in die endgültige Realisierung übernommen.

Im folgenden Kapitel stelle ich zunächst die Entwicklungsumgebung vor, in der die beschriebenen Systeme entstanden sind. Danach schildere ich die Eigenschaften der ersten beiden Prototypen und den darauf aufbauenden Entwurf der AMEngine.

KAPITEL 5 Plattformen und Werkzeuge

Die Hard- und Softwareplattform der Realisierung wurde durch den Projektkontext vorgegeben: Von der Firma Hewlett-Packard wurden der TU HP-Workstations mit dem Betriebssystem HP-UX zur Verfügung gestellt, die für das Projekt als Entwicklungs-, Test- und Vorführmaschinen benutzt werden.

5.1 Hardware-Plattform

Zum Einsatz im Projekt kamen Workstations des Typs HP 9000/712, HP 9000/725, HP 9000/735 und HP 9000/755. Sie basieren auf der Architektur HP PA-RISC 1.1 und sind mit Audio-AD/DA-Wandlern ausgestattet, die die üblichen Abtastraten von 8 bis 48 kHz und die Datenformate µlaw, Alaw und 16 Bit linear in Mono und Stereo zur Verfügung stellen.

5.2 Software-Plattform

5.2.1 Betriebs- und Windowsystem

Als Betriebssystem wird HP-UX 9.05 benutzt, eine Weiterentwicklung von UNIX System V Release 3 mit POSIX- und BSD-Erweiterungen. So kann zur Programmierung der Netzwerkschnittstelle die von 4.2BSD bekannte Socket-Abstraktion benutzt werden. [HP 1992, Leffler et al. 1989, Stevens 1990]

Als Window-System stand das Window-System X in der Version 11 zur Verfügung. Es wurden sowohl das von HP mitgelieferte Release

5 als auch das Release 6 in der Distribution vom MIT benutzt. Da das Audio-Subsystem keine graphische Benutzungsschnittstelle hat, hatte das Window-System hier lediglich als Entwicklungsplattform eine Bedeutung.

5.2.2 Schnittstelle zur Audio-Hardware

Audio Application Programming Interface

HP bietet mit HP-UX 9 zur Programmierung der Audio-Hardware ein *Audio Application Program Interface* (AAPI) an. Diese Schnittstelle benutzt den von HP mitgelieferten Audioserver, der als Hintergrundprozeß den Zugriff zur Betriebssystemschnittstelle der Audio-Hardware verwaltet.

Bei Tests des AAPI zeigte sich, daß der Audioserver eine sehr hohe Verzögerung der Audiowiedergabe bewirkt. Diese Verzögerung liegt bei etwa einer Sekunde und macht das AAPI für eine Videokonferenz-Applikation unbrauchbar. Experimente brachten zutage, daß die Audio-Wiedergabe hin und wieder für ein paar Sekunden stehenblieb. Auch dieses Verhalten macht den Audioserver und damit das AAPI für eine Verwendung mit MMC ungeeignet.

Treiber des Betriebssystems

Der Audioserver setzt auf dem Audio-Treiber des Betriebssystems auf, der Unix-typisch einen Zugriff über sogenannte Device-Files ermöglicht. Diese Schnittstelle kann auch von anderen Programmen genutzt werden. Dieser Zugriff ist von HP für andere Applikationen als den eigenen Audioserver allerdings nicht vorgesehen und deshalb vollkommen undokumentiert.

Durch ein Hilfeersuchen via Usenet kam ich in Kontakt mit Frode Kileng, der als Programmierer für das Norwegian Telecom Research Department arbeitet. Für eine ähnliche Applikation benutzt er den direkten Zugriff auf die Device Files. Er gab mir die nötige Starthilfe, um anhand von eigenen Versuchen und der Informationen im System-Headerfile `<sys/audio.h>` die Funktionsweise der Audio-Schnittstelle zu ergründen. [Kileng 1993]

Mit den so gewonnenen Informationen konnte ich mir eine eigene einfache Programmierschnittstelle für die Audio-Hardware der HP-Workstations schreiben. Diese Schnittstelle ist im Kapitel „Entwurf“ näher beschrieben. Sie wurde bisher getestet auf HP-Workstations

der Typen HP 9000/712, HP 9000/715, HP 9000/725, HP 9000/735 und HP 9000/755.

Ein Nachteil des HP-Audio-Treibers lag zunächst in seinem Zeitverhalten: Intern wird ein DMA-Puffer mit einer Größe von 4 KB verwendet. Ein DMA-Transfer fand erst statt, wenn dieser Puffer gefüllt war [Kileng 1993]. Bei der in MMC vereinbarten Standard-Audiokodierung G.711 (8 bit ulaw, 8 kHz mono) bewirkte allein das schon eine Verzögerung des Audiosignals von 512 ms; mit der später von mir gewählten internen Kodierung (16 bit linear, 16 kHz stereo) verringerte sich diese Hardware-bedingte Verzögerung auf 64 ms.

Inzwischen gibt es einen Patch für HP-UX, der diese Verzögerung im Audio-Treiber deutlich verringert. Bei eigenen Messungen konnte ich feststellen, daß mit diesem Patch die komplette Systemdurchlaufzeit des Tonsignals (vom Mikrophon der einen Maschine zum Lautsprecher der anderen) bei der benutzten Abtastrate und Kodierung von über 200 ms auf etwa 100 ms verringert wird.

Funktionalität des HP-Audio-Device-Treibers

Der Audio-Treiber von HP-UX 9 stellt als Schnittstelle zwei Device Files zur Verfügung, `/dev/audio` und `/dev/audioCtl`. Das Device `/dev/audio` dient als Quelle und Senke für Audiodaten. Dementsprechend kann es zum Lesen oder zum Schreiben oder für beides geöffnet werden. Ein `read`(2) liest einen vom AD-Wandler gelieferten Datenstrom, der das über einen analogen Eingang aufgenommene Tonsignal repräsentiert; `write`(2) schreibt Daten zum D/A-Wandler, die dann auf einem der analogen Ausgänge als Tonsignal ausgegeben werden. Per `ioctl`(2) können die Parameter der Datenkonvertierung von `/dev/audio` gesteuert werden, beispielsweise das Setzen der Abtastrate. `/dev/audio` kann immer nur von einem Prozeß gleichzeitig geöffnet werden; versucht ein weiterer Prozeß ein `open`(2), erhält er den Fehlercode `EBUSY` („Device busy“, siehe dazu `errno`(2)). Damit kann kein konkurrierender Prozeß gleichzeitig (störende) Daten auf `/dev/audio` schreiben oder (dem ersten Prozeß dann fehlende) Daten von `/dev/audio` lesen oder durch Ändern der Parameter die Wiedergabe oder Aufnahme des ersten Prozesses stören.

Das Device `/dev/audioCtl` kann von mehreren Prozessen gleichzeitig zum Lesen und Schreiben geöffnet werden. Lesen von `/dev/audioCtl` liefert eine End-Of-File-Bedingung, Schreiben den Fehler `EACCES` („Permission denied“). (Interessanterweise ist diese Fehlerbedingung in der Manual Page zu `write`(2) nicht unter den möglichen aufgeführt.) Über `ioctl`(2) können die Parameter von `/dev/audio` ausgelesen und einige davon auch gesetzt werden, beispiels-

weise die Aufnahme- und Wiedergabelautstärke. Gelesen werden können dabei auch die Werte, die nur per ioctl(2) auf /dev/audio gesetzt werden können. Damit ist es zum Beispiel allen Prozessen, die /dev/audioCtl öffnen dürfen, möglich, die Abtastrate auszulesen, aber nicht, die Abtastrate zu ändern.

Eine Tabelle zeigt die möglichen ioctl(2)-Aufrufe nach <sys/audio.h>, ob sie von AIO und der AMEngine benutzt werden, ob sie mit /dev/audioCtl ausgeführt werden können und kurz ihre Bedeutung. Ein Fragezeichen steht dort, wo Informationen nicht bekannt sind. Manches davon ließe sich sicher noch durch gezieltes Ausprobieren herausfinden, aber das war für diese Anwendung bisher nicht notwendig.

TABELLE 2 ioctl(2)-Aufrufe des Audio-Device-Treibers

ioctl Call	benutzt von AIO	benutzt von AME	Ctl	Bedeutung
AUDIO_DESCRIBE	•		•	Beschreibung der Geräteeigenschaften
AUDIO_SET_LIMITS			?	?
AUDIO_GET_LIMITS			?	?
AUDIO_SET_RXBUFSIZE	•	•		Größe des Lesepuffers setzen
AUDIO_GET_RXBUFSIZE			?	Größe des Lesepuffers auslesen
AUDIO_SET_TXBUFSIZE	•	•		Größe des Schreibpuffers setzen
AUDIO_GET_TXBUFSIZE			?	Größe des Schreibpuffers auslesen
AUDIO_SET_OUTPUT	•	•	•	Ausgabekanal setzen
AUDIO_GET_OUTPUT	•		•	Ausgabekanal auslesen
AUDIO_SET_GAINS	•	•	•	Lautstärken setzen
AUDIO_GET_GAINS	•		•	Lautstärken auslesen
AUDIO_SET_DATA_FORMAT	•	•		benutztes Datenformat setzen
AUDIO_GET_DATA_FORMAT	•			benutztes Datenformat setzen
AUDIO_SET_SEL_THRESHOLD	•	•		Select-Threshold setzen
AUDIO_GET_SEL_THRESHOLD	•		•	Select-Threshold auslesen
AUDIO_GET_STATUS			?	?
AUDIO_RESET	•	•		Audio Device zurücksetzen
AUDIO_PAUSE			?	?
AUDIO_RESUME			?	?
AUDIO_DRAIN			?	?
AUDIO_RAW_SET_PARAMS	0	0	?	?

TABELLE 2 `ioctl`(2)-Aufrufe des Audio-Device-Treibers

ioctl Call	benutzt von AIO	benutzt von AME	Ctl	Bedeutung
AUDIO_RAW_GET_PARAMS			?	?
AUDIO_METER			?	?
AUDIO_METER_CTL			?	?
AUDIO_SET_SAMPLE_RATE	•	•	0	Abtastrate setzen
AUDIO_GET_SAMPLE_RATE	•		•	Abtastrate auslesen
AUDIO_SET_CHANNELS	•	•		Benutzte Kanäle (Mono/Stereo) setzen
AUDIO_GET_CHANNELS	•		•	Benutzte Kanäle (Mono/Stereo) auslesen
AUDIO_SET_INPUT	•	•	•	Eingabekanal setzen
AUDIO_GET_INPUT	•		•	Eingabekanal auslesen
AUDIO_SET_BEEP			?	?
AUDIO_GET_BEEP			?	?
AUDIO_SET_BEEP_OUTPUT			?	?
AUDIO_GET_BEEP_OUTPUT			?	?
AUDIO_TEST_MODE			?	?
AUDIO_TEST_RW_REGISTER			?	?
AUDIO_TEST_RW_BUFFER			?	?
AUDIO_TEST_INT			?	?

5.2.3 Tcl-Interpreter

Um das Module für die Kommandoschnittstelle einfacher und flexibler gestalten zu können, benutzte ich einen Tcl-Interpreter (siehe Kapitel „Entwurf der AMEngine“).

Die Skriptsprache Tcl („Tool Command Language“) und der dazugehörigeInterpreter wurden im Sprite-Projekt an der University of California at Berkeley unter der Leitung von John Ousterhout entwickelt . Die Sprache Tcl ist für den Einsatz als Kommando- und Erweiterungssprache in größeren Applikationen vorgesehen; der Interpreter ist so gestaltet, daß er leicht inApplikationen mit eingebunden werden kann. Er stellt eine ausgesprochen gut durchdachte prozedurale Schnittstelle (API) zur Verfügung, mit der die Funktionalität der Applikation dem Interpreter als eine Reihe von Tcl-Kommandos bekannt gemacht wird. Weitere Funktionen ermöglichen die Manipulation der Rückgabewerte dieser Tcl-Kommandos von der Applikation aus.

Als Grundfunktionalität enthält der Tcl-Interpreter bereits Kontrollstrukturen, String- und Listenbearbeitung und Dateioperationen.

[Ousterhout 1990, 1994]

5.3 Benutzte Werkzeuge

5.3.1 Programmiersprache

Eine besonders wichtige Entscheidung in Softwareprojekten ist die Wahl der Programmiersprache. Folgende Kriterien beeinflußten diese Wahl:

1. Verfügbarkeit auf der verwendeten Hardware
2. Verwendbarkeit für die Aufgabe
3. hinreichende Effizienz
4. Know-How/Erfahrung mit der Sprache

Für HP-Workstations sind Implementierungen sehr vieler Programmiersprachen verfügbar. Da die Implementierung des Audio-Subsystems die Benutzung einer sehr systemnahen Schnittstelle erforderte, verringerte sich die Auswahl auf die Sprachen C und C++. Beide Sprachen sind für systemnahe Programmierung entwikkelt, der von C-Compilern erzeugte Code ist verglichen mit anderen Sprachen sehr effizient.

Wegen des Zeitdrucks bei der Entwicklung schied die Sprache C++ aus, da für die Verwendung von C++ eine längere Einarbeitungszeit notwendig gewesen wäre.

Mittlerweile hat sich der ANSI-Standard für die Sprache C (American National Standard for Information Systems -- Programming Language C, X23.159-1989 [Kernighan, Ritchie 1990]) weitgehend durchgesetzt. Im Gegensatz zum „traditionellen“ C kann der Compiler bei der Verwendung von ANSI C die korrekte Verwendung von Prozedurschnittstellen sehr viel besser prüfen. Da es einen festgeschriebenen Standard gibt, sind Programme in ANSI C portabler als bei der Verwendung des alten quasi-Standards.

5.3.2 Compiler

Für HP-Workstations unter HP-UX standen uns zwei C-Compiler zur Verfügung, die dem ANSI-Standard entsprechen:

- HP liefert einen Compiler, der effizienten Code erzeugt, sehr schnell kompiliert und den Sprachstandard rigoros einhält. Leider ist er mit Warnungen, die zwar standardkonforme, aber stilistisch unsaubere Stellen im Programm betreffen, sehr zurückhaltend.
- Aus dem GNU-Projekt der Free Software Foundation kommt der GNU C-Compiler (GCC). Dieser Compiler erzeugt zwar etwas weniger effizienten Code und kompiliert langsamer als der HP-Compiler, ist dafür aber sehr viel gründlicher in der statischen Programmanalyse und mahnt sehr viele potentielle Fehlerursachen in Programmen an.

Für die Programmentwicklung benutzte ich deshalb wechselseitig beide Compiler -- den GCC zur Programmüberprüfung und den HP-Compiler zur Überprüfung der ANSI-Konformität und für die ausgelieferten Versionen des Programms.

5.3.3 Debugger

Als Debugger benutzte ich GDB aus dem GNU-Projekt. GDB ist, vor allem in Kombination mit dem Editor Emacs (siehe unten), ein sehr leistungsfähiger und flexibler Sourcecode-Debugger.

5.3.4 Versions- und Konfigurationsverwaltung

Für die Versions- und Konfigurationskontrolle benutzte ich das Shape Toolkit (kurz ShapeTools). ShapeTools ist ein Toolkit für Software Configuration Management und wurde an der TU Berlin in den Projekten UNIBASE und STONE entwickelt. [Lampen, Mahler, Nickelsen 1993]

Über die Funktionalität der ebenfalls zur Verfügung stehenden Versionsverwaltungstools RCS und CVS hinaus bietet ShapeTools ein Release-Management-System, das eine kontrollierte Erstellung von Software-Releases erleichtert und es ermöglicht, jederzeit auf alte Releases der Software zurückzugreifen.

ShapeTools erleichtert gegenüber make(1) die Erstellung von Produktvarianten sowie durch eine enge Integration der Versionsverwaltung mit der Systemkonfigurierungskomponente eine dynamische Auswahl von Versionen bei der Systemkonfigurierung.

5.3.5 Texteditor

Die Auswahl eines Texteditors gilt unter Programmierern im wesentlichen als Geschmacksfrage. Natürlich ist auch meine Wahl durch meine persönlichen Vorlieben und meine vorhergehenden

Erfahrungen bestimmt, aber ich denke, daß auch eine Reihe objektiver Gründe dafür sprechen.

XEmacs ist ein Vertreter der Emacs-Familie. Die Firma Lucid, Inc. entwickelte den GNU Emacs weiter, um ihn als strategische Software-Plattform für ihre Programmierumgebung zu nutzen. Nach dem Ende der Firma Lucid wird XEmacs von der Universiy of Illinois at Urbana-Champaign weiterentwickelt.

GNU Emacs und damit auch XEmacs zeichnen sich durch eine außerordentliche Flexibilität und eine besondere Unterstützung der Programmentwicklung aus. Ein für mich wesentlicher Punkt ist die Unterstützung beim Erstellen und Bearbeiten von Programmtexten in der Sprache C.

- Der Editor nimmt die Einrückung des Quelltextes automatisch syntaxgesteuert vor. Das erspart einerseits Arbeit und macht andererseits dadurch schon vor dem Compilerlauf auf Fehler bei Verschachtelungen wie zum Beispiel vergessene Klammern aufmerksam.
- Mit einer Tastenkombination können Definitionen von Konstanten, Typen, Variablen und Prozeduren im jeweiligen Modul aufgesucht werden. Das verringert Suchzeiten in den Programmquellen.
- Durch die Möglichkeit, mehrere Module gleichzeitig in gleichzeitig sichtbaren Fenstern zu bearbeiten, werden Änderungen unterstützt, die mehrere Module betreffen.

Erhebliche Vorteile bietet Emacs durch die Integration mit anderen Programmierwerkzeugen:

- Emacs unterstützt das Arbeiten mit GDB. Dabei wird die Ausgabe von GDB in einen Emacs-Buffer umgeleitet; Kommandos an den GDB werden hier eingegeben. Beim schrittweisen Ablaufen des Programms oder bei einem Laufzeitfehler wird in einem anderen Fenster die aktuelle Stelle im Quelltext angezeigt.
- Zum Übersetzen und Linken des Programms wird der Compiler (bzw. make oder das Programm shape aus dem Shape Toolkit) vom Emacs aus gestartet. Beim Auftreten von Fehlermeldungen kann mit einer Tastenkombination oder einem Mausklick direkt zur fehlerhaften Stelle des Programms gesprungen werden.

XEmacs ist gut in das X Window System integriert. Zum Beispiel konnte ich mit ein paar Zeilen in Emacs-Lisp, der Makro- und Erweiterungssprache von Emacs, die GDB-Schnittstelle so erweitern, daß

ich Buttons für die am häufigsten benutzten Kommandos (Step, Next, etc.) zur Verfügung hatte.

5.4 Zusammenfassung

TABELLE 3 Benutzte Plattformen und Werkzeuge

Hardware	HP 9000/712, 725, 735, 755
Betriebssystem	HP-UX 9.05
Window-System	X11, Releases 5 und 6
Audio Hardware	HP Audio II (serienmäßig eingebaut)
Audio-Schnittstelle	`/dev/audio`, `/dev/audioCtl`
Programmiersprache	C
Compiler	HP ANSI C, GCC
Debugger	GDB
Versionverwaltung	ShapeTools
Texteditor	XEmacs
Sonstiges	Tcl-Interpreter

KAPITEL 6 Frühe Prototypen

In diesem Kapitel schildere ich die Erfahrungen mit den ersten beiden Prototypen des Audio-Subsystems und den darauf aufbauenden Entwurf der AMEngine.

6.1 IFA-Prototyp

Da das MMC-Projekt an der TU Berlin wegen organisatorischer Verzögerungen verspätet begonnen hatte, mußte für die Präsentation auf der Internationalen Funkausstellung (IFA) im August 1993 eine schnelle Lösung für eine provisorische Implementierung der AVC gefunden werden. Während ich noch an konzeptionellen Überlegungen für das Audio-Subsystem arbeitete, implementierte Ansgar Kückes für die IFA einen bereits funktionsfähigen Prototyp mit eingeschränkter Funktionalität.

6.1.1 Architektur / Implementierung

Die AVC war zu diesem Zeitpunkt noch in einem einzigen Programm realisiert, das die Funktionalitäten von Audio, Video und der Schnittstelle zum AVM realisierte.

Die Audio-Qualität war auf einen Datentyp beschränkt (G.711); der Audio-Teil der AVC konnte dadurch sehr einfach gehalten werden. Das Endpoint-Modell von MMC wurde hier nur unvollständig verwirklicht. Beim Start des Programms wurde ein einziger Sink Endpoint erzeugt, der während der gesamten Laufzeit der AVC existierte.

Die beim Sink Endpoint eintreffenden Daten wurden in Listen von Datenblöcken geschrieben, die nach dem FIFO-Prinzip (First In, First Out) abgearbeitet wurden. Über einen Intervall-Timer wurde die Hauptschleife des Sink-Prozesses regelmäßig unterbrochen, um die Daten aus den FIFO-Listen auf das Audio-Device auszugeben.

Mehrere eingehende Datenströme wurden der Einfachheit halber nicht gemischt, sondern über je einen Stereo-Kanal wiedergegeben. Dadurch ergab sich eine subjektiv nicht unangenehme Trennung der Tonwahrnehmung der beiden Gesprächspartner; diese Idee wurde später wieder aufgegriffen (siehe Kapitel „Erfahrungen und Ausblick“). Die Anzahl der Konferenzteilnehmer war damit auf maximal drei begrenzt.

6.1.2 Nützlichkeit und Einschränkungen

Der IFA-Prototyp erfüllte seine Aufgabe gut. Er zeigte ein funktionsfähiges Audio-Subsystem für die MMC-Implementierung der TU Berlin. Die Präsentation auf der Funkausstellung verlief damit ohne wesentliche Probleme.

Die Einschränkung auf drei Gesprächspartner war einer der entscheidenden Nachteile dieses Prototyps. Sehr störend war die durch die häufig sehr lang werdenden FIFO-Listen verursachte hohe Verzögerung, die teilweise Werte von ein bis zwei Sekunden erreichte. Es blieb unklar, ob die über einen Zeittakt gesteuerte Ausgabe auf das Audio-Device nicht schließlich zu Synchronisierungsproblemen und damit zu Wiedergabefehlern führen würde, da die Timer-Funktionen von Unix relativ ungenau sind. Zwar konform zur derzeitigen MMC-Spezifikation, aber trotzdem auf Dauer unbefriedigend war die Beschränkung auf einen einzigen Audio-Datentyp.

Da die Realisierung des IFA-Prototyps eher ad hoc entstanden war, bot sich ein weiterer Ausbau nicht an. Die Erfahrungen damit waren aber für die weitere Entwicklung nützlich.

6.2 Der zweite Prototyp: AMServer

Schon während der Entwicklung und Erprobung des IFA-Prototyps arbeitete ich an einer Entwicklung für das Audio-Subsystem, die eine deutlich flexiblere Verarbeitung der Audio-Datenströme bieten sollte.

6.2.1 Anforderungen

Nach den Erfahrungen mit dem IFA-Prototyp erachteten wir für die weitere Entwicklung folgende Punkte als besonders wichtig:

- Verringerung des Delays bei der Audio-Übertragung,
- Eine höhere mögliche Anzahl von Audio-Verbindungen,
- Nutzung verschiedener Audio-Datentypen,
- Höhere Stabilität.

6.2.2 Redesign der AVC

Für diese Anforderungen war der IFA-Prototyp nicht mehr die geeignete technische Basis. Wir machten deshalb einen kompletten Neuentwurf der gesamten AVC mit dem Ziel, eine hinreichend flexible Grundlage für die geplanten Erweiterungen zu schaffen. Diesen Entwurf habe ich in der Einleitung schon kurz vorgestellt; ich gehe hier näher auf den Audio-Teil ein.

Schnittstellen des Audio-Subsystems

Das Audio-Subsystem hat drei verschiedene Arten von Schnittstellen zu anderen Systemen:

1. Prozedurale Schnittstelle zum Betriebssystem: Systemaufrufe (`open`(2), `close`(2), `read`(2), `write`(2), `etc.`).
2. Kommandoschnittstelle zum AVC-Controller: Auf Unix-Pipes [Stevens 1992] basierende Interprozeßkommunikation; Nachrichten als textuelle Kommandos (Endpoint Control Protocol).
3. Netzwerkschnittstelle zu AVCs auf anderen Rechnern bzw. zu Anwendungen: Im AVXP definiertes Format für UDP-Pakete oder Übertragung von reinen Audio-Daten.

6.2.3 Hardware-Schnittstelle: AIO

Die von HP vorgesehene Schnittstelle zur Audio-Hardware der Workstation ist, wie im Kapitel „Plattformen und Werkzeuge" geschildert, nicht geeignet. Um die Betriebssystemaufrufe in gut handhabbare Aufrufe zu kapseln, entwarf ich mir eine eigene Programmierschnittstelle, die ich AIO (für „Audio I/O") nannte. Für diese Schnittstelle gab es keine äußeren Vorgaben, und da es keinen plattformübergreifenden Standard für diese Funktionalität gibt, ist sie auch nicht an ein bekanntes Vorbild angelehnt.

Nach dem Motto „make simple things simple; make complicated things possible" entwickelte ich eine Schnittstelle, die für einfache

Anwendungen sehr einfach zu benutzen ist, aber trotzdem alle Möglichkeiten der Audio-Hardware zur Verfügung stellt, die für das Audio-Subsystem benötigt werden.

Außer dem reinen Zugriff auf die Audio-Hardware enthält diese Schnittstelle auch andere Basisfunktionen, die beim Umgang mit Audiodaten oft benötigt werden, wie Mischen und Konvertierung von Audiodaten. Um die Erweiterbarkeit des Audio-Subsystems zu verbessern, bekamen die Konvertierungsfunktionen eine einheitliche Schnittstelle. Damit können die Konvertierungsfunktionen mit ihrem Ein- und Ausgabeverhalten (Größenverhältnis zwischen Ein- und Ausgabedaten) in Variablen gehalten werden, die einem Datenstrom zugeordnet sind, und über diese Variablen aufgerufen werden. Eine unflexible explizite Fallunterscheidung je nach Typ der Konvertierung entfällt damit.

Die AIO-Schnittstelle wurde als eine Library implementiert, die auch von anderen Programmen benutzt werden kann. Sie ist im Anhang ausführlicher beschrieben.

6.2.4 Architektur

Vier verschiedene Arten von äußeren Ereignissen betreffen das Audio-Subsystem:

- Ein Paket mit Audiodaten kommt über das Netzwerk an.
- Vom Audio-Device aufgenommene Audio-Daten stehen zum Lesen bereit.
- Auf das Audio-Device können Audio-Daten zum Abspielen geschrieben werden.
- Über die Steuerverbindung kommt ein ECP-Request vom AVC-Controller.

All diese Ereignisse treten asynchron auf. Die genannten Echtzeitanforderungen für Live-Kommunikation erfordern eine (möglichst) sofortige und adäquate Behandlung jedes Ereignisses.

Diese Art von Anforderung legt eine Architektur nahe, in der in einer Schleife auf ein von außen eintreffendes Ereignis gewartet wird und bei seinem Eintreffen die entsprechende Aktion angestoßen wird, ähnlich wie in der XtMainLoop(2) von X11. [Nye, O'Reilly 1992]

Der Kern einer solchen Schleife ist der Unix-Systemaufruf `select`(2), der auf ein Ereignis wartet, das auf einem von mehreren Ein-

Ausgabekanälen eintreffen kann. Eine solche Schleife wird deshalb üblicherweise Select-Loop genannt. Sie bildet den Kern des Programms. Von hier aus werden Prozeduren aus den anderen Programmteilen aufgerufen, um das eingetretene Ereignis zu bearbeiten.

Steuerung über ECP

Das Audio-Subsystem wird als eigenständiger Prozeß vom AVC-Controller gestartet und kommuniziert mit diesem über Unix-Pipelines. Um den AMServer auch ohne MMC testen zu können, sah ich die Möglichkeit vor, die ECP-Kommandos von der Standardeingabe zu lesen und die Rückgabewerte und Fehlermeldungen auf die Standardausgabe zu schreiben. So ließ sich der AMServer als interaktives Programm mit einer kommandozeilenorientierten Benutzungsschnittstelle betreiben.

Endpoints

Die gleichzeitige Wiedergabe von mehreren eingehenden Datenströmen erfordert ein Mischen der Audiodaten, ähnlich dem Mischen der Audio-Signale in einem analogen Mischer.

Im Gegensatz zur im Projekt zunächst verwendeten µlaw-Kodierung des G.711-Formats können linear kodierte Audiodaten durch einfache arithmetische Mittelung gemischt werden. Da die lineare Kodierung mit 16 Bit von den zur Verfügung stehenden den größten Wertebereich hat, bietet es sich an, diese Kodierung allgemein als internes Datenformat zu verwenden. Dazu werden eingehende Datenströme anderer Kodierung zu linearer Kodierung konvertiert; ausgehende Datenströme werden von der linearen Kodierung in die entsprechende externe Kodierung konvertiert.

Da das Eintreffen von Audiodaten über das Netzwerk und die Wiedergabe der Daten auf das Audio-Device asynchron geschieht, werden die Datenblöcke in Listen zwischengespeichert. Das gleiche gilt für das Lesen von Audiodaten vom Audio-Device und das Verschikken der Daten über das Netzwerk. Um die korrekte Reihenfolge der Daten zu wahren, werden die zuerst in die Liste aufgenommenen Blöcke als erste wieder entnommen (First In, First Out). Ich nenne diese Listen im folgenden FIFO-Listen.

Sink Endpoints

Bei eingehenden Datenströmen werden die Daten

1. vom Netzwerk gelesen,

2. von der jeweiligen externen in die interne lineare Kodierung konvertiert,
3. in die FIFO-Liste des jeweiligen Sink Endpoints übertragen,
4. mit den eingehenden Datenströmen anderer Sink Endpoints gemischt und
5. auf dem Audio-Device ausgegeben.

Die Punkte 2 und 3 werden in einem erledigt, da mit der Datenkonvertierung grundsätzlich auch ein Kopiervorgang verbunden ist. In Punkt 4 werden mehrere eingehende Datenströme zu einem vereinigt.

Wenn, wie im AVXP vorgesehen, die Menge der Nutzdaten in einem Audio-Paket nicht festgelegt ist, kann nicht davon ausgegangen werden, daß diese Nutzdaten (nach der Umkodierung) eine ganzzahlige Anzahl von Blöcken in den FIFO-Listen füllen. Deshalb entwickelte ich die Listenverwaltung so, daß sie auch teilweise gefüllte Blöcke verwalten kann.

Source Endpoints

Die Daten eines ausgehenden Datenstroms werden

1. von der Audio-Hardware gelesen,
2. vom internen in das externe Datenformat konvertiert,
3. in die Warteschlange für den korrespondierenden Sink Endpoint der AVC auf dem anderen Rechner übertragen und
4. über das Netzwerk an den korrespondierenden Sink Endpoint geschickt.

Bei Punkt 2 findet eine Verzweigung von Datenströmen statt, wenn mehr als ein externes Datenformat verwendet wird. Bei Punkt 3 findet eine Verzweigung statt, wenn an mehr als einen Sink Endpoint das gleiche Datenformat geschickt wird.

Endpoints mit verschiedenen Audio-Qualitäten

Für den Austausch von Audiodaten wurde generell G.711 benutzt. Da die Tonqualität von G.711 nur die Qualität eines ISDN-Telefons

erreicht, integrierte ich zusätzlich noch zwei Audio-Datentypen mit einer Abtastrate von 16 kHz:

TABELLE 4 Audio-Datenformate beim AMServer

Abtastrate	Kodierung	Kanäle
8000	μlaw	1
16000	μlaw	1
16000	linear	2

Da das letzte Datenformat durch die höhere Datenrate eine kürzere Systemdurchlaufzeit in der Audio-Hardware bewirkt, benutzte ich es als internes Format.

Um (bei Sink Endpoints) 8-kHz-Daten in 16-kHz-Daten umzusetzen, implementierte ich eine Konvertierungsprozedur, die nach jedem Sample ein weiteres einfügt. Der eingefügte Wert wird zwischen den beiden benachbarten linear interpoliert. Bei der umgekehrten Konvertierung wird jedes zweite Sample weggelassen. So können auch Endpoints mit verschiedenen Abtastraten gleichzeitig benutzt werden.

Die resultierende Tonqualität ist für Sprache brauchbar, wenn auch nicht optimal. Korrekte Algorithmen zur Abtastratenkonvertierung sind sehr aufwendig und als Softwarelösung nicht effizient genug, kommen also für den Einsatz im Audio-Susbsystem nicht in Frage.

Debugging

Für Debugging-Ausgaben während des Programmlaufs implementierte ich eine Prozedur, die abhängig von einem angegebenen „Debuglevel" ähnlich wie printf(3) Meldungen ausgeben kann. Dieser Debuglevel kann über eine Kommandozeilen-Option oder über die Steuerverbindung gesetzt werden. Meldungen über selten auftretende Ereignisse werden schon bei einem niedrigen Debuglevel ausgegeben, Meldungen über sehr häufig auftretende Ereignisse dagegen erst bei einem hohen Debuglevel.

6.2.5 Erfahrungen mit dem AMServer

Die Erfahrungen mit der AMServer-Implementierung führten zur weiteren Präzisierung der Anforderungen für das Audio-Subsystem.

AIO

Die Kapselung des Zugriffs auf die Audio-Hardware in der AIO-Library bewährte sich sehr gut. Die Schnittstelle erwies sich als brauchbar, die nur auf relativ vagen Erkenntnissen über die benutzte Betriebssystemschnittstelle aufbauende Implementierung als zuverlässig. Lediglich die Schnittstelle zu den Datenkonvertierungsfunktionen konnte später noch verbessert werden, nachdem die Anforderungen in diesem Punkt besser bekannt waren.

Übertragungszeit (Delay)

Das Delay-Verhalten des AMServers erwies sich als sehr gut. Während der IFA-Prototyp etwa ein bis zwei Sekunden Verzögerung zwischen Aufnahme und Wiedergabe erreichte, erzielte der AMServer Werte um 250 bis 300 Millisekunden. (Diese Werte gelten für HP-UX ohne den im vorigen Kapitel erwähnten Patch.) Nicht ganz geklärt werden konnte eine Erhöhung des Delays, sobald ein dritter Konferenzteilnehmer hinzukam.

Mehrere Audio-Verbindungen

Die Verwaltung mehrerer Audio-Verbindungen zu den Konferenzteilnehmern verlief problemlos. Die größte Auslastung fand am Ende der Hannover-Messe CeBIT 1994 bei einer Videokonferenz mit sieben Teilnehmern statt. Hier nahm die Verarbeitung der Videobilder einen großen Teil der Rechenleistung in Anspruch, so daß nicht immer genug für die Verarbeitung der Audio-Daten zur Verfügung stand. Die Funktionen für den Verbindungsauf- und -abbau waren dadurch nicht beeinträchtigt.

ECP / Kommandozeile

Es erwies sich als außerordentlich nützlich, das Programm über die Standardein- und -ausgabe via ECP steuern zu können. Die Möglichkeit, das Audio-Subsystem zu testen, ohne dafür eine MMC-Konferenz starten zu müssen, sparte bei der Entwicklung und beim Testen und Debuggen viel Zeit. Ansgar Kückes, der diese Möglichkeit beim Video-Subsystem ebenfalls nutzte, berichtete von ähnlich positiven Erfahrungen.

Die Benutzbarkeit des AMServers als Standalone-Programm legte die Idee nahe, das MMC-Audio-Subsystem ähnlich dem HP-Audioserver auch für andere Applikationen zugänglich zu machen. Da der AMServer eingehende Audio-Datenströme mischen konnte, war

auch die Benutzung durch mehrere Applikationen gleichzeitig denkbar. Dazu fehlte beim AMServer noch die Möglichkeit, mehrere Steuerverbindungen gleichzeitig zu bedienen.

Stabilität

Wie sich im Betrieb des AMServers zeigte, ließ die Stabilität des Programms etwas zu wünschen übrig. Typisch für den AMServer waren weniger spontane Abstürze (die nur selten vorkamen) als vielmehr die Neigung, während einer laufenden Konferenz auf einmal sämtliche CPU-Zeit an sich zu reißen. Obwohl durch die Scheduling-Eigenschaften von Unix ein Fortführen der Konferenz möglich war, ist ein solches Verhalten nicht tolerierbar.

Pakete beliebiger Größe

Die Prozedur, die die Audio-Daten von der externen in die interne Kodierung konvertiert und dabei gleichzeitig in die internen Warteschlangen kopiert, sollte eigentlich in der Lage sein, externe Pakete beliebiger Größe aufzunehmen. Leider gelang es mir aus ungeklärt gebliebenen Gründen nicht, dieses Verhalten auch tatsächlich zu erreichen. Sobald die Größe der ankommenden AVXP-Pakete in einem nicht ganzzahligen Verhältnis zur internen Blockgröße stand, gab es bei der Audio-Wiedergabe schnelle Folgen von Aussetzern, die einen Fehler in diesem Bereich anzeigten.

Da in dieser Phase des Projekts für die Meilensteinabnahme noch eine feste Paketgröße benutzt wurde, ließ sich der AMServer dafür noch verwenden.

Debugging

Das Auswählen von Debugging-Meldungen nur über einen Debuglevel erwies sich als unbefriedigend. Typischerweise ist man nicht an allen Meldungen gleich stark oder wenig interessiert, sondern eher an Meldungen zu bestimmten Themen.

Als ungünstig zeigte sich das Fehlen einer Möglichkeit, während des Betriebs des AMServers Informationen über den AMServer und die Endpoints und Verbindungen zu erhalten (Monitoring).

Wartbarkeit / Erweiterbarkeit

Bei der Fehlersuche und bei Überlegungen zur Erweiterung des Programms zeigte sich der AMServer als zu unübersichtlich. In der evolutionär gewachsenen Struktur des Programms waren die einzelnen Module zu eng verflochten, die Funktionalität war nicht klar geord-

net. Damit war der AMServer für eine weitere Arbeit nicht mehr geeignet.

6.2.6 Zusammenfassung

Die Implementierung „AMServer“ stellte gegenüber dem IFA-Prototypen eine erhebliche Verbesserung in der Funktionalität dar. Die durch die Erfahrungen mit dem IFA-Prototypen geklärten Anforderungen wurden bis auf den Punkt der Stabilität im wesentlichen erfüllt. Die mit dem AMServer gemachten Erfahrungen präzisierten die Anforderungen für den Entwurf der Folgeimplementierung AMEngine, der im folgenden Kapitel beschrieben ist.

KAPITEL 7 Entwurf der AMEngine

7.1 Erweiterte Anforderungen an die AMEngine

Durch die im vorigen Kapitel beschriebenen Erfahrungen mit dem Prototyp AMServer konnten die Anforderungen an das Audio-Subsystem präzisiert und ergänzt werden:

- Beibehalten der Effizienz und des guten Delay-Verhaltens
- Erweiterung der ECP-Schnittstelle für mehrere Steuerverbindungen
- Verbesserung der Stabilität
- Verbesserung des Verhaltens bei AVXP-Paketen beliebiger Größe
- Verbesserung der Debugging- und Monitoring-Möglichkeiten
- Verbesserung der Wartbarkeit und Erweiterbarkeit sowie der Wiederverwendbarkeit einzelner Module.

7.2 Entwurf

7.2.1 Entwurfsentscheidungen

Modularisierung

Die Verbesserung der Wartbarkeit und der Wiederverwendbarkeit von Programmteilen erforderte es, das Programm konsequent modular zu entwerfen und dabei die Schnittstellen zwischen den Modulen möglichst einfach zu halten. Um die Implementierung eines Moduls möglichst unabhängig von der Schnittstelle gestalten

zu können, sollten nach Möglichkeit abstrakte Datentypen benutzt werden.

Debugging / Monitoring

Bei der bisherigen Methode zur Ausgabe von Debugging-Information ließ sich nur die „Geschwätzigkeit“ des Systems global über einen Parameter einstellen. Dabei gingen oft die gewünschten Informationen über das Systemverhalten in einem Schwall momentan uninteressanter Meldungen unter. Dieser Punkt ist besonders deswegen von Bedeutung, weil Debug-Informationen von einem Menschen ausgewertet werden müssen, dessen Aufmerksamkeit und Motivation dazu neigen, negativ mit dem Umfang der zu sichtenden Datenmenge zu korrelieren.

Zur Verbesserung dieser Situation entschied ich mich, einen Mechanismus zur Ausgabe von Debugging-Informationen zu realisieren, der es erlaubt, Meldungen nach Themen zu gruppieren und diese Gruppen von Meldungen zur Laufzeit ein- und auszuschalten, und der dabei einfach erweiterbar ist.

Für die Überwachung einer laufenden AMEngine sollte es möglich sein, möglichst umfassend den Systemzustand abfragen zu können. Dazu gehört mindestens eine detaillierte Information über die existierenden Endpoints und Verbindungen. Welche Informationen außerdem interessant sein würden, sollte die weitere Entwicklung zeigen. Zusätzlich zu einer Auswertung durch einen Menschen ist angedacht, diese Informationen durch ein Programm zu lesen und aufzubereiten, etwa durch graphische Visualisierung oder statistische Auswertung. Deshalb entschied ich mich, die Ausgabe dieser Abfragen nicht nur menschen-, sondern auch maschinenlesbar zu gestalten.

Da die AMEngine schon aus anderen Gründen mehrere Steuerverbindungen unterhalten sollte, bot es sich an, Monitoring und Debugging ebenfalls über die Steuerschnittstelle zu realisieren. Die dafür nötigen Kommandos ließen sich als zusätzliche ECP-Requests einbinden. Eine Auflistung findet sich im Anhang „Endpoint Control Protocol“.

Tcl-Interpreter

Die neuen Konzepte für Debugging und Monitoring machten es wünschenswert, die Erweiterung des ECP-Kommandosatzes einfach zu gestalten.

Im zweiten Prototyp des Audio-Subsystems (AMServer) hatte ich für die ECP-Schnittstelle einen einfachen Parser implementiert. Um das Schnittstellenmodul zu vereinfachen und flexibler gestalten zu können, benutzte ich für die AMEngine einen Tcl-Interpreter (siehe Kapitel „Plattformen und Werkzeuge“).

Da das ECP bewußt einfach entworfen wurde, ließ es sich ohne weiteres in die Sprache Tcl integrieren; die Requests des ECP sind syntaktisch eine Untermenge von Tcl. Die ECP-Requests und weitere Kommandos werden dem Interpreter als neue Tcl-Kommandos bekannt gemacht.

Damit ist es möglich, im Interpreter eigene Tcl-Prozeduren zu definieren. Diese Prozeduren können neben der Grundfunktionalität des Interpreters auch die ECP-Kommandos benutzen. Diese Möglichkeit ist besonders zum Testen des Programms nützlich.

Eine Liste der für die AMEngine definierten ECP-Kommandos ist im Anhang „Endpoint Control Protocol“ zu finden.

Mehrere Steuerverbindungen

Die AMEngine hat grundsätzlich eine Steuerverbindung zu dem Prozeß, von dem sie gestartet wurde.

Zum Aufbau weiterer Steuerverbindungen zur AMEngine von anderen Prozessen aus muß die AMEngine über einen Mechanismus zur Interprozeßkommunikation erreichbar sein. Um die Komplexität des Programms gering zu halten, bietet es sich an, Sockets zu verwenden, da diese bereits für die Audio-Datenströme verwendet werden. Dazu akzeptiert die AMEngine auf einem bekannten Port eine TCP-Verbindung, über die dann eine ECP-Verbindung von der steuernden Instanz hergestellt wird. [Stevens 1990]

Die Verwendung von Sockets ermöglicht gleichzeitig den Zugriff auf eine AMEngine über das Netzwerk. Die Anzahl der ECP-Verbindungen ist nicht eingeschränkt.

Der Tcl-Interpreter kennt über die von der AMEngine definierten ECP-Kommandos viele andere Kommandos, zum Beispiel Dateioperationen. Kann nun jeder beliebige Prozeß auf eine laufende AMEngine zugreifen, besteht ein Sicherheitsrisiko: Von jedem Rechner am gleichen Netzwerk aus kann in beliebiger Weise auf die Dateien des Benutzers zugegriffen werden, der die AMEngine gestartet hat.

Hier ist eine Zugangskontrolle notwendig. Dazu wird im Filesystem des Rechners ein Schlüssel abgelegt, der nur für den Benutzer

zugänglich ist, unter dessen User ID die AMEngine läuft. Wird eine zusätzliche ECP-Verbindung zur AMEngine aufgebaut, fragt die AMEngine diesen Schlüssel ab. Wird ein falscher Schlüssel angegeben, bricht die AMEngine die Verbindung wieder ab.

Echtzeitverhalten

Um das Echtzeitverhalten weiter zu optimieren, wurde gegenüber dem AMServer die Nutzung der vom Betriebssystem bereitgestellten Möglichkeiten vorgenommen. Weil HP-UX wie die meisten Unix-artigen Betriebssysteme kein Echtzeitsystem ist, kann es nicht garantieren, daß ein Prozeß in Echtzeit arbeiten kann. HP-UX bietet jedoch immerhin Vorkehrungen, um die Chance zu erhöhen, daß ein Prozeß seine Echtzeitanforderungen erfüllt:

- Die Priorität des Prozesses kann mit dem Systemaufruf `rtprio`(2) auf eine sogenannte Echtzeitpriorität (*real-time priority*) gesetzt werden. Ein Prozeß mit einer *real-time priority* wird nicht von einem Prozeß mit einer *time-sharing priority* (die Prozesse üblicherweise haben) oder einer niedrigeren *real-time priority* unterbrochen.
- Der Adreßraum eines Prozesses kann mit dem Systemaufruf `plock`(2) im physikalischen Speicher verankert werden. Damit wird verhindert, daß der Adreßraum des Prozesses oder Teile davon bei Speicherengpässen auf die Festplatte ausgelagert werden.

[HP 1992]

Beide Möglichkeiten wurden bei der Realisierung der AMEngine berücksichtigt.

Audioformate

Coder und Decoder von Kodierungen wie zum Beispiel ADPCM oder GSM speichern für jeden Datenstrom gewisse Statusinformationen. Zur Vorbereitung auf die Verwendung solcher Kodierungen wurde die Schnittstelle von AIO geändert, um diesen Status an die Kodierungsprozeduren weitergeben zu können.

Es wurden jedoch bisher keine weiteren Audioformate implementiert.

Zur Unterstützung verschiedener Audio-Datentypen übernahm ich die schon im AMServer benutzte Lösung mit Abtastratenkonvertierung durch lineare Interpolation.

Zusammenfassung

Die wesentlichen Entwurfsentscheidungen für die AMEngine:

- konsequente Modularisierung; abstrakte Datentypen, wo möglich;
- Gruppen von Debugging-Meldungen ein- und ausschaltbar;
- Einbau von Monitoringfunktionen;
- Nutzung eines Tcl-Interpreters;
- Ermöglichen von mehreren Steuerverbindungen über TCP-Verbindungen;
- Nutzung der Möglichkeiten von HP-UX für Echtzeitverhalten;
- Vorbereitung auf weitere Kodierungen.

7.2.2 Architektur

Die AMEngine wurde in folgende Module aufgeteilt:

TABELLE 5 Module der AMEngine

Modul	Beschreibung
AIO:	Kapselung des Zugriffs auf die Audio-Hardware
AME_TCL:	Kapselung des Tcl-Interpreters (siehe unten)
AUDIO:	Bearbeitung der Audio-Datenströme
CONTROL:	ECP-Schnittstelle
DEBUG:	Unterstützende Funktionen für Debugging und Monitoring
ENDPOINTS:	ADTs: Endpoints und Verbindungen
ERRORS:	Prozeduren für Fehlermeldungen
GLOBALS:	Globale Variablen
MAIN:	Hauptprogramm, Bearbeitung der Kommandozeilen-Argumente, Initialisierungen
SBUF_QUEUES:	ADT: FIFO-Listen
SELECT_LOOP:	Select-Loop
SOCKETS:	Kapselung der Socket-Schnittstelle
UTILS:	Schlüsselverwaltung, Zeitstempel

Ein Bild zeigt die wichtigsten Aufrufbeziehungen:

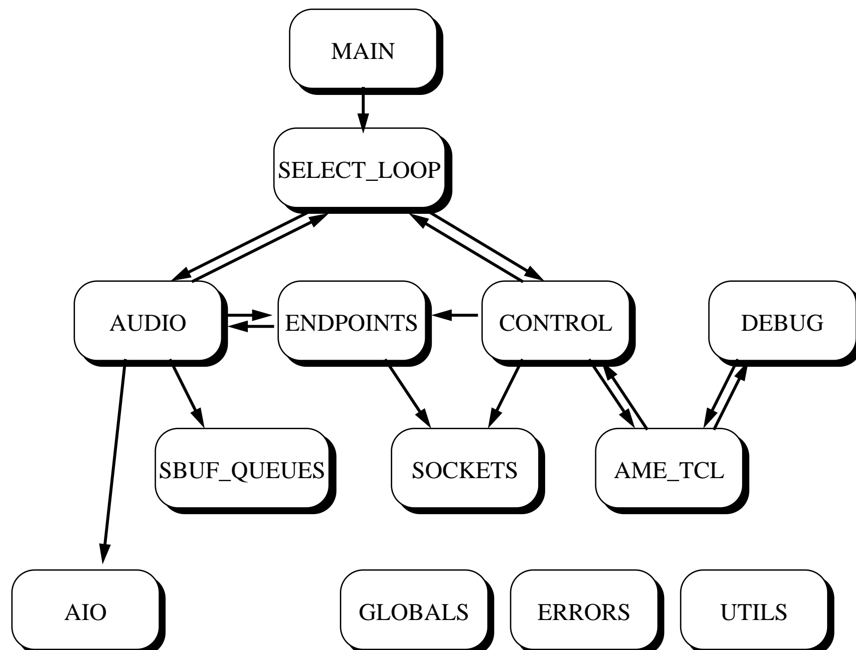

ABBILDUNG 7 Modulaufrufbeziehungen der AMEngine

- MAIN ruft nach den Initialisierungen der Module SELECT_LOOP auf.
- CONTROL und AUDIO melden Prozeduren zum Aufruf bei SELECT_LOOP an, die Ein-/Ausgabe-Ereignisse bearbeiten.
- SELECT_LOOP ruft diese Prozeduren aus AUDIO und CONTROL beim Eintreten von Ein-/Ausgabe-Ereignissen auf.
- DEBUG und CONTROL melden Tcl-Kommandos bei AME_TCL an.
- CONTROL ruft AME_TCL zur Bearbeitung der ECP-Kommandos auf.
- AME_TCL ruft Prozeduren zur Bearbeitung dieser Kommandos in CONTROL und DEBUG auf.
- CONTROL ruft Prozeduren in ENDPOINTS auf, um Endpoints zu manipulieren.
- ENDPOINTS ruft Prozeduren zur Manipulation von Audioströmen aus AUDIO auf.
- AUDIO ruft Prozeduren zum Schließen von Endpoints und zum Zugriff auf Audio-Verbindungen aus ENDPOINTS auf.

- AUDIO ruft Prozeduren zur Manipulation von FIFO-Listen aus SBUF_QUEUES auf.
- ENDPOINTS und CONTROL rufen zur Herstellung von Netzwerkverbindungen Prozeduren aus SOCKET auf.
- AUDIO ruft Prozeduren zum Zugriff auf die Audio-Hardware aus AIO auf.

7.2.3 Schnittstellen der Module

Zur besseren Verständlichkeit verzichte ich auf eine vollständige Beschreibung der Modulschnittstellen und führe in dieser Arbeit nur die wichtigsten auf. Die vollständigen Schnittstellen können in den Definitionsdateien der Programmquellen nachgelesen werden.

MAIN

In MAIN befindet sich das Hauptprogramm (`main()`), das beim Starten des Programms aufgerufen wird. Hier werden globale Initialisierungen durchgeführt, die Initialisierungsprozeduren der einzelnen Module aufgerufen, soweit nötig, und Argumente aus der Kommandozeile ausgewertet. Schließlich wird die Select-Loop aufgerufen.

Gemäß dem ANSI-C-Standard ist die äußere Schnittstelle von MAIN das Hauptprogramm:

```
int main(int argc, char *argv[]) ;
```

ERRORS

Das Modul ERRORS enthält Prozeduren zur Ausgabe und Formatierung von Fehlermeldungen sowie zur kontrollierten Beendigung des Programms bei nicht behebbaren Fehlern in der Initialisierungsphase oder beim Entdecken interner Inkonsistenzen. Vier verschiedene Typen von Fehlerbedingungen werden unterschieden:

- Interne Programmfehler
- Fehler bei Betriebssystemaufrufen
- Kommandofehler
- sonstige Fehler

```
char *ame_strerror(const char *format, ...) ;
void ame_perror(const char *format, ...) ;
volatile void cry_and_die(const char *format, ...) ;
```

Ein impliziter Parameter für die Prozeduren in ERRORS ist die globale Variable `ame_errno`. Analog zu `errno`(2) wird beim Entdecken einer Fehlerbedingung ein entsprechender Code in ame_errno geschrieben, bevor eine der oben angeführten Prozeduren aufgerufen wird.

`ame_strerror()` formatiert eine Fehlermeldung ähnlich wie `printf`(3) und liefert einen Zeiger auf die formatierte Meldung zurück. `ame_perror()` formatiert entsprechend, gibt aber die Meldung auf der Standardfehlerausgabe aus. `cry_and_die()` beendet nach der Ausgabe der Fehlermeldung das Programm. Die in GLOBALS definierte Variable `ame_errno` enthält einen Code für die Fehlerbedingung.

Bei internen Fehlern oder Kommandofehlern wird die Meldung von einem entsprechenden Präfix eingeleitet. Darauf folgt eine der Fehlerbedingung entsprechende Beschreibung und, falls vorhanden, in Klammern die formatierten Argumente der Funktion. Bei Systemfehlern folgt der Wert von `errno`(2) mit der entsprechenden Beschreibung.

Beispiel für einen Systemfehler:

```
bind socket failed (additional control). Syscall error
226: Address already in use
```

Beispiel für einen Kommandofehler:

```
Command error: unsupported connection type (xtp).
```

ERRORS definiert eine Reihe von Konstanten für die globale Fehlervariable `ame_error`. Die Werte sind in der Reference Manual Page `amengine`(1) beschrieben (siehe Anhang).

(Um die Meldung leichter maschinenlesbar zu machen, erscheint bei der Ausgabe der Meldungen über die Steuerverbindung in der formatierten Fehlermeldung zuerst der numerische Fehlercode. Diese Ausgabe wird Modul CONTROL vorgenommen.)

SELECT_LOOP

Wie im Abschnitt über den AMServer schon beschrieben, kommen über mehrere Kanäle verschiedene Arten von Ereignissen auf die AMEngine zu, die jeweils entsprechend behandelt werden müssen. Dazu dient unter UNIX der Systemaufruf `select`(2). Es bot sich an, eine Behandlung dieser Ereignisse in verallgemeinerter Form zu verwirklichen.

Die Ereignisse sind Ein-/Ausgabeanforderungen; technisch gesehen kann auf einem Filedeskriptor, der blockiert war, geschrieben oder gelesen werden oder es liegt eine Ausnahmebedingung für einen Filedeskriptor vor. (Dieser Fall wird in der AMEngine nicht benutzt.) Ein Ereignis wird dabei identifiziert durch den Kanal (Filedeskriptor), auf dem es eintritt, und durch seinen Typ (Eingabe, Ausgabe oder Ausnahme).

Für Ereignisse können Prozeduren (*Callbacks*) definiert werden, die aufgerufen werden, wenn das Ereignis eintritt. Dabei wird für jeden Callback ein Kontext mit angegeben, auf den er sich später beziehen kann; das ist ein Zeiger auf beliebige Daten (*Clientdata*), der dem Callback beim Aufruf als Argument zusammen mit dem Filedeskriptor übergeben wird.

Die Kontrolle über den Programmablauf hat dabei die Select Loop, nicht das „eigentliche" Programm. Dieses Prinzip ist als „Hollywood-Prinzip" bekannt („Don't call us, we'll call you"). [Sweet 85, zitiert nach Gamma et al. 95]

Um einheitlich aufgerufen werden zu können, haben alle Callback-Prozeduren den selben Typ:

```
int callback_procedure(int fd, void *clientdata) ;
```

`fd` ist der Filedeskriptor, bei dem das Ereignis aufgetreten ist. `clientdata` ist ein Zeiger auf ein beliebiges Datenobjekt, das der Callback-Prozedur als Kontext dient. Die Interpretation dieses Zeigers liegt bei der Callback-Prozedur; nicht bei allen wird dieses Argument benutzt.

Die Select-Loop kann drei auf verschiedene Ereignisse auf Filedeskriptoren reagieren:

1. Eine Eingabe kann ohne Blockieren gelesen werden,
2. eine Ausgabe kann ohne Blockieren geschrieben werden,
3. eine Ausnahmebedingung liegt vor.

Für jeden Filedeskriptor und jedes Ereignis kann mit `add_callback()` eine Prozedur (Callback) installiert werden. Dabei werden der Filedeskriptor, der Typ (Eingabe, Ausgabe oder Ausnahme), eine Priorität und ein Zeiger auf einen Kontext (`clientdata`) angegeben. Callbacks können mit `remove_callback()` wieder gelöscht werden.

Wenn ein Ereignis auf einem oder mehreren Filedeskriptoren eintritt, werden die entsprechenden Callbacks in der Reihenfolge ihrer

Priorität aufgerufen, niedrigste Priorität zuerst. Ein Callback bekommt den Filedeskriptor und den Clientdata-Zeiger übergeben.

Installierte Callbacks werden mit `activate_callback()` aktiviert und und mit `deactivate_callback()` deaktiviert. Ein deaktivierter Callback wird nicht aufgerufen, wenn das entsprechende Ereignis auf seinem Filedeskriptor eintritt. Callbacks sind nach dem Installieren zunächst aktiviert.

Normalerweise gibt ein Callback null zurück. Gibt er einen positiven Wert zurück, wird die Select-Loop neu gestartet, ohne daß die anderen Callbacks aufgerufen werden. Diese Methode ist für Callbacks vorgesehen, die andere Callbacks ändern. Gibt ein Callback einen negativen Wert zurück, terminiert die Select-Loop sofort und gibt diesen Wert zurück.

Schlägt der Systemaufruf `select`(2) aus einem anderen Grund als `EINTR` fehl, wird eine vorher definierte Fehlerbehandlungsprozedur aufgerufen, falls vorhanden. (Diese Möglichkeit wird von der AMEngine nicht benutzt.) Andernfalls wird `ame_errno` auf `E_SELECT` gesetzt, und `select_loop()` gibt -1 zurück. In `errno`(2) ist weitere Information zur Fehlerbedingung zu finden.

Die Select-Loop kann beendet werden, indem ein Callback die Prozedur `terminate_select()` aufruft. In diesem Fall wird die Schleife nach dem aktuellen Durchlauf beendet.

Die Prozeduren im Überblick:

```
int add_callback(int fd, void *clientdata,
                 unsigned int priority, int type,
                 int (*callback)(int fd,
                                 void *clientdata),
                 char *description) ;
int remove_callback(int fd, int type) ;
int activate_callback(int fd, int type) ;
int deactivate_callback(int fd, int type) ;
int select_loop(const struct timeval *tvp) ;
void terminate_select(void) ;
```

Die Prozeduren im einzelnen:

```
int add_callback(int fd, void *clientdata,
                 unsigned int priority, int type,
                 int (*callback)(int fd,
                                 void *clientdata),
                 char *description) ;
int remove_callback(int fd, int type) ;
```

Definieren und Entfernen eines Callbacks. `fd` ist der Filedeskriptor, `clientdata` der Zeiger auf einen Kontext, `priority` die Priorität, `type` ist eine der in SELECT_LOOP definierten Konstanten `SEL_INPUT`, `SEL_OUTPUT` oder `SEL_EXCEPTION` für Eingabe, Ausgabe und Ausnahmen, `callback` ist die aufzurufende Prozedur. `description` ist ein kurze Zeichenkette, die die Funktion dieses Callbacks beschreibt. Sie in der Ausgabe des ECP-Kommando „`callbacks`" mit angezeigt.

```
int activate_callback(int fd, int type) ;
int deactivate_callback(int fd, int type) ;
```

Aktivieren und Deaktivieren eines Callbacks. Es ist kein Fehler, einen aktiven Callback zu aktivieren bzw. einen inaktiven Callback zu deaktivieren.

```
void terminate_select(void) ;
```

Beenden der Select Loop.

```
int select_loop(const struct timeval *tvp) ;
```

Aufruf der eigentlichen Select Loop. Falls der Zeiger tvp nicht NULL ist, enthält die referenzierte Struktur einen Timeout-Wert für `select`(2).

Ist in einer der Prozeduren von SELECT_LOOP ein Fehler aufgetreten, enthält `ame_errno` (aus ERRORS) Information über die Fehlerbedingung analog zu `errno`(2). Die Fehlerbedingungen gliedern sich in drei Gruppen:

- Das Modul wurde nicht initialisiert,
- bei einem Aufruf wurden ungültige Argumente übergeben,
- der Systemaufruf `select`(2) lieferte einen Fehler.

SOCKETS

Im Modul SOCKETS werden die Systemaufrufe gekapselt, die das Netzwerk für die Einrichtung von Endpoints und das Akzeptieren zusätzlicher Steuerverbindungen ansprechen.

Die Prozeduren im Überblick:

```
int open_server_socket(int port, int ctype) ;
int open_client_socket(const char *hostname, int port,
int ctype) ;
```

```
int socket_accept_callback(int fd, void *clientdata) ;
```

Die Prozeduren im einzelnen:

```
int open_server_socket(int port, int ctype) ;
```

`open_server_socket()` erzeugt einen Server-Socket auf dem Port port mit dem Verbindungstyp `ctype`. `ctype` ist dabei eine der in `<sys/socket.h>` definierten Konstanten `SOCK_STREAM` (für TCP-Sockets) oder `SOCK_DGRAM` (für UDP-Sockets). Im Erfolgsfall wird der Filedeskriptor des Sockets zurückgegeben. Im Fehlerfall ist der Rückgabewert -1, und die Werte von `ame_errno` und `errno`(2) geben die Fehlerursache an.

```
int open_client_socket(const char *hostname, int port,
                       int ctype) ;
```

`open_client_socket()` erzeugt einen Client-Socket, mit dem Daten an den Port port auf dem Rechner hostname geschickt werden können. `ctype` ist eine der in `<sys/socket.h>` definierten Konstanten `SOCK_STREAM` (für TCP-Sockets) oder `SOCK_DGRAM` (für UDP-Sockets). Im Erfolgsfall wird der Filedeskriptor des Sockets zurückgegeben. Im Fehlerfall ist der Rückgabewert -1, und die Werte von `ame_errno` und `errno`(2) geben die Fehlerursache an.

Bei TCP-Server-Sockets werden Verbindungen explizit mit `accept`(2) angenommen, wenn ein Kommunikationspartner sich mit diesem Socket verbindet. Von `accept`(2) wird ein neuer Socket zurückgegeben, über den mit dem Kommunikationspartner Daten ausgetauscht werden können; über den ursprünglichen Socket können weiterhin Verbindungen angenommen werden. [Comer 1988, Stevens 1990, HP 1992]

Der erste Callback für einen TCP-Server-Socket, der `accept`(2) aufruft, wenn eine Verbindung von der Gegenseite initiiert wird, braucht daher Informationen über den endgültigen Callback, seine Eigenschaften, und dessen `clientdata`. Da ein Callback nur ein `clientdata`-Argument erhält, werden diese Informationen in einer Struktur namens „Callback-Deskriptor“ zusammengefaßt. Strukturen dieses Typs können verkettet werden; das `clientdata`-Feld der Struktur kann auf einen weiteren Callback-Deskriptor verweisen.

Über weitere Felder der Struktur kann bestimmt werden, ob der „accept“-Callback (der diese Struktur als Argument erhält) sich

deinstallieren soll und ob die Deskriptor-Struktur mit `free`(3) freigegeben werden soll.

Variiert werden kann das Verhalten mit dem Feld `call_at_once_p`: Ist es nicht null, wird die in `callback` angegebene Prozedur nicht als Callback installiert, sondern sofort aufgerufen. Diese Methode wird bei einer neuen Steuerverbindung angewendet, wo beim Zustandkommen einer neuen Verbindung zunächst eine Prozedur aufgerufen wird, die den Schlüssel liest und prüft, bevor sie (bei korrektem Schlüssel) die Prozedur als Callback installiert, die ECP-Kommandos liest und ausfuehrt.

```
typedef struct _cb_descriptor {
    int (*callback)(int fd, void *clientdata) ;
    void *clientdata ;
    char *description ;
    int priority ;
    int type ;
    int call_at_once_p ;
    int remove_accept_cb_p ;
    int free_descriptor_p ;
} cb_descr_t ;
```

Die Bedeutung der einzelnen Felder:

`callback`	Der zu installierende neue Callback
`clientdata`	Das clientdata-Argument für callback
`description`	Das description-Argument für add_callback()
`priority`	Die Priorität von callback
`type`	Der Typ von callback
`call_at_once_p`	Falls nicht null, callback sofort aufrufen
`remove_accept_cb_p`	Falls nicht null, accept-Callback deinstallieren
`free_descriptor_p`	Falls nicht null, Struktur mit free(3) freigeben

```
int socket_accept_callback(int fd, void *clientdata) ;
```

`socket_accept_callback()` erwartet als clientdata-Argument eine Struktur vom Typ `cb_descr_t` (Callback-Deskriptor). Anhand dieser Information wird ein neuer Callback installiert oder eine andere Prozedur aufgerufen.

SBUF_QUEUES

Vom AMServer übernahm ich das Konzept der Listen von Datenblöcken, die nach dem „*First In, First Out*“-Prinzip verwaltet wer-

den. Vom Modul SBUF_QUEUES werden diese Listen als abstrakter Datentyp zur Verfügung gestellt.

Die Definition des abstrakten Datentyps:

```
typedef struct _sbuf_queue *sbuf_q ;
```

Die Prozeduren im Überblick:

```
sbuf_q sbq_alloc(void) ;
void sbq_free(sbuf_q q) ;
void sbq_flush(sbuf_q q) ;
void sbq_flush_nonpartial(sbuf_q q) ;
char *sbq_append_buffer(sbuf_q q,
                        unsigned long timestamp,
                        audio_state_t *stateptr) ;
char *sbq_get_first(sbuf_q q) ;
audio_state_t *sbq_get_first_state(sbuf_q q) ;
void sbq_drop_buffer(sbuf_q q) ;
void sbq_copy(sbuf_q toq, const sbuf_q fromq) ;
int sbq_ispartial(sbuf_q q) ;
int sbq_length_nonpartial(sbuf_q q) ;
int sbq_cpin(sbuf_q q, const char *data,
             int datasize, aio_convfunc_t *copydata,
             unsigned long timestamp, void *addnl) ;
```

Die Prozeduren im einzelnen:

```
sbuf_q sbq_alloc(void) ; void sbq_free(sbuf_q q) ;
```

Analog zu `malloc`(3) und `free`(3) wird mit `sbq_alloc()` eine FIFO-Liste erzeugt und mit `sbq_free()` wieder freigegeben.

```
void sbq_flush(sbuf_q q) ;
void sbq_flush_nonpartial(sbuf_q q) ;
```

`sbq_flush()` entfernt alle Datenpuffer aus der FIFO-Liste `q`. `sbq_flush_nonpartial()` läßt dagegen den letzten in der Schlange `q`, wenn er nur teilweise gefüllt ist.

```
char *sbq_append_buffer(sbuf_q q,
                        unsigned long timestamp,
                        audio_state_t *stateptr) ;
```

`sbq_append_buffer()` hängt einen neuen Puffer an die FIFO-Liste `q` an. Der Puffer wird mit Werten für den Zeitstempel (`timestamp`, für ausgehende Datenströme) und einen Coder/Decoder-Status (falls `stateptr` nicht NULL ist) initialisiert. Ein Zeiger auf den

Datenbereich des Puffers wird zurückgegeben; in diesen Bereich kann die Datenkopier- bzw. -konvertierungsprozedur die Daten schreiben.

```
char *sbq_get_first(sbuf_q q) ;
```

`sbq_get_first()` gibt einen Zeiger auf den Datenbereich des ersten Puffers in der FIFO-Liste `q` zurück.

```
audio_state_t *sbq_get_first_state(sbuf_q q) ;
```

`sbq_get_first_state()` gibt einen Zeiger auf den Coder/Decoder-Status des ersten Puffers in der FIFO-Liste `q` zurück.

```
void sbq_drop_buffer(sbuf_q q) ;
```

`sbq_drop_buffer()` entfernt den ersten Puffer aus der FIFO-Liste `q`.

```
void sbq_copy(sbuf_q toq, const sbuf_q fromq) ;
```

`sbq_copy()` kopiert die Puffer der FIFO-Liste `fromq` in die FIFO-Liste `toq`. Die Puffer werden nicht physikalisch kopiert, aber danach von beiden FIFO-Listen referenziert. Vorher vorhandene Puffer in `toq` werden entfernt.

```
int sbq_ispartial(sbuf_q q) ;
```

`sbq_ispartial()` gibt 1 zurück, wenn der letzte Puffer in `q` nur teilweise gefüllt ist, ansonsten 0.

```
int sbq_length_nonpartial(sbuf_q q) ;
```

`sbq_length_nonpartial()` gibt die Anzahl der vollständig gefüllten Puffer in `q` zurück.

```
int sbq_cpin(sbuf_q q, const char *data, int datasize,
             aio_convfunc_t *copydata,
             unsigned long timestamp, void *addnl) ;
```

`sbq_cpin()` kopiert mit der gegebenen Prozedur `copydata` Daten in die FIFO-Liste `q`. `copydata` ist eine Datenkonvertierungsprozedur des in AIO definierten Typs `aio_convfunc_t`. `datasize` ist die Menge der Daten `nach` der Konvertierung. `addnl` ist ein Zeiger auf den Coder/Decoder-Status. Ein Puffer wird *vor* dem Füllen mit die-

sem Status markiert. Die Konvertierungsprozedur erhält den Zeiger auf den Status als Argument. Die resultierenden Puffer werden mit dem Zeitstempel `timestamp` markiert.

AUDIO

Das Modul AUDIO kapselt den Zugriff auf die Audio-Datenströme. Audio-Datenströme sind die Datenströme von eingehenden und ausgehenden Netzwerkverbindungen. Audio-Datenströme haben einen Typ, der durch die Abtastrate, die Kodierung und die Zahl der Kanäle festgelegt wird. Sie enthalten die ihrem Typ entsprechenden Konvertierungsfunktionen für die Umwandlung von der externen in die interne Kodierung und umgekehrt, sowie die FIFO-Liste mit Puffern von Audiodaten.

Für jeden Audio-Datentyp existiert ein vordefinierter Audiostrom. Diese Audioströme dienen als Prototypen für die Audioströme, über die in den Endpoints der Datenfluß verläuft: Bei der Erzeugung eines Endpoints wird der dem Audio-Datentyp des Endpoints entsprechende vordefinierte Audiostrom kopiert. Diese Kopie dient wiederum als Prototyp für die Datenströme für die Verbindungen dieses Endpoints. (Diese Methode entspricht in etwa dem Prototyp-Pattern in [Gamma et al. 1995].)

Da die Datenströme alle für ihren Typ spezifische Information enthalten, brauchen die Endpoints diesen Typ nicht zu kennen.

Neben den Prozeduren zur Bearbeitung von Audioströmen enthält das Modul AUDIO Prozeduren, die als Callbacks von der Select-Loop aufgerufen werden. Sie bearbeiten Ereignisse, die das Audio-Device und die Netzwerkverbindungen betreffen.

Die Definition des abstrakten Datentyps:

```
typedef struct _audio_stream *astream_t ;
```

Die Prozeduren im Überblick:

```
void audio_list_types(void) ;
astream_t audio_find_type(int sample_rate,
                          int data_format, int stereo) ;
astream_t audio_dup_astream(astream_t prototype,
                            int needs_queue,
                            int src_conn_p,
                            int has_headers) ;
int audio_stream_has_headers(astream_t astream) ;
int audio_recv_udp_cb(int fd, void *clientdata) ;
```

```
int audio_recv_tcp_cb(int fd, void *clientdata) ;
int audio_send_cb(int fd, void *clientdata) ;
int audio_play_cb(int fd, void *clientdata) ;
int audio_record_cb(int fd, void *clientdata) ;
void audio_list_types(void) ;
```

`audio_list_types()` listet die vorhandenen Audio-Datentypen auf. Für jeden Datentyp wird eine Zeile in das Rückgabefeld des Tcl-Interpreters geschrieben, die die Abtastrate, die Kodierung und die Zahl der Kanäle auffuehrt.

```
astream_t audio_find_type(int sample_rate,
                          int data_format, int stereo) ;
```

`audio_find_type()` sucht den vordefinierten Audiostrom mit der Abtastrate sample_rate, der Kodierung data_format; in Stereo, falls stereo null ist, ansonsten in Mono. Existiert dieser Audiostrom, wird er zurückgegeben; exitiert er nicht, wird ein NULL-Zeiger zurückgegeben.

```
astream_t audio_dup_astream(astream_t prototype,
                            int needs_queue,
                            int src_conn_p,
                            int has_headers) ;
```

`audio_dup_astream()` erzeugt einen Audiostrom mit dem in prototype angegebenen Audio-Datentyp (Abtastrate, Kodierung, Zahl der Kanäle). Falls `needs_queue` ungleich null ist, erhält der erzeugte Datenstrom eine FIFO-Liste. (Für einen Audiostrom, der selbst wieder als Prototyp dienen soll, ist keine FIFO-Liste erforderlich.) Ist `src_conn_p` ungleich null, soll der erzeugte Audiostrom für eine ausgehende Verbindung verwendet werden. Ist `has_headers` ungleich null, sollen auf der Verbindung Pakete mit AVXP-Headern verwendet werden.

```
int audio_stream_has_headers(astream_t astream) ;
```

`audio_stream_has_headers()` gibt einen Wert ungleich null zurück, falls auf der Verbindung mit dem Audiostrom `astream` AVXP-Header benutzt werden sollen.

Die folgenden Prozeduren dienen als Callbacks für das Senden und Empfangen von Netzwerkpaketen und das Aufnehmen und Abspielen von Audiodaten über AIO. Sie werden von der Select-Loop aufgerufen.

```
int audio_recv_udp_cb(int fd, void *clientdata) ;
```

`audio_recv_udp_cb()` wird aufgerufen, sobald ein Paket auf einer eingehenden UDP-Audio-Verbindung gelesen werden kann. `fd` ist der Filedeskriptor für die Verbindung. In clientdata wird ein Zeiger auf die Verbindung (Typ `conn_ptr_t`) erwartet. `audio_recv_udp_cb()` liest die empfangenen Daten, konvertiert sie in das interne Datenformat und reiht sie in die FIFO-Liste des entsprechenden Sink Endpoints ein.

```
int audio_recv_tcp_cb(int fd, void *clientdata) ;
```

`audio_recv_tcp_cb()` wird aufgerufen, sobald Daten auf einer eingehenden TCP-Audio-Verbindung gelesen werden können. `fd` ist der Filedeskriptor für die Verbindung. In clientdata wird ein Zeiger auf die Verbindung (Typ `conn_ptr_t`) erwartet. `audio_recv_tcp_cb()` liest die empfangenen Daten, konvertiert sie in das interne Datenformat und reiht sie in die FIFO-Liste des entsprechenden Sink Endpoints ein.

```
int audio_send_cb(int fd, void *clientdata) ;
```

`audio_send_cb()` wird aufgerufen, sobald Daten auf eine ausgehende Audio-Verbindung geschrieben werden können. In `clientdata` wird eine Verbindung (Typ `conn_ptr_t`) erwartet. `audio_send_cb()` nimmt einen Puffer aus der FIFO-Liste der Verbindung und sendet die enthaltenen Daten über die Verbindung.

```
int audio_play_cb(int fd, void *clientdata) ;
```

`audio_play_cb()` wird aufgerufen, wenn Daten auf dem Audio-Device abgespielt werden können. `fd` ist der Filedeskriptor für das Audio-Device. Das Argument clientdata wird nicht benutzt. `audio_play_cb()` nimmt je einen Puffer aus den FIFO-Listen aller Sink Endpoints, mischt die enthaltenen Audiodaten und spielt sie über das Audio-Device ab.

```
int audio_record_cb(int fd, void *clientdata) ;
```

`audio_record_cb()` wird aufgerufen, wenn Daten vom Audio-Device aufgenommen werden können. `fd` ist der Filedeskriptor für das Audio-Device. Das Argument clientdata wird nicht benutzt. `audio_record_cb()` liest Audiodaten vom Audio-Device (über AIO), konvertiert sie in die Datentypen der benutzten Audioströme

und überträgt die Daten in die FIFO-Listen der ausgehenden Verbindungen.

DEBUG

DEBUG stellt den anderen Modulen eine Prozedur zur Verfügung, um Meldungen zu Debugging-Zwecken auszugeben. Jede Meldung ist dabei einem benannten Thema zugeordnet; die Ausgabe der Meldungen zu einem Thema kann zur Laufzeit über ein ECP-Kommando aktiviert und deaktiviert werden.

Die Menge der Themen ist erweiterbar. Sie wurde nicht vor der Implementierung festgelegt, sondern währenddessen nach Bedarf erweitert.

Definierte Themen:

`callbacks`	Aufruf von Callback-Prozeduren
`control`	Aufruf von ECP-Prozeduren
`commands`	Aufruf von sonstigen Kommandos
`fdsets`	Filedeskriptor-Mengen vor und nach select(2)
`network`	Senden und Empfangen von Audio-Paketen
`queuef`	Aufruf von Prozeduren aus SBUF_QUEUES

exportierte Prozedur:

```
extern void dbg_dprintf(int topic, char *format, ...) ;
```

`dbg_dprintf()` gibt seine Argumente ähnlich wie `printf`(3) aus, wenn die Ausgabe des Themas `topic` aktiviert ist. Die Ausgabe erfolgt normalerweise über die Standardfehlerausgabe, kann aber auch über das Kommando „`debug`“ (siehe Anhang Endpoint Control Protocol) auf die Steuerverbindung umgelenkt werden.

ENDPOINTS

Im Modul ENDPOINTS werden Endpoints und die ihnen zugeordneten Verbindungen verwaltet. Entsprechend dem MMC-Modell zur audiovisuellen Kommunikation wird einem Sink Endpoint bei der Erzeugung sofort eine Verbindung zugeordnet (die allerdings erst später aufgebaut wird), einem Source Endpoint dagegen erst später nach Bedarf.

Nach ihrer Erzeugung werden Endpoints durch eine „Endpoint ID“ referenziert. Dieselbe Endpoint ID wird auch im ECP benutzt.

Für die Verbindungen, die Endpoints zugeordnet sind, wird ein abstrakter Datentype definiert:

```
typedef struct _connection *conn_ptr_t ;
```

Die Prozeduren im Überblick:

```
int ep_opensource(astream_t astream, int ctype,
                  int has_headers, int flow_control) ;
int ep_opensink(astream_t astream, int ctype,
                int *portp, int has_headers,
                int flow_control) ;
int ep_close(int endpoint) ;
int ep_closeall(void) ;
int ep_addsink(int endpoint, const char *host,
               int port) ;
int ep_remsink(int endpoint, const char *host,
               int port) ;
int ep_transmit(int endpoint, int flag) ;
```

Beispiele für Prozeduren zur Manipulation von Verbindungen:

```
astream_t conn_get_astream(conn_ptr_t cp) ;
int conn_incr_packets(conn_ptr_t cp) ;
```

```
int ep_opensource(astream_t astream, int ctype,
                  int has_headers, int flow_control) ;
int ep_opensink(astream_t astream, int ctype,
                int *portp, int has_headers,
                int flow_control) ;
```

`ep_opensource()` öffnet einen Source Endpoint, `ep_opensink()` einen Sink Endpoint. Der Endpoint bekommt den Audiostrom astream als Prototypen für seine Verbindungen. Seine Verbindungen sind vom Typ `ctype` (SOCK_STREAM für TCP, SOCK_DGRAM für UDP als Transportprotokoll). Wenn `has_headers` ungleich null ist, werden auf den Verbindungen AVXP-Header benutzt. Wenn `flow_control` ungleich null ist, wird auf den Verbindungen Flußkontrolle benutzt, falls das verwendete Transportprotokoll das zuläßt. `portp` ist ein Zeiger auf eine Integervariable, in die von `ep_opensink()` die Portnummer des Sink Endpoints geschrieben wird.

Konnte der Endpoint eingerichtet werden, geben `ep_opensource()` und `ep_opensink()` die Nummer des Endpoints zurück (eine kleine positive Zahl), andernfalls wird -1 zurückgegeben. In diesem Fall gibt der Wert von `ame_errno` die Fehlerursache an.

```
int ep_close(int endpoint) ;
int ep_closeall(void) ;
```

`ep_close()` schließt den angebenen Endpoint, `ep_closeall()` schließt alle Endpoints. Die Verbindungen der betroffenen Endpoints werden abgebaut.

```
int ep_addsink(int endpoint, const char *host,
               int port) ;
int ep_remsink(int endpoint, const char *host,
               int port) ;
```

`ep_addsink()` baut die beschriebene Verbindung vom Source Endpoint endpoint zu einem Sink Endpoint auf, `ep_remsink()` baut sie ab. `host` ist der Name des Rechners, auf dem sich der Sink Endpoint der Gegenseite befindet, `port` die Portnummer des Sink Endpoints.

```
int ep_transmit(int endpoint, int flag) ;
```

`ep_transmit()` schaltet die Übertragung von Daten über den betreffenden Endpoint ab, wenn `flag` gleich null ist, oder an, wenn `flag` ungleich null ist.

Für Audioverbindungen wird der abstrakte Datenyp `conn_ptr_t` definiert:

```
typedef struct _connection *conn_ptr_t ;
```

Die Prozeduren zum Zugriff auf diesen Datentyp dienen zum Zugriff auf die Attribute einer Verbindung. Diese Attribute können gelesen werden:

- Der Rechner der Gegenseite (nur bei Verbindungen an Source Endpoints)
- Der Port des Sink Endpoints der Verbindung (bei ausgehenden Verbindungen der Port der Gegenseite, bei eingehenden der eigene)
- Der Socket-Filedeskriptor der Verbindung
- Der zur Verbindung gehörende Audiostrom
- Die letzte gelesene Paketnummer (bei eingehenden Verbindungen)
- Der Endpoint, zu dem die Verbindung gehört
- Transmit-Status der Verbindung
- Benutzung von Flußkontrolle

Als Beispiel zeige ich hier die Prozedur, um den Audiostrom der Verbindung zu erhalten:

```
astream_t conn_get_astream(conn_ptr_t cp) ;
```

Folgende Attribute können gesetzt werden:

- Filedeskriptor der Verbindung (wichtig für eingehende TCP-Verbindungen, da hier `accept`(2) einen neuen Filedeskriptor zurückgibt)
- Die letzte gelesene Paketnummer

Die Zähler für gesendete Pakete (bei ausgehenden Verbindungen) und verworfene Pakete (bei eingehenden Verbindungen) können erhöht werden. Beispiel:

```
int conn_incr_packets(conn_ptr_t cp) ;
```

`conn_incr_packets()` erhöht den Paketzähler der Verbindung `cp` um eins.

GLOBALS

GLOBALS enthält Variablen, die zum gesamten Programm global sind:

- Default-Internet-Adresse des eigenen Rechners,
  ```
  extern int ame_errno ;
  ```
- eine globale Fehlervariable entsprechend zu `errno`(2).
  ```
  extern unsigned long own_ipaddr ;
  ```

UTILS

Das Modul UTILS wurde als Sammelbecken für Prozeduren vorgesehen, die keinen Bereich bilden, der groß genug ist, um dafür ein eigenes Modul zu rechtfertigen. Das sind

- Prozeduren zur Verwaltung des Zugangsschlüssels für ECP-Verbindungen und
- eine Prozedur zur Erzeugung von Zeitstempel, die für AVXP-Pakete gebraucht werden.

Die Prozeduren im Überblick:

```
void ut_generate_and_deposit_key(void) ;
int ut_read_and_check_key(int fd) ;
unsigned long ut_make_timestamp(void) ;
```

Die Prozeduren im einzelnen:

```
void ut_generate_and_deposit_key(void) ;
```

`ut_generate_and_deposit_key()` generiert einen Schlüssel und legt ihn im Home-Directory des Benutzers in der Datei `.ame-key.`*hostname* ab, wobei *hostname* der Name des Rechners ist, auf dem die AMEngine läuft.

```
int ut_read_and_check_key(int fd) ;
```

`ut_read_and_check_key()` liest einen Schlüssel vom angegebenen Filedeskriptor `cd` und vergleicht ihn mit dem vorher erzeugten. Sind die Schlüssel gleich, wird ein Wert ungleich null zurückgegeben, ansonsten null.

```
unsigned long ut_make_timestamp(void) ;
```

`ut_make_timestamp()` erzeugt einen Zeitstempel mit einer Auflösung von Millisekunden, wie er für das Feld „Timestamp“ im Header von AVXP-Paketen vorgesehen ist. Da der Wert auf eine Wortbreite von 32 Bit beschränkt ist, wiederholt er sich etwa alle sieben Wochen.

AIO

Da sich das entwickelte Interface zur Audio-Hardware als erfolgreich erwiesen hatte, wurde es beibehalten. Lediglich die Schnittstelle zu den Konvertierungsfunktionen wurde geändert, um einen Status übergeben zu können.

Die Prozeduren im Überblick:

```
int aio_get_output_port(void) ;
int aio_set_output_port(int port) ;
int aio_get_input_port(void) ;
int aio_set_input_port(int port) ;
int aio_get_left_play_gain(void) ;
int aio_get_right_play_gain(void) ;
int aio_set_play_gain(int gain) ;
int aio_set_play_gain2(int lgain, int rgain) ;
int aio_get_sample_rate(void) ;
int aio_set_sample_rate(int rate) ;
int aio_get_device_fd(void) ;
int aio_begin_record(void) ;
int aio_stop_record(void) ;
int aio_begin_play(void) ;
int aio_stop_play(void) ;
int aio_begin_both(void) ;
int aio_stop_both(void) ;
```

```
int aio_play_buffer(const char *buffer, int len) ;
int aio_record_buffer(char *buffer, int len) ;
```

Die Prozeduren im einzelnen:

```
int aio_get_output_port(void) ;
int aio_set_output_port(int port) ;
int aio_get_input_port(void) ;
int aio_set_input_port(int port) ;
```

Auslesen und Setzen des Ein- bzw. Ausgabekanals. Der Parameter port ist dabei eine der dafür in `<sys/audio.h>` definierten Konstanten.

```
int aio_get_left_play_gain(void) ;
int aio_get_right_play_gain(void) ;
nt aio_set_play_gain(int gain) ;
int aio_set_play_gain2(int lgain, int rgain) ;
```

Auslesen bzw. Setzen der Wiedergabelautstärke. `aio_set_play_gain()` setzt dabei beide Kanäle auf den gleichen Wert; mit `aio_set_play_gain2()` lassen sich die Werte für den linke und den rechten Kanal getrennt setzen. Dabei steht der Parameter `lgain` für die Lautstärke des linken Kanals, `rgain` für die des rechten Kanals. Gleichartige Prozeduren existieren für die Aufnahmelautstärke und die Lautstärke des Durchschleifens vom Eingang zum Ausgang („Monitor“-Wert).

```
int aio_get_sample_rate(void) ;
int aio_set_sample_rate(int rate) ;
```

Auslesen und Setzen der Abtastrate. Die Abtastrate ist dabei eine der mit `aio_describe()` erhaltenen. Wird die Abtastrate gesetzt, bevor das Audio Device zum Lesen oder Schreiben geöffnet wird, kann die Gültigkeit der Abtastrate nicht sofort überprüft werden; das geschieht erst, wenn das Device zum Lesen oder Schreiben geöffnet wird (`aio_begin*()`). Dieser Aufruf kehrt dann bei einer ungültigen Abtastrate mit dem entsprechenden Fehlerwert zurück.

Gleichartige Prozeduren existieren für das Setzen und Auslesen der verwendeten Kodierung und die Zahl der Kanäle.

```
int aio_get_device_fd(void) ;
```

Liefert den Filedeskriptor für das Audio Device zurück. Die Verwendung beziehungsweise das Anbieten dieser Prozedur ist in gewisser

Weise eine Verletzung des Geheimnisprinzips (*Information Hiding*), aber für die Benutzung von `select`(2) unbedingt notwendig. Eine Alternative wäre gewesen, die Select Loop in AIO zu integrieren. Das hätte aber die Verwendung von AIO zusammen mit anderen Paketen unmöglich gemacht, wenn diese selbst eine Select Loop benutzen, wie zum Beispiel das X Toolkit [Nye, O'Reilly 1992].

```
int aio_begin_record(void) ;
int aio_stop_record(void) ;
int aio_begin_play(void) ;
int aio_stop_play(void) ;
int aio_begin_both(void) ;
int aio_stop_both(void) ;
```

Öffnen und Schließen des Audio Device zum Aufnehmen, zum Wiedergeben oder für beides. Ab dem Aufruf von `aio_begin_record()` oder `aio_begin_both()` werden vom A/D-Wandler Daten aufgezeichnet und stehen zum Lesen zur Verfügung, zumindest, bis ein interner Puffer überläuft.

```
int aio_play_buffer(const char *buffer, int len) ;
int aio_record_buffer(char *buffer, int len) ;
```

Aufnehmen und Wiedergeben eines Datenblocks über das Audio Device.

```
void aio_mix_buffers_16bit(const char *bufv[],
                           int bufc, int size,
                           char *res_buf) ;
```

Mischen von Audiodaten. `bufv` ist ein Array von `char` Zeigern, die auf die zu mischenden Datenblöcke zeigen, `bufv` die Anzahl dieser Blöcke, `size` deren Größe. `res_buf` zeigt auf einen Block, in den die resultierenden Daten geschrieben werden. Die Daten müssen im Format linear (16 bit) vorliegen.

Die Prozeduren zur Konvertierung von Blöcken von Audiodaten haben eine einheitliche Schnittstelle, die in dem dafür definierten Typ `aio_convfunc_t` zu sehen ist. Die ersten drei Argumente sind analog zur Prozedur `memcpy`(3): `to` zeigt auf den Block der Zieldaten, `from` auf die Ausgangsdaten, `out_size` gibt die Menge der Daten *nach* der Konvertierung an. `channels` gibt nach den in `<sys/audio.h>` definierten Flags `AUDIO_CHANNEL_LEFT` und `AUDIO_CHANNEL_RIGHT` an, welche Kanäle an der Konvertierung beteiligt sein sollten. Der Zeiger `addnl` zeigt auf ein beliebiges Datenfeld; hier wird für statusabhängige Kodierungen der Coder-/Decoderstatus referenziert.

```
typedef int aio_convfunc_t(char *to, const char *from,
                           int out_size, int channels,
                           void *addnl) ;
```

Für folgende Konvertierungen existieren Prozeduren:

- Identität (nur kopieren),
- jede Kodierung in jede (unabhängig von der Zahl der Kanäle und der Abtastrate),
- mono nach stereo und umgekehrt für 16-bit-Werte (linear) und 8-bit-Werte (ulaw, alaw),
- linear stereo nach ulaw mono und umgekehrt,
- linear stereo nach ulaw mono mit Halbierung der Abtastrate,
- ulaw mono nach linear stereo mit Verdopplung der Abtastrate.

```
extern int aio_errno ;
```

Globale Fehlervariable; hier steht im Fehlerfall ein in `aio.h` definierter Code, der die Fehlerursache beschreibt. Bei fehlgeschlagenen Betriebssystemaufrufen kann zusätzliche Information in der systemglobalen Fehlervariablen `errno`(2) gefunden werden.

```
extern char aio_errstr[] ;
```

Im Fehlerfall steht in dieser Variablen eine textuelle Beschreibung des Fehlers.

```
int (*aio_set_error_handler(int (*new_handler)
                                        (void)))(void) ;
```

Setzen einer Fehlerbehandlungsprozedur. Die vorige Fehlerbehandlungsprozedur wird zurückgegeben. Die Prozedur `new_handler` wird danach beim Auftreten einer Fehlerbedingung in einem AIO-Aufruf aufgerufen. So läßt sich in vielen Fällen in den aufrufenden Modulen eine lästige explizite Fehlerbehandlung für jeden AIO-Aufruf sparen.

```
int aio_print_error_and_exit(void) ;
```

Beispiel für eine Fehlerbehandlungsprozedur: Ausgeben einer Fehlermeldung (`aio_errstr`) und Beenden des Programms.

CONTROL

Die Prozeduren zur Behandlung der ECP-Schnittstelle finden sich im Modul CONTROL. Zentral ist die Prozedur zum Lesen und Auswerten von ECP-Requests. Sie wird von der Select-Loop aufgerufen, wenn ein ECP-Request von der Schnittstelle gelesen werden kann. Der gelesene Request wird vom Tcl-Interpreter ausgewertet; dieser ruft daraufhin Prozeduren aus CONTROL auf, um den entsprechenden Request auszuführen. Für jeden ECP-Request gibt es ein Tcl-Kommando, das in CONTROL implementiert ist.

Das Modul CONTROL hat eine zweigeteilte Schnittstelle:

- Prozeduren zum Aufruf durch den Tcl-Interpreter. Diese Prozeduren implementieren die Requests des ECP.
- Exportierte Prozeduren.

Prozeduren zum Aufruf durch den Tcl-Interpreter

Die vom Tcl-Interpreter aufgerufenen Prozeduren haben alle den Typ `atcl_command_t`, der in der Schnittstelle von AME_TCL definiert ist:

```
typedef int atcl_command_t(int argc, char *argv[],
                           void *clientdata) ;
```

Ein Beispiel:

```
static atcl_command_t ctl_openep ;
```

`ctl_openep()` (kurz für „Open Endpoint“) implementiert die ECP-Requests `opensink` und `opensource`.

Diese Prozeduren werden nicht exportiert in dem Sinne, daß sie anderen Modulen namentlich bekannt sind, sondern ihre Adresse wird mit dem Aufruf von `atcl_create_command()` dem Tcl-Interpreter bekannt gemacht, der sie dann darüber aufruft.

Exportierte Prozeduren

Diese Prozeduren werden anderen Modulen namentlich bekanntgemacht.

```
void ctl_async_error(int condition,
                     const char *message) ;
```

Melde eine asynchron aufgetretene Fehlerbedingung über die ECP-Schnittstelle.

```
void ctl_event_sinkdropped(char *host, int port,
                           int error) ;
```

Melde den Event SSCP_SinkDropped (siehe Einleitung) über die ECP-Schnittstelle.

```
int ctl_current_fd(void) ;
```

Gib den für die aktuelle Steuerverbindung benutzten Filedeskriptor zurück. Diese Prozedur wird vom Debug-Kommando „`debug here`" (siehe Anhang „Endpoint Control Protocol") benutzt, um die Ausgabe von Debugging-Meldungen auf die Steuerverbindung umzulenken.

AME_TCL

Das Modul AME_TCL kapselt das Tcl-API für die Verwendung in der AMEngine. Da die AMEngine nicht den vollen Umfang des Tcl-APIs benötigt, konnte ich die Schnittstelle vereinfachen. Zusätzlich wird weitere Funktionalität hinzugefügt: Alle Tcl-Kommandos, die die AMEngine zur Verfügung stellt, sind mit dem ECP-Kommando „`help`" in einer Liste abrufbar.

```
int atcl_create_command(char *name,
                        atcl_command_t *cmdproc,
                        void *clientdata,
                        int minargs, int maxargs,
                        char *usage) ;
```

`atcl_create_command()` ist eine Kapselung von `Tcl_CreateCommand()` [Ousterhout 1994] und erzeugt ein neues Tcl-Kommando im Tcl-Interpreter. `name` ist der Name des Tcl-Kommdos, `cmdproc` ein Zeiger auf die Prozedur, die dieses Kommando implementiert. `clientdata` ist ein Zeiger auf beliebige Daten, der `cmdproc` beim Aufruf über den Tcl-Interpreter mitgegeben wird.

Über `Tcl_CreateCommand()` hinausgehende Funktionalität wird mit Hilfe der Parameter `minargs`, `maxargs` und `usage` verwirklicht: `minargs` gibt die minimale Anzahl an Argumenten an, `maxargs` die maximale. Ist `maxargs` kleiner null, ist die Anzahl der Argumente nicht nach oben begrenzt. `usage` ist ein String mit einer Beschreibung der Agumente und wird vom ECP-Kommando „`help`" benutzt.

```
void atcl_append_result(char *, ...) ;
void atcl_result_printf(const char * format, ...) ;
```

Diese Prozeduren dienen zur Rückgabe von Ergebnissen an den Tcl-Interpreter, der sie an CONTROL zurückgibt, und werden von den Prozeduren benutzt, die die ECP-Kommandos implementieren. Existiert bereits ein Rückgabewert, das heißt, wurde eine der beiden Prozeduren während der Ausführung des aktuellen Kommandos schon einmal aufgerufen, wird der neue Wert angehaengt. `atcl_append_result()` fügt dem Rückgabewert eine beliebige Anzahl von Strings hinzu. Das letzte Argument muß ein NULL-Zeiger sein. `atcl_result_printf()` formatiert seine Argumente wie `printf`(3).

```
int atcl_eval(char *cmd) ;
int atcl_eval_file(char *filename) ;
```

`atcl_eval()` und `atcl_eval_file()` führen Tcl-Kommandos aus. Das Argument `cmd` wird als String interpretiert, der Tcl-Kommandos enthält, `filename` als Name einer Datei, die die auszuführenden Tcl-Kommandos enthält.

```
hashtable_t atcl_create_hashtable(void) ;
void atcl_set_hash_entry(hashtable_t hashtab,
                         char *key, void *value) ;
int atcl_hash_entry_exists_p(hashtable_t hashtab,
                             char *key) ;
void *atcl_get_hash_value(hashtable_t hashtab,
                          char *key) ;
```

Zum Zugriff auf die Hash-Tabellen des Tcl-Interpreters gibt es eine vereinfachte Schnittstelle. Es werden nur mit Strings indizierte Hash-Tabellen benutzt, und weder Tabellen noch einzelne Einträge werden gelöscht. Der Wert ist immer ein Zeiger.

Das Argument `hashtab` ist eine mit `atcl_create_hashtable()` erzeugte Hash-Tabelle, `key` der Schlüssel, `value` der zugewiesene Wert. `atcl_set_hash_entry()` erzeugt einen Eintrag oder verändert einen vorhandenen; `atcl_hash_entry_exists_p()` prüft, ob ein Eintrag existiert; `atcl_get_hash_value()` liefert den Wert eines Eintrags. Falls er nicht existiert, wird ein NULL-Zeiger zurückgegeben. Da das gleichzeitig auch ein Wert eines Eintrags sein kann, ist es eine gute Idee, zu wissen oder zu prüfen, ob dieser Eintrag existiert.

7.2.4 Aufrufoptionen

Beim Start der AMEngine können auf der Kommandozeile verschiedene Optionen angegeben werden, die das Verhalten der AMEngine steuern. Diese Optionen sind im Anhang „Reference Manual Page“ im einzelnen beschrieben. Ich nenne hier nur die wichtigsten:

`-version`

Gib ausführliche Versionsinformationen aus.
Um beim Testen, aber auch bei Fehlerberichten von Anwendern verifizieren zu können, daß die korrekte Version der AMEngine benutzt wird, gibt die AMEngine mit dieser Option ausführliche Informationen aus über die AMEngine-Version, AIO-Version, Kompilierungszeit, benutzten Compiler sowie die bei der Kompilierung benutzten Compiler- und Linkeroptionen. Danach terminiert das Programm.

Ein Beispiel:

```
AIO: aio-1.3pre6 (Sun Aug 20 15:52:43 1995 by nickel@prz.tu-berlin.de)
Build: Dec 27 1995, 21:36:37 on keitum.prz.tu-berlin.de
CC: c89 -z -D_HPUX_SOURCE
CFLAGS: -I/imports/teleservices/mmc/avc2/development/include -I/usr/contrib/
packages/tcl/include -DHPUX -g -DDEBUG
LDFLAGS: -g
```

`-init` *initfile*

Lies die Datei *initfile* und werte den Inhalt als ECP-Kommandos aus.
Zum Testen lassen sich mit dieser Option schon beim Start der AMEngine Tcl-Prozeduren definieren, Endpoints öffnen etc.

`-control` *controlfd*

Lies ECP-Kommandos vom Filedeskriptor.
Diese Option benutzt der AVC-Controller, um der AMEngine mitzuteilen, über welchen Filedeskriptor sie mit dem AVC-Controller kommunizieren soll. Ist diese Option nicht angegeben, werden ECP-Kommandos von der Standardeingabe gelesen und die Rückgabewerte und Fehlermeldungen auf die Standardausgabe geschrieben.

`-dump`

Löse beim Auftreten eines Fehlers einen Programmabbruch mit core dump aus.
Um bei Tests eine Fehlerquelle leichter finden zu können, kann mit dem Debugger im core dump (Speicherabzug) des Adreßraums des Prozesses die Fehlerstelle lokalisiert werden.

7.3 Testumgebung

Die Tests an der AMEngine teilten sich in drei Bereiche:

- Tests der Audio-Schnittstelle AIO
- Standalone-Tests der AMEngine
- Integrationstests mit der AVC und MMC

7.3.1 Tests der Audio-Schnittstelle AIO

Die Tests der AIO-Library fanden schon während der Entwicklung des AMServers statt.

Als Testumgebung schrieb ich eine Reihe von Programmen, die einerseits über die AIO-Library die Funktionen der Audio-Hardware der verwendeten Rechner ansteuerten, andererseits die hardware-unabhängigen Funktionen von AIO testen.

Verwendung im MMC-Projekt fand das ACPanel, das eine Einstellung zahlreicher Parameter der Audio-Hardware erlaubt (siehe Abbildung).

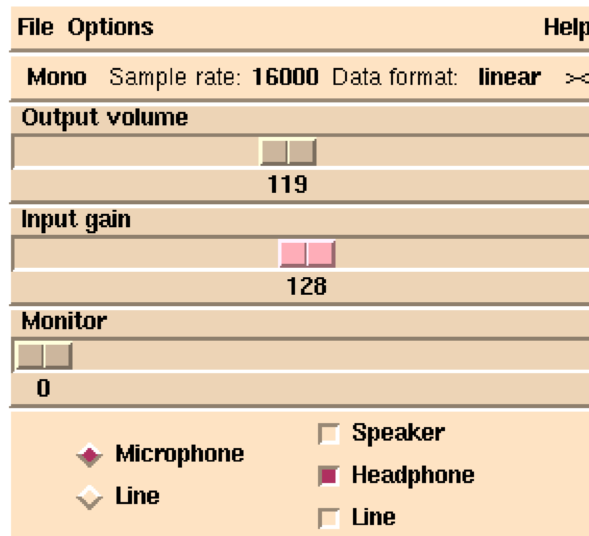

ABBILDUNG 8 ACPanel

7.3.2 Standalone-Tests der AMEngine

Die AMEngine war durch ihre ECP-Kommandoschnittstelle schon darauf vorbereitet, auch unabhängig von MMC arbeiten zu können. Die Tests wurden im allgemeinen durch Eingeben der ECP-Kommandos von Hand durchgeführt, teilweise mit Unterstützung durch beim Start der AMEngine eingelesene Tcl-Skripte.

Hier ist ein Beispiel für ein solches Skript. Es definiert eine Prozedur, die einen Source Endpoint und einen Sink Endpoint öffnet und Daten vom Source Endpoint an den Sink Endpoint schickt. Diese Prozedur wird initial einmal aufgerufen.

```
#!./amengine -init
# -*- tcl -*-

proc do_loop {} {
    set src_ep [opensource 16000 0 0 udp live]
    set sink_ep [opensink 16000 0 0 udp live]
    addsink $src_ep localhost [lindex $sink_ep 1]
    transmit $src_ep 1
```

```
    transmit [lindex $sink_ep 0] 1
    puts -nonewline [callbacks]
}

puts stderr „proc do_loop defined“

do_loop
```

Diese Prozedur benutzt einen großen Teil der Funktionen der AMEngine. Sie ist damit gut geeignet, um schnell die wesentliche Funktionalität des Programms zu überprüfen. Durch wiederholtes Aufrufen der Prozedur `do_loop` lassen sich Lasttests durchführen.

7.3.3 Integrationstests mit der AVC und MMC

Um das Zusammenwirken der einzelnen AVC-Subsysteme zu testen, benutzten wir das Programm `avctester`. Es wurde von Mitarbeitern der Firma Digital für Tests der AV-Komponente von MMC entwickelt und projektweit zur Verfügung gestellt.

Um bei Integrationstests mit der AVC und mit dem Gesamtsystem eine externe Steuerverbindung zur AMEngine aufzubauen, wurde das Programm `Socket`(1) [Nickelsen 1992] benutzt. Es ermöglicht die Aufnahme einer TCP-Verbindung zu einem spezifizierten TCP-Port des angegebenen Rechners. Standardein- und ausgabe des Programms werden mit der TCP-Verbindung gekoppelt. Durch Umlenken der Ein- und Ausgabe können in Gegensatz zu `Telnet`(1) auch nichtinteraktiv Kommandos an die AMEngine geschickt und ausgewertet werden.

7.4 Zusammenfassung

Beim Entwurf des Audio-Subsystems für MMC wurden die mit den ersten beiden Prototypen gemachten Erfahrungen berücksichtigt. Besonderer Wert wurde auf eine modulare Architektur mit klarer Trennung der Funktionalität sowie auf erweiterte Möglichkeiten zur Überwachung der Funktionsweise des Programms gelegt.

Im folgenden Kapitel schildere ich die beim Einsatz der AMEngine gemachten Erfahrungen.

KAPITEL 8 Erfahrungen und Ausblick

In diesem Kapitel schildere ich zunächst die Erfahrungen beim Einsatz der AMEngine, besonders unter Berücksichtigung der geforderten Eigenschaften, und die sich daraus ergebenden Ideen für die Weiterentwicklung.

8.1 Allgemeine Erfahrungen

Die Erfahrungen mit der AMEngine sind sehr gut. Sie erfüllt die im Kapitel „Anforderungen“ dargestellten Anforderungen an das Audio-Subsystem von MMC. Die AMEngine ist seit Anfang 1995 mit MMC im Einsatz und wurde mehrfach auf Messen und Präsentationen vorgeführt. Probleme des Audio-Subsystems wurden dabei nicht beobachtet.

8.2 Erfüllen der Basisfunktionalität

- Kommunikation mit dem AVC-Controller über das ECP
- Einrichten und Löschen von Source und Sink Endpoints
- Übertragung von Audiodaten über das AVXP
- Wiedergabe (Mischen) von mehreren Audio-Datenströmen auch verschiedener Kodierung
- Aussenden von mehreren Audio-Datenströmen auch verschiedener Kodierung
- Verwendung der Transportprotokolle UDP und TCP
- Schnittstelle für externe Quellen

- Erweiterung der ECP-Schnittstelle für mehrere Steuerverbindungen
- Flußkontrolle für gespeicherte Audiodaten
- Datenübertragung ohne AVXP
- Verbesserung des Verhaltens bei AVXP-Paketen beliebiger Größe

Die vom Audio-Subsystem der AVC geforderte Basisfunktionalität wird von der AMEngine in vollem Umfang erbracht. Beim Einsatz der AMEngine mit und ohne MMC ergaben sich keine Probleme. Die Schwierigkeit im Umgang mit AVXP-Paketen verschiedener Größe, die der AMServer noch aufgewiesen hatte, ist beseitigt.

8.3 Eigenschaften der AMEngine

8.3.1 Erweiterbarkeit um andere Transportprotokolle

Da bisher in MMC noch kein anderes Transportprotokoll als UDP eingesetzt wird, wurde die AMEngine auch noch nicht um weitere Transportprotokolle erweitert. Es liegen also zu diesem Punkt keine Erfahrungen vor. Die problemlose gleichzeitige Nutzung von UDP und TCP zeigt jedoch, daß zumindest Protokollimplementierungen mit einer Socket-ähnlichen Schnittstelle (wie zum Beispiel XTP oder die AAL-Implementierung von Fore Systems) ohne weiteres zu integrieren sind. Für andere Protokollschnittstellen müßte eine Abstraktion gefunden werden, die in etwa gleich der Socket-Schnittstelle benutzt werden kann.

8.3.2 Vermeidung von Aussetzern

Nutzung von Betriebssystemfunktionen

Im Kapitel „Entwurf der AMEngine“ bin ich auf die Möglichkeiten eingegangen, die das Betriebssystem der verwendeten Maschinen durch Erhöhung der Priorität und Verhindern des Auslagerns eines Prozesses bietet, um Aussetzer und Verzögerungen im Audio-Subsystem zu reduzieren. In Versuchen zeigte sich, daß diese Möglichkeiten effektiv sind, solange CPU- und Speicher-Engpässe auf der lokalen Maschine die Einhaltung der Echtzeitanforderungen erschweren.

Ursachen von Aussetzern

In Situationen, in denen die Tonqualität der MMC-Konferenz problematisch war, war die Maschinenbelastung jedoch meistens nicht die hauptsächliche Ursache. Problematischer wirkten sich Engpässe in der verwendeten Netzwerkverbindung aus. Bei Verwendung von Schmalbandnetzen (ISDN) tritt häufig die Situation auf, daß bei einer Erhöhung des Bandbreitenbedarfs, etwa durch gemeinsames Benutzen eines graphikintensiven Programms, die Audio-Übertragung zu Aussetzern neigt. Da hier auch die gesamte zur Verfügung stehende Bandbreite für Bursts bei der Übertragung von umfangreichen Grafikdaten nicht ausreicht, kann diese Situation auch durch die Verwendung einer „sparsamen" Audio-Kodierung nicht verbessert werden.

Hier fehlt bisher in den verwendeten Transportprotokollen (UDP/IP und TCP/IP) die Möglichkeit,Bandbreitenreservierung, um eine einwandfreie Übertragung der zeitkritischen Audiodaten zu gewährleisten. Das Netzwerkprotokoll IP kennt zwar ein Feld „Type of Service", mit dem grobe Abstufungen von Prioritäten angegeben werden können, aber ebenfalls keine Bandbreitenreservierung. [RFC 760]

Bandbreitenreservierung im Netzwerk

Das Reservierungsprotokoll RSVP wird mittlerweile schon mit mehreren kommerziellen IP-Implementierungen angeboten, kann aber noch nicht als gegeben vorausgesetzt werden. [RFC 1633, Metzler 1996] Mit MMC ist es noch nicht getestet worden.

Das kommende neue Internet-Protokoll, „IP New Generation", kennt Bandbreitenreservierung schon von sich aus. Es wird allerdings noch einige Zeit dauern, bis es flächendeckend zur Verfügung steht. [Metzler 1996]

Speziell unter dem Gesichtspunkt Multimedia entwickelte Protokolle wie XTP und ST-II kennen ebenfalls Bandbreitenreservierung, sind aber auch nicht überall verfügbar. [RFC 1190, Metzler 1996]

Bei der Nutzung von ATM-Verbindungen gibt es diese Möglichkeit zwar, wenn direkt das AAL-Interface benutzt wird, aber gerade hier treten wegen der typischerweise hohen zur Verfügung stehenden Bandbreite solche Probleme kaum auf. [Hoffmann 1996]

Neue Möglichkeiten

Ende 1995 wurde im MMC-Projekt ein Protokoll zur Signalisierung zwischen den AVCs der an einer Konferenz beteiligten Rechner spezifiziert. Damit ist es unter anderem für einen Sink Endpoint möglich, beim Source Endpoint eine Erhöhung oder Verringerung der Video-Framerate anzufordern.

Das bietet eine neue Möglichkeit zur Vermeidung von Audio-Aussetzern: Die Verlustrate bei den Audio-Paketen wird über einen bestimmten, nicht zu kurzen Zeitraum beobachtet. Ist sie zu hoch, wird die Framerate beim Videobild der eigenen Source Endpoints reduziert und den AVCs auf den anderen Rechnern signalisiert, sie mögen die Framerate der ausgesendeten Videoströme ebenfalls reduzieren, um damit Netzwerkbandbreite für eine bessere Audio-Übertragung freizugeben.

Dieses Konzept ist noch nicht implementiert; Erfahrungen damit liegen demnach noch nicht vor.

8.3.3 Beibehalten der Effizienz und des guten Delay-Verhaltens

Die guten Eigenschaften des AMServers wurden auch von der AMEngine erreicht. Wie im Kapitel „Plattformen und Werkzeuge“ bereits erwähnt, konnte das Delay-Verhalten durch einen von HP bereitgestellten Patch des Audio-Treibers im Betriebssystem deutlich verbessert werden. Die Zeit zwischen der Aufnahme des Tons auf der einen Maschine und der Wiedergabe auf der anderen liegt jetzt zwischen 100 und 150 Millisekunden.

8.3.4 Verbesserung der Stabilität

Die AMEngine kann ohne weiteres als stabil bezeichnet werden; Abstürze sind nicht bekannt. Das fatale Verhalten des AMServers, ohne ersichtlichen Grund CPU-Zeit zu verbrauchen, ist bei der AMEngine nicht aufgetaucht.

8.3.5 Verbesserung der Debugging- und Monitoring-Möglichkeiten

Wie im Kapitel „Entwurf der AMEngine“ beschrieben, wurden die Debugging- und Monitoring-Möglichkeiten der AMEngine verbessert. Die neuen Kommandos sind im Anhang „Endpoint Control Protocol“ ausführlich beschrieben. Sie erwiesen sich als sehr nütz-

lich, um Fehler im Zusammenwirken der MMC-Komponenten aufzuspüren.

Sie waren dagegen noch nicht ausreichend, um Probleme auf der Audioverbindung genauer bewerten zu können. Hier wären weitere Funktionen wünschenswert:

- Messung der Pakethäufigkeit auf einer Verbindung (Pakete pro Sekunde)
- Messung der Häufigkeit von fehlenden Paketen (Aussetzern)

8.3.6 Verbesserung der Wartbarkeit und Erweiterbarkeit sowie der Wiederverwendbarkeit einzelner Module.

Die Aufteilung der Funktionalität der AMEngine in Module hat sich gut bewährt. Das Programm ist wesentlich übersichtlicher und damit leichter wartbar als der AMServer.

Die Erweiterbarkeit kann dagegen noch verbessert werden. Beispiele:

- Die GSM-Kodierung von Audiodaten verlangt die Umwandlung von genau 160 Samples in einen GSM-Frame von 33 Bytes und umgekehrt. Das Schema zur Audiodatenkonvertierung der AMEngine geht jedoch davon aus, daß die Menge der Eingabedaten für eine Konvertierungsfunktion beliebig ist. Die Anpassung an GSM erfordert die Einführung einer zusätzlichen Ebene, in der die vom Netzwerk oder vom Audio-Device gelesene Datenmenge in die für den GSM-Coder/Decoder passenden Portionen vorgenommen wird. [Degener 1994; ETSI GSM 6.10]
- Als Ansatzpunkte für benutzerdefinierte Monitoringfunktionen lassen sich „Hooks“ denken, über die sich an bestimmte Ereignisse gebundene Funktionen definieren lassen.

  Ein mögliches Beispiel: Bei der Prozedur, die Pakete über das Netzwerk verschickt, läßt sich eine Tcl-Prozedur anmelden, die für jedes verschickte Paket aufgerufen wird und die Nummer des Source Endpoints und der Verbindung sowie die Paketgröße als Argumente übergeben bekommt. Die Tcl-Prozedur könnte folgendermaßen aussehen:

```
proc send_packet_hook {source connection size} {
   global packet_count
   global transmit_count

   incr packet_count($source,$connection)
   set transmit_count($source,$connection) \
```

```
   [expr $transmit_count($source,$connection) + size]
}
```

Mit dieser Prozedur werden im globalen Array packet_count für jede Verbindung jedes Source Endpoints die verschickten Pakete gezählt; in transmit_count werden entsprechend die verschickten Datenmengen aufsummiert. Das An- und Abmelden ließe sich ebenfalls als Tcl-Prozedur zur Verfügung stellen. Es könnte etwa so aussehen:

```
add_hook send_packet send_packet_hook
```

beziehungsweise

```
delete_hook send_packet send_packet_hook
```

8.3.7 Anwendungsmöglichkeiten außerhalb von MMC

Die AMEngine ist sowohl mit als auch ohne MMC als Schnittstelle zur Audio-Hardware für andere Applikationen geeignet. Nennenswerte Anwendungen, die diese Schnittstelle nutzen, gibt es allerdings bisher nicht. Im Anhang „Nutzung der AMEngine durch externe Anwendungen“ stelle ich in zwei Beispielen vor, wie externe Anwendungen die ECP-Schnittstelle der AMEngine nutzen können.

8.4 Weiterentwicklung der AMEngine

8.4.1 Spatial Mixing

Der erste Prototyp für das Audio-Subsystem, der auf der IFA 1993 vorgeführt wurde, verteilte zwei eingehende Audioströme auf die beiden Stereokanäle. Diese Idee läßt sich noch verfeinern: Wenn man davon ausgeht, daß die Videobilder der Konferenzpartner auf dem Bildschirm nebeneinander zu sehen sind, läßt sich das Tonsignal der Konferenzpartner entsprechend der Anordnung der Videobilder in den Stereoraum einblenden. Wir habe auch damit noch keine Erfahrung, erwarten aber eine deutliche Verbesserung der Hörempfindung.

8.4.2 Audioformate

Die AMEngine bietet bis jetzt eine zwar den MMC-Spezifikationen entsprechende, aber doch recht kleine Auswahl an Audioqualitäten. Geplant ist die Erweiterung um zunächst drei weitere Datentypen:

- IMA-ADPCM 22 kHz

  Als Audio-Datentyp bei noch geringem Bandbreitenbedarf (88,2 kBit/s), aber mit höherer Tonqualität als die bisher verwendete G.711-Kodierung wird in der kommenden Phase des MMC-Projekts die Kodierung IMA-ADPCM bei einer Abtastrate von 22,05 kHz, mono, eingesetzt werden. Eine frei verfügbare Implementierung steht zur Verfügung.
- GSM 6.10

  Die für den Einsatz in Mobiltelefonen entwickelte Kodierung GSM 6.10 benötigt bei einer Abtastrate von 8 kHz eine Bandbreite von 13 kBit/s bei guter Sprachverständlichkeit. Sie ist damit für den Einsatz auf ISDN-Leitungen sehr gut geeignet. Eine frei verfügbare Implementierung steht zur Verfügung [Degener, Bormann 1995].
- CD-Qualität

  Für den Einsatz in lokalen oder regionalen Breitbandnetzen bietet das bei Audio-CDs benutzte Format mit einer Kodierung von 16 Bit linear, einer Abtastrate von 44,1 kHz und zwei Kanälen eine excellente Tonqualität. Das CD-Datenformat wird von der Audio-Hardware direkt zur Verfügung gestellt.

TABELLE 6 In Zukunft unterstütze Audioformate

	Abtastrate	Kodierung	Wortbreite	Kanäle
GSM 6.10	8000 Hz	GSM 6.10	n/a	1
G.711	8000 Hz	µlaw	8 Bit	1
IMA-ADPCM	22050 Hz	ADPCM	4 Bit	1
CD	44100 Hz	linear	16 Bit	2

Um die verschiedenen Abtastraten 8 kHz, 16 kHz, 22,05 kHz und 44,1 kHz zu unterstützen, ist eine Abtastratenkonvertierung mit linearer Interpolation nicht mehr geeignet. Vom gleichzeitigen Anbieten verschiedener Abtastraten muß daher Abstand genommen werden; es ist nun nötig, die tatsächliche Abtastrate des AD/DA-Wandlers in der Audio-Hardware umzuschalten. Das kann natürlich nur dann passieren, wenn zum Moment des Umschaltens kein Endpoint mehr in Benutzung ist.

8.4.3 Gemeinsame Datenübertragung von Audio- und Videodaten

Die Integration von H.320-Endgeräten in MMC-Konferenzen benötigt Endpoints, die Audio- und Videodaten in einem Datenstrom übertragen. Für die MMC-Implementierung der TU, bei der Video- und Audio-Endpoints in getrennten Subsystemen verwaltet werden, erfordert das eine Zusammenführung der Audio- und Video-

Ströme. Ein Konzept dafür ist im Rahmen des MMC-Projekts noch zu erarbeiten.

8.4.4 Zusammenfassung

Die Implementierung erfüllt die gestellten Anforderungen an das Audio-Subsystem des Video-Konferenzsystems MMC sehr gut. Die Wartbarkeit und Erweiterbarkeit wurde gegenüber dem Prototyp AMServer deutlich verbessert. Die zukünftige Entwicklung von MMC erfordert weitere Arbeit an der AMEngine.

KAPITEL 9 Literaturverzeichnis

Altenhofen et al. 1993 — Michael Altenhofen, Jürgen Dittrich, Rainer Hammerschmidt, Gabriela Gahse, Sigrid Gleser, Dietmar Hehmann, Ralf Guido Herrtwich, Karl Jonas, Thomas Käppner, Carsten Kruschel, Angar Kückes, Walter Reinhard, Thomas Steinig, Oliver Wenzel, Kathrin Werner, Jörg Winckler: *The BERKOM-II Multimedia Collaboration Service (Release 2.0)*. Internal BERKOM working document, Berlin 1993.

Comer 1988 — Douglas E. Comer: *Internetworking with TCP/IP – Principles, Protocols, and Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

Degener 1994 — Jutta Degener: *Digital Speech Compression*. In *Dr. Dobb's Journal*, Ausgabe 224, Dezember 1994. Miller Freeman, San Mateo, CA, 1994.

Degener 1996 — Jutta Degener: *GSM 06.10 digital speech compression*. WWW-Seite `http://www.cs.tu-berlin.de/~jutta/toast.html`, TU Berlin 1996.

Degener, Bormann 1995 — Jutta Degener, Carsten Bormann: *GSM 06.10 1.0, Patchlevel 7*. Software-Distribution, TU Berlin 1995.

Dermler, Froitzheim 1992 — Gabriel Dermler, Konrad Froitzheim: *JVTOS – A Reference Model for a New Multimedia Service*. Proceedings, *4th IFIP Conference on High Performance Networking (hpn `92)*. Edited by A. Danthine, O. Spaniol. Liege, 1992.

Dermler et al. 1993 — Gabriel Dermler, Thomas Gutekunst, Bernhard Plattner, Edgar Ostrowski, Frank Ruge, Michael Weber: *Constructing a Distributed Multimedia Joint Viewing and Tele-Operation Service for Heterogeneous*

Workstation Environments. In *Fourth IEEE Workshop on Future Trends of Distributed Com puting Systems*. Lisboa, 1993.

Dermler et al. 1994 — Gabriel Dermler, Thomas Gutekunst, Edgar Ostrowski, Frank Ruge: *Sharing Audio/Video Applications Among Heterogeneous Platforms*. In *Fifth IEEE COMSOC International Workshop on Multimedia Communications*. Kyoto, 1994.

ETSI GSM 6.10 — European Telecommunications Standards Institute: *European digital cellular telecommunications system (Phase 1); Full-rate speech transcoding (GSM 06.10)*. 1992.

Gamma et al. 1995 — Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

Gehring 1996 — Andreas Gehring: Persönliches Gespräch, 1996.

Gleser, Kückes, Nickelsen 1994 — Sigrid Gleser, Ansgar Kückes, Jürgen Nickelsen: *BERKOM Multimedia Teleservices Multimedia Collaboration – User's Guide & Installation Guide*. Version 2.0, TU Berlin, 1994.

HP 1992 — Hewlett-Packard: *HP-UX Reference – HP 9000 Computers, HP-UX Release 9.0*. Third Edition, Volumes 1 - 3. Hewlett-Packard Company, Palo Alto 1992.

Hoffmann 1996 — Gero Hoffmann: Persönliches Gespräch, 1996.

ISO 1990 — ISO/IEC: *Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]*. ISO/IEC Standard 9945-1:1990.

ITU-T G.711 — ITU-T: *Pulse code modulation (PCM) of voice frequencies*. ITU-T Recommendation G.711, 1990.

ITU-T H.320 — ITU-T: *Line Transmission of Non-Telephone Signals – Narrow-Band Visual Telephone Systems and Terminal Equipment*. ITU-T Recommendation H.320, 1993.

KKawalek 1994 — Jürgen Kawalek: *A User Oriented Approach to QoS*. In R. Roth, J. De Meer, D. Cochran (Hrsg.): *Proceedings of the Workshop on „Quality of Service and Network Performance" September 8th 1994 alighned with the „2nd International Conference on Intelligence in Broadband Services and Networki (IS&N 94)" on September 7 to 9, 1994*. Aachen 1994.

Kawalek 1995 — Jürgen Kawalek: *A User Perspective for QoS Management*. Vorgestellt beim *QoS Workshop aligned with the 3rd International Conference on Intelligence in Broadband Services and Network (IS&N 95)*, 16. September 1995, Kreta, Griechenland.

Kernighan, Ritchie 1990 — Brian W. Kernighan, Dennis M. Ritchie: *The C Programming Language – Second Edition, ANSI C.* Deutsche Übersetzung: *Programmieren in C*. Hanser, München 1990.

Kileng 1993 — Frode Kileng: Persönliche Kommunikation (E-Mail), 1993.

Lampen, Mahler, Nickelsen 1993 — Andreas Lampen, Axel Mahler, Jürgen Nickelsen: *ShapeTools – User's Handbook*. In *STONE System Documentation*, TU Berlin 1993.

Leffler et al. 1989 — Samuel J. Leffler, Michael J. Karels, Marshall Kirk McKusick, John S. Quarterman: *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA, 1989.

Lewine 1994 — Donald Lewine: *POSIX Programmers Guide – Writing Portable UNIX Programs with the POSIX.1 Standard*. O'Reilly, Sebastopol, CA, 1994.

Malm 1994 — Pål S. Malm: *The unOfficial Yellow Pages of CSCW*. WWW-Dokument `http://www.tft.tele.no/cscw/`, Tromsø, Norwegen, 1994.

Metzler 1996 — Bernard Metzler: Persönliches Gespräch, 1996.

Nickelsen 1992 — Jürgen Nickelsen: *Socket-1.1*. Software-Distribution, Usenet-Newsgroup `comp.sources.unix`, Volume 26, Issue 75, 1992.

Nye, O'Reilly 1992 — Adrian Nye, Tim O'Reilly: X *Toolkit Intrinsics Programming Manual – OSF/Motif 1.1 Edition, for X11, Release 5*. O'Reilly, Sebastopol, CA, 1992.

Ousterhout 1990 — John Ousterhout: *Tcl: An Embeddable Command Language*. Usenix Conference Proceedings, Winter 1990.

Ousterhout 1994 — John Ousterhout: *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.

POSIX.1 — IEEE: Portable Operating System Interface for Computing Environments, IEEE Std. 1003.1-1990.

RFC 760 — Jonathan Postel: DoD Standard Internet Protocol. RFC 760, Information Sciences Institute, 1980.

RFC 821 — Jonathan Postel: *Simple Mail Transfer Protocol*. RFC 822, Information Sciences Institute, 1982.

RFC 959 — Jonathan Postel, J. Reynolds: *File Transfer Protocol Specification*. RFC 765, Information Sciences Institute, 1985.

RFC 977 — Brian Kantor, Phil Lapsley: *Network News Transfer Protocol – A Proposed Standard for the Stream-Based Transmission of News*. RFC 977, University of California at San Diego & University of California at Berkeley, 1986.

RFC 1190 — C. Topolcic (Hrsg.): *Experimental Internet Stream Protocol, Version 2 (ST-II)*. RFC 1190, CIP Working Group, 1990.

RFC 1633 — Bob Braden, D. Clark, S. Shenker: *Integrated Services in the Internet Architecture: an Overview.* RFC 1633, Information Sciences Institute, Massachussets Institute of Technology, Xerox PARC, 1994.

Rosen, Howell 1991 — Stuart Rosen, Peter Howell: *Signals and Systems for Speech and Hearing*. Academic Press, London 1991.

Sage, Palmer 1990 — Andrew P. Sage, James D. Palmer: *Software Systems Engineering*. Wiley & Sons, 1990.

Stevens 1990 — W. Richard Stevens: *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.

Stevens 1992 — W. Richard Stevens: *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA, 1992.

Sweet 85 — Richard E. Sweet: *The Mesa Programming Environment*. In SIGPLAN Notices, 20(7), Juli 1985.

Tanenbaum 1989 — Andrew S. Tanenbaum: *Computer Networks. Second Edition*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

Wallmüller 1990 — Ernest Wallmüller: *Software-Qualitätssicherung in der Praxis*. Hanser, München, Wien, 1990.

ANHANG A

Reference Manual Page

AMENGINE(1) **AMENGINE(1)**

NAME

AMEngine — Audio Multiplexer Engine

SYNOPSIS

amengine [*options*]

DESCRIPTION

The AMEngine is the audio subsystem of the Multimedia Collaboration videoconferencing system (MMC) for HP workstations. Although the AMEngine is designed primarily for use in the MMC environment, it can be used independently as an interface to the audio hardware of HP workstations.

The AMEngine can establish audio stream connections with other processes on the same or on a different computer over the network. Audio data arriving on incoming streams is mixed with audio data from other incoming streams (if present) and played on the local audio hardware; audio data recorded on the local audio hardware is sent to outgoing audio streams.

When invoked, the AMEngine listens for audio stream control commands on a bidirectional communication channel, called the control connection. The control connection may be the standard input / standard output or a file descriptor provided by the parent process (see the description of options below). The protocol on the control connection is a simple text-based protocol called Endpoint Control Protocol (ECP, see below for a description). When the AMEngine reads a control command, the requested operation is performed and a result message (which may be

empty) is sent to the controlling entity. In case of an error condition, an appropriate error message is written to the control connection.

The AMEngine terminates when a "`quit`" or "`exit`" command is read on a control connection or when the last control connection is closed by the controlling process.

External Access

Other processes can open a control connection to a running AMEngine by opening a TCP connection to the port 64927. All control connections are created equal; each control connection has complete control over the AMEngine.

In order to access an already running AMEngine, a process has to provide an authorization key. The key consists of 128 printable characters and a terminating newline. It is stored by the AMEngine in the user's home directory in the file $HOME/.ame-key.hostname, where hostname is replaced by the name of the host the AMEngine is running on. This file is made readable only by the user running the AMEngine. To gain access, the key has to be written to the control connection after it has been opened. See below for an example.

If the AMEngine fails in writing the authorization key to the file or in setting the proper access permission for the file, additional control connections are not permitted.

Endpoints

Each audio stream consists of a data source and a data sink. Sources and sinks are called endpoints; there are source endpoints and sink endpoints, respectively.

Source Endpoint
A source endpoint acts as a source of audio data. It reads the audio data from the workstation's audio hardware, processes it for transmission through the network, and sends it to the connected sinks. A source endpoint can serve multiple connected sink endpoints.

Sink Endpoint
A sink endpoint acts as a sink for audio data. It reads packets of audio data from a network connection and processes the received audio data for playback on the workstation's audio hardware. A sink endpoint can serve only one connected source endpoint. When the sink endpoint is opened, a server socket of the specified connection type is opened and the port number of this socket is returned with the endpoint ID. The associated source can then connect to this port and send audio packets to the sink through this connection.

Each endpoint has a unique ID, which is a small number, and certain characteristics. These characteristics include characteristics of the connection as well as characteristics of the audio data transmitted or received.

Characteristics of the audio stream connection:

Transport Protocol
Audio streams are (currently) be based on the UDP or TCP transport protocol. (Use of other transport protocols is planned.)

Application Protocol
Audio streams can use MMC's Audiovisual Exchange Protocol (AVXP) or consist of raw audio data. With the AVXP, each portion of audio data is prefixed with a header describing the data. (See below for a description of AVXP headers.)

Flow Control
Incoming audio streams using the TCP transport protocol can use flow control to block the process writing to the stream (the source) when the AMEngine is busy processing the data.

Characteristics of the audio data:

Encoding
The AMEngine currently supports two different encoding formats: 16-bit linear and 8-bit ulaw.

Sample Rate
Currently 8 kHz and 16 kHz sample rates are supported.

Mono / Stereo
Audio data can consist of one channel (mono) or two interleaved channels (stereo).

Activity Type

Each endpoint is associated an activity type. The activity type "`live`" means that incoming audio data arrives synchronously as it is recorded. Endpoints with this activity type do very little buffering of incoming audio data in order to keep the delay between recording and playback low. If more AVXP packets arrive than the AMEngine can write to the audio hardware, older packets are dropped.

With the activity type "`application`" a sink endpoint uses the flow control capabilities of the underlying transport protocol to prevent buffer overrun. This is only available with a transport protocol that supports flow control. Currently this is only TCP.

Connection Types

The current implementation of the AMEngine supports the TCP and UDP transport protocols. With UDP, flow control is not available with the activity type "`application`".

OPTIONS

–version
Write extended version information to standard error and quit the program.

–init *initfile*

Read ECP commands from the specified file on startup.

–control *controlfd*

Use the open file descriptor controlfd for the ECP connection. This file descriptor has to be provided open read/write by the parent process. If this option is not given, ECP requests are read from the standard input; ECP return values and error messages are written to the standard output.

–acpanel

Open the MMC Audio Control Panel on startup. In order to use this option, the acpanel executable must be available in $PATH.

–debug *ignored_argument*

This option has no effect and is provided only for backward compatibility.

–dump

Dump core on a fatal error by calling abort(3). This option has an effect only if the AMEgine has been compiled with the `–DDEBUG` option.

–help

Write a short summary of the command line options to standard error on startup.

ENDPOINT CONTROL PROTOCOL

The protocol to control the AMEngine is text-based. It consists mainly of five types of messages:

- Commands from the controlling instance to the AMEngine. Commands consist of several words on a single line separated by blanks or TAB characters; the maximum line length is 256 characters including the newline character.
- Return values sent by the AMEngine to the controlling instance after successful processing of a command. A return value can span several lines. Each line is prefixed with the token“`RET `”; the last line of a command consist of the “`RET `” token only.
- Error codes sent by the AMEngine to the controlling instance after a failure processing a command. An error code consists of the token “`ERR `” followed by a numeric error code and a textual error description.
- Asynchronous error notifications sent by the AMEngine. These usually occur while processing audio data.
- Event notifications sent by the AMEngine to the controlling instance. An event notification consists of the token “`EVT `”, the event type and additional parameters depending on the event type.

In addition to these message types the AMEngine returns on startup either a return value containing version information or an error code if the startup of the AMEngine has failed.

ECP Commands

`listav` [*activity_type*]
List the currently available audio data types with sample rate, coding, and stereo flag, one per line. The following combinations of audio parameters are currently supported:

Sample rate	Coding	Stereo
8000	0	0
16000	0	0
16000	2	1

(The coding value is 0 for ulaw coding, 2 for 16 bit linear coding as defined in <sys/audio.h>.)

The set of audio data is likely to be expanded in the future to achieve higher quality audio transmission as well as the capability to use MMC on connections with limited bandwidth.

`opensink` *sample_rate coding stereoflag connection_type activity_type* [*options*]
Open a sink endpoint with the specified sample rate, audio coding, stereo or mono, for connections of the specified type and the specified type of activity. Return the numeric endpoint ID and the port number of the sink endpoint. The activity types are "`live`" (no flow control) and "`appl`" for application (with flow control). The only option currently supported is "`noheaders`" for audio streams with raw audio data. If this option is not given, the endpoint expects to receive audio packets with AVXP headers.

`opensource` *sample_rate coding stereoflag connection_type activity_type* [*options*]
Open a source endpoint with the specified sample rate, audio coding, stereo or mono, for connections of the specified type and the specified type of activity. Return the numeric endpoint ID of the source endpoint. The activity types are "`live`" (no flow control) and "`appl`" for application (with flow control). The only option currently supported is "`noheaders`" for audio streams with raw audio data. If this option is not given, the endpoint transmits audio packets with AVXP headers.

`addsink` *endpoint host port*
Add a sink to the specified source endpoint. The sink can be reached at the specified port number on the specified host.

`transmit` *endpoint* `0 | 1`
Switch transmission state of the specified endpoint off or on. When the transmission state is off, a source endpoint does not send AVXP packets; a sink endpoint does not write the received data to the audio hardware.

`remsink` *endpoint host port*
Remove the sink located at port on host from the specified source endpoint.

`close_ep` *endpoint*
: Close the specified endpoint.

`closeall`
: Close all endpoints.

`exit`
: Terminate the AMEngine gracefully.

`quit` [*exitcode*]
: Terminate the AMEngine immediately, exit the program with exitcode (0 if not specified).

`help`
: Return a summary of the ECP commands supported by the AMEngine.

Several other ECP commands are not documented here; they exist for debugging purposes only and are constantly subject to change.

Tcl Interpreter

The ECP commands are interpreted by an interpreter of the Tool Command Language (Tcl). In addition to the ECP commands, arbitrary Tcl commands can be issued on a control connection. ECP commands can be used in Tcl programs as well.

Error codes

Each ECP error code consists of a numeric code and a message describing the cause of the error. The message is meant to be understood by humans, but not necessarily by the end user of the MMC system.

On fatal errors, the AMEngine process is terminated and the error code is returned as the exit code to the parent process.

1 The socket(2) system call failed.

2 The bind(2) system call failed.

3 A file descriptor returned by a system call was larger than the system's FD_SETSIZE. (This is extremely unlikely.)

4 The listen(2) system call failed.

5 The select(2) system call failed.

6 The write(2) system call failed.

7 Not used.

8 The read(2) system call failed.

9 The read(2) or writev(2) system call returned an unexpected result.

10 The accept(2) system call failed.

11 The setsockopt(2) system call failed.

12 The read(2) system call failed when reading from the network.

13 Not used.

14 Not used.

15 An integer argument was expected but not found.

16 An unexpected internal error occured. This can't happen, of course.

17 A usage error on the command line occured.

18 The specified host is unknown.

19 The connect(2) system call failed. This is usually caused by trying to connect to an unreachable host.

20 The given ECP command is unknown.

21 The parameters to the given ECP command don't have the expected type.

22 The specified audio data type is not supported.

23 The number of available endpoints is exhausted.

24 The number of available connections is exhausted.

25 The specified connection type is not supported.

26 An error has occured during the initialization of the audio hardware.

27 An end-of-file condition has occured on the ECP connection.

28 Not used.

29 The specified endpoint ID is invalid.

30 The specified remote endpoint ID is invalid.

31 Not used.

32 Not used.

33 The gettimeofday(2) system call failed.

34 The host's own IP address could not be determined.

35 The specified sink is already connected to this endpoint.

36 The specified activity type is not supported.

37 Not used.

38 – 46 An internal error occured.

47 The initialization of the Tcl interpreter failed.

48 The fdopen(3) call failed.

49 The Tcl interpreter found an error in the Tcl code.

50 The specified endpoint option is invalid.

51 Setting real-time priority for the AMEngine failed. (This is not serious.)

52 Locking the AMEngine process in memory failed. (This is not serious.)

53 Resetting the group ID failed.

54 The no longer supported **–debug** option has been given on the command line. (This is not serious.)

55 The authorization key file could not be accessed.

56 The authorization key could not be read from the control connection.

AUDIOVISUAL EXCHANGE PROTOCOL

Each packet on an AVXP connection consists of a header and a payload. The header can be defined as a C struct as follows:

```
typedef struct _avxp_header {
   unsigned long ah_ipaddr ;     /* 4 bytes */
   unsigned short ah_epid ;      /* 2 bytes */
   unsigned long ah_seqnum ;     /* 4 bytes */
   unsigned char ah_chunk ;      /* 1 byte  */
   unsigned char ah_chunktot ;   /* 1 byte  */
   unsigned long ah_offset ;     /* 4 bytes */
   unsigned long ah_time ;       /* 4 bytes */
   unsigned int ah_eos:1 ;       /* 1 bit   */
   unsigned int ah_unused1:6 ;   /* 6 bits  */
   unsigned int ah_pause:1 ;     /* 1 bit   */
   unsigned char ah_unused2 ;    /* 1 byte  */
   unsigned long ah_payload ;    /* 4 bytes */
} avxp_header_t ;
```

All values are in network byte order, and we assume that the compiler does not insert padding bytes between the struct's fields. The meanings of the fields:

`ah_ipaddr`
: The sender's IP address.

`ah_epid`
: ID of the sender's source endpoint.

`ah_seqnum`
: Sequence number of packet.

`ah_chunk`
: Chunk number of packet (not used for audio data).

`ah_chunktot`
: Number of chunks in current frame (not used for audio data).

`ah_offset`
: Offset of chunk in frame; total size of frame if packet is the first chunk (not used for audio data).

`ah_time`
: Timestamp of packet.

`ah_eos`
: “End Of Stream”; indicates that the connection is to be closed.

`ah_unused1`
: (unused)

`ah_pause`
: To be set when no data should be expected for some time.

`ah_unused2`
: (unused)

`ah_payload`
: Length of the payload part of the packet.

The AMEngine does neither set nor recognize the pause and EOS flags. The AMEngine sends packets with a payload of 512 bytes.

REAL-TIME ISSUES

For the playback and recording of audio data it is particularly important to do the processing in real time in order to avoid dropouts and increased delays. Since HP-UX is, like most other UNIX-like operating systems, not really suited for real-time processing, it cannot guarantee that a process will be able to operate in real time.

It has, however, provisions to increase the chance of a process to meet his real-time constraints: Setting the process’s priority to a so-called real-time priority using the system call rtprio(2) and locking the process’s address space in real memory with plock(2). A process with real-time priority will not be interrupted by a process with a time-sharing priority or a process with a lower real-time priority. A process address space which is locked in real memory (as opposed to virtual memory) will neither be swapped out nor have parts of it paged to disk.

These system calls require special privileges. The AMEngine can be given these privileges by creating a special UNIX user group for real-time applications, granting the group the privileges RTPRIO and MLOCK using setprivgrp(1), and then running the AMEngine under this group ID.

The AMEngine is prepared for running as a set-group-ID application; it resets its group ID immediately after the calls to rtprio(2) and plock(2) such that the special group privileges cannot be exploited by the user.

If the user ID the AMEngine is running with does not belong to a group having the privileges to call rtprio(2) and plock(2), the calls simply fail and an appropriate message is issued. This is not considered a serious error, but the AMEngine's performance may be less than optimal if the machine is heavily loaded.

EXAMPLES

The first example shows a communication with the AMEngine on the ECP connection during an MMC conference. After querying the AMEngine for the available audio data types, a source and a sink for G.711 data are opened; a sink on the remote host midlum is added to the source endpoint. The first attempt to add the sinks fails because the specified endpoint ID is not the ID of a source endpoint. Transmission is then switched on. At the end of the conference all endpoints are closed and the AMEngine process is terminated.

Setting real-time priority and locking the process in memory fails because the process doesn't have sufficient privileges.

```
AME: setting real-time priority failed (not serious). Syscall
error 1: Not owner
AME: lock process in memory failed (not serious). Syscall error 1:
Not owner
RET AMEngine ame-1.0 (Thu Jan 12 20:07:23 1995 by nickel@prz.tu-
berlin.de) running on 130.149.226.49
RET
listav live
RET 16000 0 0
RET 16000 2 1
RET  8000 0 0
RET
opensource 8000 0 0 udp live
RET 0
RET
opensink 8000 0 0 udp live
RET 1 23100
RET
addsink 1 midlum 23100
ERR 29 Command error: invalid endpoint ID.
addsink 0 midlum 23100
RET
transmit 0 1
RET
transmit 1 1
RET
closeall
```

```
RET
exit
RET
```

The second example shows a typical application szenario. The application process has connected to the AMEngine via TCP port 64927 After providing the authorization key read from `$HOME/.ame-key.`*hostname*, it issues commands to open a TCP audio stream for 16 kHz linear stereo raw audio data with flow control. After that, the control connection is closed.

```
RET AMEngine ame-1.0 (Thu Jan 12 20:07:23 1995 by nickel@prz.tu-
berlin.de) running on 130.149.226.49
RET Additional control on fd 10
RET
68298a162c03342eb90da115252cb13638ecf54c3c69aed23b6182eac453072226
3126ff23b1a9565e08cccddd1d8f8f576c5165352f76ec7516a205b2906c90
RET
opensink 16000 2 1 tcp appl noheaders
RET 0 23100
RET
transmit 0 1
RET
close
RET
```

FILES

`$HOME/.ame-key.`*hostname*

File containing the authorization key.

SEE ALSO

setprivgrp(1), accept(2), bind(2), connect(2), gettimeofday(2), listen(2), plock(2), read(2), rtprio(2), select(2), setsockopt(2), socket(2), write(2), writev(2), abort(3), fdopen(3), group(4)
MMC User's Guide

WARNINGS

Security Concerns

Since the Tcl interpreter is capable to execute commands far beyond the intended scope of the AMEngine, it is not a wise idea to give the AMEngine's authorization key to an untrusted process.

Limitations

The AMEngine can receive AVXP packets with a size of up to 4096 bytes; it will behave unexpectedly if larger packets are received.

After sending a command, the process controlling the AMEngine must wait for the return value or error code before sending another command. The AMEngine may behave unexpectedly otherwise.

The size of a control message must not exceed 2048 bytes.

Bugs

The set of error return values does not have a reasonable order or structure.

The current access control scheme is not really satisfactory.

Surely more. Please report other bugs to the author or to <mmcs@prz.tu-berlin.de>.

VERSION

This manual page describes AMEngine ame-1.0.

AUTHOR

The AMEngine and this manual page have been written by Juergen Nickelsen <nickel@prz.tu-berlin.de>.

The AMEngine contains an interpreter for the Tcl Tool Command Language. The Tcl interpreter is copyright © 1987-1993 by The Regents of the University of California and is used by permission.

The AMEngine contains code for coding and decoding Intel/DVI ADPCM audio data. This code is copyright © 1992 by Stichting Mathematisch Centrum, Amsterdam, The Netherlands, and is used by permission.

The AMEngine contains code for conversion of µlaw audio data that has been donated to the public domain by Sun Microsystems, Inc.

ANHANG B

Nutzung der AMEngine durch externe Anwendungen

In diesem Kapitel stelle ich zwei Programme vor, die als externe Anwendungen die ECP-Schnittstelle der AMEngine als Zugang zur Audio-Hardware des Rechners benutzen.

ame_play

Um zu demonstrieren, daß die ECP-Schnittstelle der AMEngine einfach anzusprechen ist, implementierte ich einen Zugriff darauf als Bourneshell-Skript (siehe `sh`(1)) unter Verwendung von `Socket`(1) [Nickelsen 1992]. Dabei habe ich der Einfachheit halber auf eine Fehlerbehandlung verzichtet.

Das Programm `Socket`(1), das die Verbindungen zur AMEngine herstellt, wird hier dreimal aufgerufen:

1. Eine Verbindung zur Steuerschnittstelle der AMEngine öffnet einen Sink Endpoint und liest Endpoint ID und Portnummer aus dem Rückgabewert.
2. Mit der nun bekannten Endpoint ID wird der Transmission Status des Endpoint auf „on“ gesetzt.
3. Die Nutzdaten werden zur AMEngine übertragen. Die Wiedergabe kann dabei auf einem beliebigen Rechner erfolgen, solange der Autorisierungsschlüssel in $HOME/.ame-key.<hostname> liegt.

```
#!/bin/sh -e

host=$1
srate=$2
```

```
coding=$3
stereo=${4?‘usage: ame_play host srate coding stereo [file]‘}
file=${5}

epport=`(cat $HOME/.ame-key.$host
 echo opensink $srate $coding $stereo tcp appl noheaders ; sleep 1 ;
     echo close ) |
    socket $host 130463 |
    (read line ; read line ; read line ; read line
     read ret ep port ; read ret
 echo $ep $port)`

 echo $epport

ep=`echo $epport | sed ,s/\([^ ]*\) .*/\1/‘`
port=`echo $epport | sed ,s/.* \([^ ]*\)/\1/‘`

echo $ep $port

(cat $HOME/.ame-key.$host
 echo transmit $ep 1 ; sleep 1 ; echo close) | socket $host 130463 > /dev/null
cat $file | socket -q $host $port
```

ame-example

Das zweite Beispiel ist als Beispielprogramm für die Verwendung der AMEngine von einer nicht-MMC-Applikation aus entworfen. Es ist zwar unflexibel, sollte aber in Zusammenhang mit der Reference Manual Page hinreichend übersichtlich die Benutzung der AMEngine zeigen.

```
/*
 * $Header: ame_example.c[3.0] Tue May 16 19:20:49 1995 nickel@prz.tu-
berlin.de proposed $
 * Example program for connecting to the AMEngine and playing a
 * file. */

#ifndef _HPUX_SOURCE
#define _HPUX_SOURCE
#endif /* _HPUX_SOURCE */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/param.h>
#include <netinet/in.h>
#include <netdb.h>
```

```
#include <stdlib.h>

#define AME_PORT   130463       /* Port of the AMEngine. */

#define ERR_OPENSINK    1       /* Error opening sink. */
#define ERR_OPENFILE    2       /* Error opening file. */
#define ERR_USAGE       3       /* Usage error. */
#define ERR_PLAY        4       /* Error playing file. */

int create_client_socket(char *hostname, int port) ;
int open_sink(void) ;           /* Return port. */
int play_file(int port, int fd) ;


int main(int argc, char *argv[])
{
    int port ;                  /* Sink port to connect to. */
    int fd ;                    /* Descriptor for file to play. */

    /* Open sink on AMEngine. */
    if ((port = open_sink()) == -1) {
        fputs(„Open sink on AMEngine failed\n‟, stderr) ;
        exit(ERR_OPENSINK) ;
    }

    /* Determine what to play: file or stdin. */
    switch (argc) {
      case 1:
        fd = STDIN_FILENO ;
        break ;
      case 2:
        if ((fd = open(argv[1], O_RDONLY)) == -1) {
            perror(argv[1]) ;
            exit(ERR_OPENFILE) ;
        }
        break ;
      default:
        fputs(„usage: ame_example [file]\n‟, stderr) ;
        exit(ERR_USAGE) ;
    }

    /* Play. */
    if (play_file(port, fd) == -1) {
        fputs(„Playing file on AMEngine failed\n‟, stderr) ;
        exit(ERR_PLAY) ;
    }
    close(fd) ;

    exit(0) ;
}
```

```
/* Open a sink on the local host‘s AMEngine. The sink accepts a TCP
 * connection without AVXP headers and will do flow control. The
 * format of the audio data is 8 kHz, ulaw mono (G711). */
int open_sink()
{
    int ctl_fd ;                /* File descriptor for control
                                 * connection. */
    FILE *ctl_w ;               /* FILE *for writing to control
                                 * connection. */
    FILE *ctl_r ;               /* File * for reading from control
                                 * connection. */
    char ret_msg[256] ;         /* Buffer for return message. */
    int port ;                  /* Port number. */
    int endpoint ;              /* Endpoint ID. */
    char auth_key[130] ;        /* Authorization key. */
    int key_fd ;                /* File descriptor for authorization key. */
    char key_file[MAXPATHLEN+1];/* Name of authorization key file. */
    char hostname[MAXHOSTNAMELEN+1]; /* Name of current host. */
    int n_bytes ;               /* No. of bytes returned by read(2). */

    /* Read authorization key from $HOME/.ame-key.<hostname> */
    gethostname(hostname, MAXHOSTNAMELEN) ;
    sprintf(key_file, „%s/.ame-key.%s“, getenv(„HOME“), hostname) ;
    if ((key_fd = open(key_file, O_RDONLY, 0)) == -1) {
        perror(key_file) ;
        return -1 ;
    }
    if ((n_bytes = read(key_fd, auth_key, 129)) == -1) {
        perror(„read auth key“) ;
        return -1 ;
    }
    if (n_bytes != 129) {
        fprintf(stderr, „auth key: short read (%d)\n“, n_bytes) ;
        return -1 ;
    }
    close(key_fd) ;
    auth_key[129] = 0 ;         /* For easier use with fputs(3). */

    /* Open socket connection to AME. */
    if ((ctl_fd = create_client_socket(„localhost“, AME_PORT)) == -1) {
        perror(„create socket“) ;
        return -1 ;
    }

    /* Opening stdio streams for the connection to the AMEngine makes
     * handling of newline on read and string formatting a little
     * easier. I encountered wierd problems when I used the same
     * stream for reading *and* writng, so I use two. The stream for
     * writing must be line buffered only. */
```

```
if ((ctl_r = fdopen(ctl_fd, „r“)) == 0) {
    perror(„fdopen“) ;
    return -1 ;
}
if ((ctl_w = fdopen(ctl_fd, „w“)) == 0) {
    perror(„fdopen“) ;
    return -1 ;
}
if (setvbuf(ctl_w, 0, _IOLBF, BUFSIZ) != 0) {
    perror(„setvbuf“) ;
    return -1 ;
}

/* Read and ignore initial greeting message. */
do {
    if (fgets(ret_msg, 255, ctl_r) == 0) {
        fputs(„Premature EOF on control connection\n“, stderr) ;
        return -1 ;
    }
} while (strlen(ret_msg) > 5) ;

/* Send authorization key. */
fputs(auth_key, ctl_w) ;
if (fgets(ret_msg, 255, ctl_r) == 0) {
    fputs(„Access to control connection denied\n“, stderr) ;
    return -1 ;
}

/* Send opensink command, expect message with endpoint ID and port
 * number. */
fprintf(ctl_w, „opensink 8000 0 0 tcp appl noheaders\n“) ;
if (fgets(ret_msg, 255, ctl_r) == 0) {
    fputs(„Premature EOF on control connection\n“, stderr) ;
    return -1 ;
}
if (!strncmp(ret_msg, „ERR“, 3)) {
    fprintf(stderr, „Error message from AMEngine: %s“, ret_msg) ;
    return -1 ;
}
if (sscanf(ret_msg, „RET %d %d“, &endpoint, &port) != 2) {
    fprintf(stderr, „Return message from AMEngine hosed: %s“, ret_msg) ;
    return -1 ;
}

/* Read final „RET“ line. */
if (fgets(ret_msg, 255, ctl_r) == 0) {
    fputs(„Premature EOF on control connection\n“, stderr) ;
    return -1 ;
}

/* Send transmit command with endpoint ID. */
```

```
    fprintf(ctl_w, „transmit %d 1\n“, endpoint) ;
    /* Read „RET“ line. */
    if (fgets(ret_msg, 255, ctl_r) == 0) {
        fputs(„Premature EOF on control connection\n“, stderr) ;
        return -1 ;
    }

    /* Send close command and close connection. */
    fputs(„close“, ctl_w) ;
    fclose(ctl_w) ;
    fclose(ctl_r) ;

    return port ;
}


/* Send audio data read from the file `fd‘ to `port‘ on the local
 * host. */
int play_file(int port, int fd)
{
    int socket_fd ;             /* Socket file descriptor for
                                 * connection to AMEngine. */
    char buffer[BUFSIZ] ;       /* Read/write buffer for audio
                                 * data. */
    int ret ;                   /* Return value of read(2) or
                                 * write(2), respectively. */
    int size ;                  /* Amount of data currently in
                                 * buffer. */
    int written ;               /* Amount of data from buffer already
                                 * written. */

    /* Open socket connection. */
    if ((socket_fd = create_client_socket(„localhost“, port)) == -1) {
        perror(„create socket“) ;
        return -1 ;
    }

    /* Write data until EOF. */
    while ((ret = read(fd, buffer, BUFSIZ)) > 0) {
        size = ret ;
        written = 0 ;
        while (written < size) {
            ret = write(socket_fd, buffer + written, size - written) ;
            if (ret == -1) {
                perror(„write data“) ;
                return -1 ;
            }
            written += ret ;
            size -= ret ;
```

```
        }
    }
    if (ret == -1) {
        perror(„read data“) ;
        return -1 ;
    }
    close(socket_fd) ;

    return 0 ;
}


/* Create a client socket connected to port on hostname. */
int create_client_socket(char *hostname, int port)
{
    struct sockaddr_in sa ;
    struct hostent *hp ;
    int s ;

    memset(&sa, 0, sizeof(sa)) ;

    /* Get host‘s address. */
    if ((hp = gethostbyname(hostname)) == NULL) {
        return -1 ;
    }
    memcpy((char *) &sa.sin_addr, hp->h_addr, hp->h_length) ;
    sa.sin_family = hp->h_addrtype ;

    sa.sin_port = htons((u_short) port) ;

    /* Open socket and connect. */
    if ((s = socket(sa.sin_family, SOCK_STREAM, 0)) < 0) { /* get socket */
        return -1 ;
    }
    if (connect(s, &sa, sizeof(sa)) < 0) {                  /* connect */
        close(s) ;
        return -1 ;
    }
    return s ;
}
```

ANHANG C Audio/Video Exchange Protocol

Das Audio/Video Exchange Protocol (AVXP) ist für den Austausch von AV-Daten zwischen den verschiedenen AVC-Instanzen in einer MMC-Konferenz definiert. Ein AVXP-Datenstrom ist unidirektional. Er besteht aus Paketen mit einem Header und der daran angehängten Payload, die die eigentlichen AV-Daten enthält. .

Die einzelnen Felder des Headers haben folgende Bedeutung:

Source Internet Number
: IP-Adresse des sendenden Rechners. Dabei ist nicht die Adresse des Netzwerkinterfaces gemeint, das dieses Paket abgeschickt hat, sondern die Default-Adresse des sendenden Rechners, um diesen eindeutig zu identifizieren.

Sequence Number
: Nummer des aktuellen Pakets. Die empfangende AVC kann anhand dieser Nummer Pakete ordnen, die in der Reihenfolge durcheinandergekommen sind. (Das tut die AMEngine nicht, um die Verzögerung bei der Wiedergabe so gering wie möglich zu halten.)

Current Chunk Number
: Nummer des Teilstücks eines Videobildes, das im aktuellen Paket enthalten ist. Videobilder werden wegen ihrer Größe manchmal in mehreren Teilen (*Chunks*) ubertragen. Bei der Audio-übertragung enthält dieses Feld immer den Wert 1.

Total Number of Chunks
: Gesamtanzahl von Chunks im aktuellen Videobild. Bei der Audioübertragung enthält dieses Feld immer den Wert 1.

Offset
: Byte-Offset des aktuellen Chunks innerhalb des aktuellen Videobildes. Besteht das Videobild aus nur einem Chunks oder handelt es

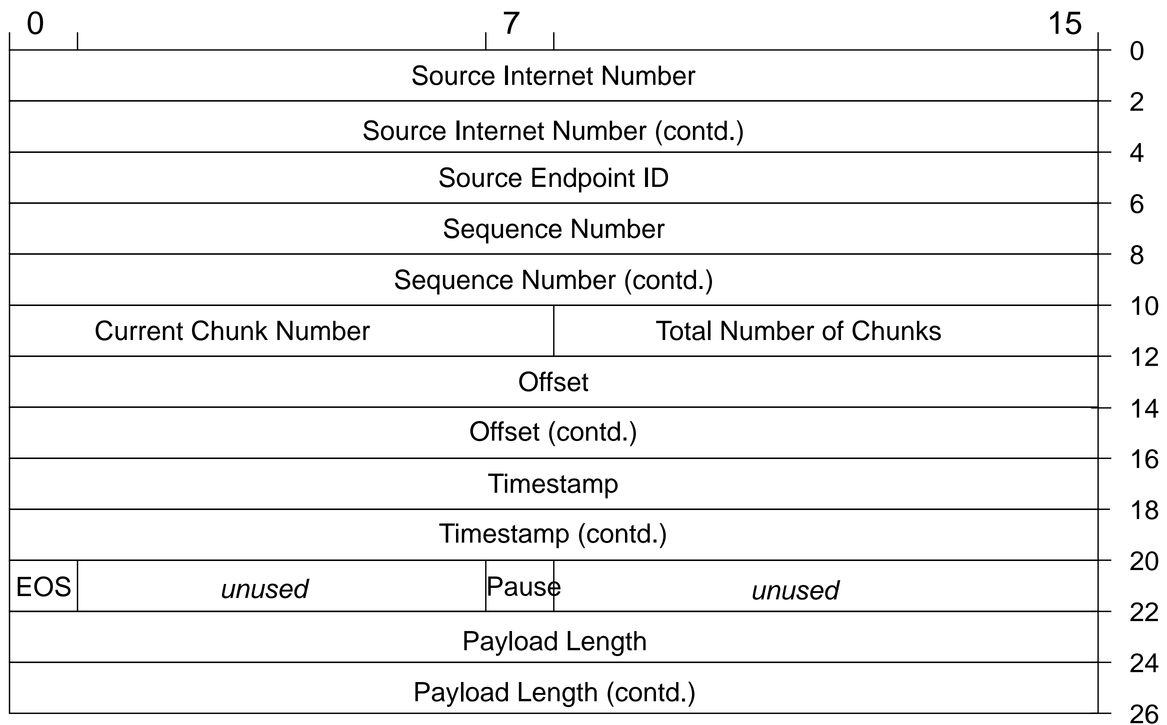

ABBILDUNG 9 AVXP-Paket-Header

sich um ein Audiopaket, ist also Total Number of Chunks gleich 1, gibt der Offset die gesamte Größe des Bildes bzw. der Audiodaten an, ist also gleich dem Wert von Payload Length (siehe unten).

Timestamp

Zeit in Millisekunden seit einem beliebigen Startpunkt in der Vergangenheit. Von einer Zeitsynchronisierung zwischen verschiedenen Rechnern in einer MMC-Konferenz kann nicht ausgegangen werden. Deshalb kann die Absendezeit eines Pakets nur relativ zu Paketen vom selben Rechner verstanden werden. Das reicht aber aus, um Video- und Audioströme von derselben Source zu synchronisieren.

EOS

„*End-Of-Stream*“-Flag. Das Setzen dieses Bits durch die Source signalisiert der Sink, daß dieses Paket das letzte in diesem Datenstrom ist.

Pause

„*Pause*“-Flag. Das Setzen dieses Bits durch die Source signalisiert der Sink, daß nach diesem Paket für eine gewisse (unspezifizierte) Zeit kein Paket folgt.

Payload Length

Länge der übertragenen Nutzdaten in Bytes.

ANHANG D Endpoint Control Protocol

Das Endpoint Control Protocol wurde für die Kommunikation zwischen dem AVC-Controller einerseits und dem Audio- oder Video-Subsystem andererseits entwickelt. Ich beschränke mich in dieser Beschreibung auf den Teil, der für die Kommunikation zwischen AVC-Controller und der AMEngine relevant ist.

Die im ECP ausgetauschten Nachrichten bestehen aus menschenlesbaren Textzeilen, die entweder über Unix-Pipes oder eine TCP-Netzwerkverbindung geschickt werden. Es gibt fünf verschiedene Typen von Nachrichten:

1. Kommandos, die von der steuernden Instanz an die AMEngine geschickt werden. Kommandos bestehen aus einem oder mehreren Worten auf einer einzelnen Zeile, die durch Leerzeichen oder Tabulatoren getrennt sind. Die maximale Zeilenlänge ist 256 Zeichen einschließlich eines abschließenden „Newline“-Zeichens.
2. Rückgabewerte, die von der AMEngine nach der erfolgreichen Bearbeitung eines Kommandos an die steuernde Instanz geschickt werden. Ein Rückgabewert kann aus mehreren Zeilen bestehen. Jede dieser Zeilen beginnt mit der Zeichenkette „RET “; die letzte Zeile besteht außschließlich aus der Zeichenkette „RET “.
3. Fehlermeldungen, die von der AMEngine an die steuernde Instanz geschickt werden, wenn die Ausführung eines Kommandos fehlgeschlagen ist. Eine Fehlermeldung besteht aus einer Zeile, die mit der Zeichenkette „ERR“ beginnt, gefolgt von einem numerischen Fehlercode und einem Text, der den Fehler beschreibt.

4. Asynchrone Fehlermeldungen, die von der AMEngine an die steuernde Instanz geschickt werden, wenn bei der Verarbeitung der Audiodaten Fehler auftreten.
5. Events, die von der AMEngine an die steuernde Instanz geschickt werden. Ein Event besteht aus einer Zeile, die mit der Zeichenkette „EVT „ besteht, gefolgt von einer Zahl, die den Event-Typ angibt, sowie weiteren Parametern, die vom Event-Typ abhängen.

Zusätzlich sendet die AMEngine beim Start entweder einen Rückgabewert, der die Version der laufenden AMEngine anzeigt, oder eine Fehlermeldung, die angibt, warum die AMEngine nicht starten konnte.

Kommandos

Bei den Kommandos, die die AMEngine auf der Steuerverbindung akzeptiert, müssen wir zwischen drei Kategorien unterscheiden:

1. Requests des Endpoint Control Protocol, wie es zwischen AVC-Controller und AMEngine definiert ist
2. Kommandos für Debugging- und Monitoring-Zwecke
3. Sonstige Kommandos des eingebauten Tcl-Interpreters

Der Interpreter enthält den vollen Satz an Tcl-Kommandos wie beschrieben in [Ousterhout 1994].

ECP-Requests

`opensource` *sample_rate coding stereo_flag ctype activity* [*options*]
`opensink` *sample_rate coding stereo_flag ctype activity* [*options*]

Öffne einen Source Endpoint oder Sink Endpoint mit der spezifizierten Charakteristik. Diese Charakteristik setzt sich zusammen aus der Abtastrate (in Hertz), der Kodierung (eine Zahl als Schlüssel für die Art der Kodierung), der Anzahl der Kanäle („0“ steht für mono, „1“ für stereo), dem Typ der Transportverbindung (Connection Type; zur Zeit „`udp`“ für UDP-Transport, „`tcp`“ für TCP-Transport), dem Activity-Typen („`live`“ oder „`appl`“) und weiteren Optionen. Die zur Zeit einzige Option ist „`noheaders`“ für eine Datenverbindung ohne AVXP-Header.

`addsink` *endpoint host port*

Füge der angegebenen Source Endpoint eine Sink hinzu, die sich auf dem Rechner *host* über den Port *port* erreichen läßt.

`remsink` *endpoint host port*

Entferne die Sink auf dem Rechner host am Port port vom angegebenen Source Endpoint.

`listav` [*activity*]

Liefere eine Liste der verfügbaren Datentypen für AV-Daten. (Die AMEngine ignoriert den optionalen *activity*-Parameter, weil die verfügbaren Datentypen für alle Activity-Typen gleich sind.

`close_ep` *endpoint*

Schließe den Endpoint *endpoint*. Falls es sich um einen Source Endpoint handelt, werden damit alle Verbindungen zu den angeschlossenen Sinks geschlossen.

`closeall`

Schließe alle Endpoints.

`transmit` *endpoint flag*

Setze den Übertragungsstatus für *endpoint*. Der Parameter *flag* gibt an, ob die Übertragung ein- oder ausgeschaltet wird („0" steht für aus, „1" steht für an).

`exit`

Beende das Programm.

`close`

Schließe diese Steuerverbindung.

`quit` *exitcode*

Beende das Programm mit dem Rückgabewert *exitcode*.

Debugging-/Monitoring-Kommandos

`help`

Gib einen Hilfstext aus, der alle ECP-Kommandos auflistet.

`version`

Gib Versionsinformationen aus. Die Ausgabe enthält Informationen über die Version der laufenden AMEngine, die benutzte Version von AIO, Datum der Kompilierung, den benutzten Compiler und die benutzten Optionen für Compiler und Linker.

`endpoints`

Zeige den Zustand aller Endpoints an.

`callbacks`

Zeige Informationen über die Callbacks in der Select-Loop an.

`debug`

Das debug-Kommando hat wiederum mehrere Unterkommandos:

`debug here`

Leite Debugging-Meldungen (so welche aktiviert sind) auf die aktuelle Steuerverbindung.

`debug stderr`

Leite Debugging-Meldungen auf die Standardfehlerausgabe.

`debug set` *topic*

Aktiviere eine oder mehrere Gruppen von Debugging-Meldungen.

`debug show`

Zeige den Aktivierungsstatus der Gruppen von Debugging-Meldungen.

`debug clear flag1 ...`

Deaktiviere eine oder mehrere Gruppen von Debugging-Meldungen.

`debug clearall`

Deaktiviere alle Gruppen von Debugging-Meldungen.

`debug on`

Ermögliche Debugging-Meldungen

`debug off`

Schalte Debugging-Meldungen ab.

`debug dummy`

Leere Prozedur; benutzt als Breakpoint-Position (dbg_summy()) für den Debugger.