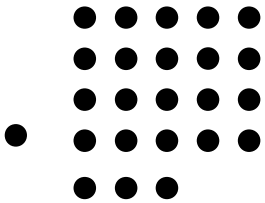

**Fachhochschule Köln**
**University of Applied Sciences Cologne**

**07 Fakultät für**
**Informations-, Medien- und Elektrotechnik**

Studiengang Elektrotechnik

Institut für Nachrichtentechnik
Labor für Datennetze

# Diplomarbeit

| | |
|---|---|
| Thema: | **Entwicklung eines Videoanalysesystems auf Basis von Java Mobile Edition für Symbian SmartPhones** |
| Student : | **Marek Urban** |
| Matrikelnummer: | **11046552** |
| Referent : | **Prof. Dr.-Ing. Andreas Grebe**<br>**Fachhochschule Köln** |
| Korreferent : | **Prof. Dr. Carsten Vogt**<br>**Fachhochschule Köln** |
| Abgabedatum : | **26.02.09** |

Hiermit versichere ich, dass ich die Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt habe.

_____

Marek Urban

# Inhaltsverzeichnis

# 1. Einleitung

Der Markt für mobile Telefone wächst schon seit mehreren Jahren rasant. Das so genannte „Handy“ wird schon lange nicht mehr nur zum Telefonieren benutzt. Es ist zu einem nicht mehr wegzudenkenden Assistenten geworden, der immer einsatzbereit und in Reichweite ist. Das Gerät kann außer den Grundfunktionen wie Telefonieren und SMS-Verschicken als persönlicher Terminplaner, Notizbuch oder auch als Foto- oder sogar Videokamera genutzt werden.

In den letzten Jahren sind Mobiltelefone auch in der Internet- und Multimediawelt vertreten. Sie besitzen mittlerweile genug Leistung, Speicher und Bildschirmfläche, um Webseiten darzustellen oder komplette Spielfilme wiederzugeben.

Die sinkenden Preise für mobile Breitband-Internetzugänge (Flatrate mit 7,2 Mbit/s schon ab 35 Euro), die wachsende Anzahl der mobilen Internet-Zugänge, die sinkenden Nutzungspreise der so genannten „HotSpots“ (in Hotels, Restaurants, etc. oft kostenfrei) und die Leistungsfähigkeit der aktuellen mobilen Endgeräte, ermöglichen die Nutzung der Videostreaming-Dienste. Diese Dienste können zum Beispiel IPTV, Video on Demand oder Videotelefonie sein.

Um die Qualität des über ein IP-Netzwerk gelieferten Videodatenstrom zu gewährleisten, ist es notwendig die Qualität dieses Datenstroms im realen Umfeld und unter realen Bedingungen messen zu können. Das erfordert die Entwicklung eines Messsystems, welches direkt auf einem Endgerät installierbar und vom Gerätetyp möglichst unabhängig ist.

Das zu entwickelnde Programm soll den Videodatenstrom mitschneiden und auswerten können ohne die Übertragung zum Wiedergabeprogramm zu stören. Es soll der Paketverlust, der Wert des Interarrival Jitters sowie die für das jeweilige Videoformat spezifischen Informationen erfasst werden. Die Auswertung soll in vom Benutzer bestimmbaren Intervallen durchgeführt werden können.

Um diese Anforderungen zu erfüllen, wird eine Anwendung in der

Programmiersprache Java Mobile Edition implementiert, die wie eine Netzwerkbrücke funktionieren wird. Sie wird Java-Threads für das parallele Empfangen und Weiterleiten der Datenpakete zwischen dem Server und dem Wiedergabeprogramm sowie für das Extrahieren und Auswerten der Daten verwenden.

In Kapitel 2 werden die meisten Begriffe, die in der Diplomarbeit vorkommen erläutert.

Einen wichtigen Punkt stellen in diesem Kapitel die Protokolle für den Transport der multimedialen Daten.

Für die Darstellung von Videodaten auf mobilen Endgeräten hat sich der MPEG-4-Standard durchgesetzt. Dessen Entwicklung hat als Ziel, Systeme mit geringen Ressourcen oder schmalen Bandbreiten bei relativ geringen Qualitätseinbußen zu unterstützen. Auch dieser Standard wird hier dargestellt.

Die Wahl der Programmiersprache, deren Hauptaspekte und die zur Fertigstellung des Messsystems nötigen Komponenten werden zum Schluss dieses vorgestellt.

In Kapitel 3 wird die Umsetzung des Systems beschrieben: Zuerst die Darstellung des Entwicklungssystems und der Entwicklungsumgebung (3.1), danach werden zwei Alternativen genannt, nach denen die Entwicklung des Grundgerüsts der Software möglich ist (3.2). In Abschnitt 3.3 werden die Methoden präsentiert, die den Kern des Messsystems bilden und die so genannte „Deep Inspection“ des Videodatenstroms durchführen.

# 2. Hintergrundwissen

## 2.1 Das Medium

Diese Diplomarbeit beschäftigt sich mit Video auf mobilen Endgeräten. An dieser Stelle wird erläutert, weshalb von „dem Medium“ die Rede ist.

Das Wort „Medium“ kommt von dem lateinischen Wort „medium“ und bedeutet „Mitte“, „Mittelpunkt“ oder „etwas dazwischen Liegendes“. Ein Medium ist ein Mittel zum Speichern oder Übertragen von Daten [Brockhaus03]. Elektronische Medien sind zum Beispiel der Computer, das Fernsehen, das Telefon oder das World Wide Web. Jedes dient dem Informationsaustausch zwischen Kommunikationspartnern (der Kommunikation). Sie können auch andere Medien enthalten, wie zum Beispiel das World Wide Web auch audiovisuelle Medien enthält. Audiovisuelle Medien bestehen aus Video- und Tonspuren, die auch Medien sind.

Das letzte, als reine Medium bezeichnet, das keine Medien mehr enthält, nennt man Daten. Daten eignen sich nicht zur zwischenmenschlichen Kommunikation, da die räumliche und zeitliche Dimension des Mediums nicht direkt sichtbar ist.

Menschen sind die Verwendung ihrer Sinne zur Kommunikation gewohnt und übernehmen diese Gewohnheit für die Kommunikation mit elektronischen Medien [Eidenberger03].

## 2.2 Medienverarbeitung

Die Medien Audio und Video sind so genannte gezeitete Datenströme (timed Streams). Ein gezeiteter Datenstrom besteht aus Teilen, die jeweils eine Zeitmarke haben. Video kann zum Beispiel aus einzelnen Bildern (25 Bilder pro Sekunde im mitteleuropäischen Fernsehen) bestehen, von denen jedes eine eigene Zeitmarke hat. Es kann in zwei Formen vorliegen: als Strom oder als Pool. Als Strom ist das Medium von der Zeit abhängig. Es verändert sich mit der Zeit. Als Pool wird ein

Medium bezeichnet, das komplett als digitaler Datenbereich vorliegt und auf das beliebig zugegriffen werden kann. Einfach ausgedrückt, ist ein Strom eine Folge von Paketen und ein Pool eine Datei.

Die Verarbeitung elektronischer Medien, kurz Medienverarbeitung, befasst sich mit der Kodierung, Komposition, Verpackung und dem Transport eines Mediums.

Mit einer Videokamera aufgenommene Videodaten beanspruchen Unmengen Speicherplatz. Die Kompression ist der wichtigste Aspekt der Kodierung. Mit deren Hilfe werden Redundanzen und unwesentliche Teile eines Mediums beseitigt. Das Entfernen von Redundanzen nennt man verlustfreie Kompression, das Entfernen unwesentlicher Teile verlustbehaftete Kompression. Es gibt aber auch Fälle, in denen die Redundanzen eingeflochten werden, um die Qualität der Darstellung zu verbessern oder die Sicherheit des Transports zu erhöhen.

Unter „Komposition“ versteht man das zeitliche und räumliche Zusammenfügen von Medien. Damit ist auch die Synchronisation von Teilmedien und die Anwendung von Effekten gemeint.

Die Begriffe „Verpackung“ und „Transport“ sind eigentlich selbsterklärend. Mehrere, voneinander unabhängige Medien können zusammengepackt (als Multiplexing bekannt) und als gemeinsamer Transportstrom (Streaming) weitergeleitet werden [Eidenberger03].

## 2.3 Videokompression

In Zeiten, in denen die Netzwerke immer größere Bitraten ermöglichen und Massenspeicher-Geräte immer größere Kapazitäten besitzen und immer günstiger werden, könnte man meinen, dass die Entwicklung neuerer, leistungsstärkerer Videokompressionsarten nicht mehr so wichtig sei. Doch das Internet wird noch lange nicht in der Lage sein, unkomprimierte („raw“) Videodaten zu transportieren. Sogar die DVD (Digital Versatile Disk) kann nur einige Sekunden solcher Daten in TV-Qualität speichern. Für die Übertragungs- und Speicherplatzkapazität ist und

bleibt die Videokompression ein entscheidender Aspekt der Multimediaverarbeitung.

Das Videosignal kann durch Beseitigen von in ihm enthaltenen Redundanzen komprimiert werden. Beim verlustfreien Komprimieren werden statistische Redundanzen entfernt. Das Signal verliert dabei nicht an Qualität. Das geschieht jedoch auf Kosten geringerer Kompressionsraten. Größere Kompressionsraten erzielt man mit einer verlustbehafteten Videokomprimierung, bei der aber das Videosignal an Qualität verliert. Das Hauptziel der Kompressionsalgorithmen ist eine möglichst hohe Kompressionsrate bei möglichst geringen Qualitätsverlusten zu erlangen.

Die Videokompressionsalgorithmen beseitigen Redundanzen im Zeit-, Raum- und / oder Frequenzbereich:

- Im Zeitbereich nutzt ein Encoder (Kodierer) die Ähnlichkeiten zwischen zwei benachbarten Einzelbildern aus, um eine Prädiktion[1] für das aktuelle Einzelbild zu berechnen. Dabei wird nur der Unterschied zwischen den Bildern, zum Beispiel in Form von Bewegungsvektoren gespeichert.
- Im Raumbereich wird das zuvor im Zeitbereich verarbeitete Bild weiter komprimiert. Dazu werden Ähnlichkeiten zwischen zwei benachbarten Mustern eines Videobildes ausgenutzt. In MPEG-4 und H.264 werden diese mit Hilfe einer Transformation in einen anderen Bereich portiert, dort als Koeffizienten dargestellt und anschließend Quantisiert. Die Quantisierung dient der Entfernung vernachlässigbarer Koeffizienten.
- Die Bewegungsvektoren, als Ergebnis der Zeitbereich-Verarbeitung und die Koeffizienten, als Ergebnis der Raumbereich-Verarbeitung wandern an den „Entropie-Kodierer“ (entropy encoder), der schließlich statistische Redundanzen entfernt. Das Ergebnis dieses Kodierers ist ein komprimierter Bitstrom, der als Datei gespeichert oder gesendet wird [Richardson03].

## 2.4 Datenformate, Kompressionsverfahren und Codecs

Unter Video versteht man grundsätzlich eine Folge digitalisierter Bilder. Das menschliche Auge nimmt diese als Bewegung wahr, wenn sie mindestens 24 Bilder pro Sekunde schnell sind. Bei einer Bildfolge kann zwischen Film und Animation

1 Eine Prädiktion nennt man eine Voraussage darüber, wie das aktuelle Bild aussehen wird.

unterschieden werden. Filme sind detailreich, da sie auf der aufgenommenen Wirklichkeit basieren. Animationen sind detailarm, weil sie die Wirklichkeit nur nachahmen. Sie bestehen aus klar strukturierten Objekten und wenigen Farbstufungen [Eidenberger03].

Filme und Animationen können komprimiert oder unkomprimiert vorliegen. Unkomprimierter Film liegt zum Beispiel im D1-Format vor. Ein komprimierter Film kann mit dem bekannten MPEG-2-Kompressionsverfahren verarbeitet werden, das unter anderem für DVDs verwendet wird. Animationen können unkomprimiert als Shockwave-Dateien (Macromedia Flash) vorliegen oder komprimiert als Listen von GIF-Bildern.

Ein Codec (Compressor / Decompressor oder Komprimierer / Dekomprimierer ) ist ein Programm oder ein Gerät, das Kompressionsverfahren anwendet und eine Multimediadatei komprimiert. Zum Abspielen dieser Multimediadatei wird diese Komprimierung wieder rückgängig gemacht und mit Hilfe einer Abspiel-Software wiedergeben [Brockhaus03].

Ein zeitabhängiges, elektronisches Medium besteht aus einem oder mehreren Tracks (zum Beispiel einem Audio- und einem Videotrack) mit bestimmten Medienformaten, die innerhalb eines Tracks immer dieselben sind. Die Tracks werden wiederum in Elemente und Samples unterteilt. Für die Weiterverarbeitung werden innerhalb eines Tracks die Samples in gleich große Gruppen, so genannte Chunks zusammengefasst. Jeder von diesen Chunks ist mit einer Zeitmarke versehen, die sein „Ablaufdatum“ markiert (Zeit, in der er angezeigt wird).

Grundsätzlich wird zwischen den eigentlichen Datenformaten, die auch Container-Formate genannt werden und den verwendeten Kodierungs-, bzw. Kompressionsverfahren (Codecs) unterschieden. Einige Datenformate sind auf spezielle Kompressionsverfahren beschränkt, andere unterstützen mehrere unterschiedliche Verfahren.

Im Folgenden werden einige Kodierungs- bzw. Kompressionsverfahren für Video

angeführt:

- MJPEG basiert auf der Kompression der einzelnen Bilder des Videostroms mit Hilfe des JPEG-Verfahrens. Nachteil: geringe Kompression, Vorteil: Zugriff auf einzelne Bilder.
- Das MPEG-1 Verfahren basiert prinzipiell darauf, dass die Einzelbilder eines Videostroms voneinander abhängig sind. Es ist somit nicht notwendig, jedes Bild einzeln zu kodieren, sondern nur die Veränderungen zum vorherigen Bild. Ein MPEG-Videostrom setzt sich aus vier verschiedenen Einzelbildern zusammen:
  - I-Bilder (Intra Coded Pictures) sind vollständig kodierte Bilder. Sie haben keinen Bezug zu anderen Bildern. Makroblöcke (16 x 16 Bildpunkte große Teile des Bildes) werden im JPEG-Verfahren komprimiert.
  - P-Bilder (Predictive Coded Pictures) beziehen sich auf die Differenz zum letzten I- oder P-Bild. Makroblöcke, die sich nicht verändert haben werden durch einen Bewegungsvektor mit der Angabe der neuen Position des Makroblocks ersetzt.
  - B-Bilder (Bidirectionally Predictive Coded Pictures) verwenden für die Berechnung der Bewegungsvektoren zusätzlich das nachfolgende I- oder P-Bild.
  - D-Bilder (DC Coded Pictures) beziehen sich wie die I-Bilder auf keine anderen Bilder des Videostroms. Bei den D-Bildern werden aber ausschließlich niederfrequente Teile des Bildes berücksichtigt. Diese Bilder eignen sich besonders für einen schnellen Vorlauf oder Index-Bilder.

  Die maximale Auflösung bei diesem Verfahren ist 352 x 288. Die maximale Datenrate beträgt 1,5 Mbit/s, die maximale Kompressionsrate 40:1
- MPEG-2 erweitert das MPEG-1-Verfahren um neue Funktionen und Auflösungen. Es ist in Profiles, in denen die verscheidenen Funktionalitäten beschrieben sind, und Levels, in denen die möglichen Auflösungen definiert sind, unterteilt. Das Verfahren wird bei Video auf DVD verwendet und als Grundlage für HDTV gebraucht.
- MPEG-4 ist ein innovatives Kompressionsverfahren. Die einzelnen Bilder

werden nicht mehr als Pixelmatrizen bearbeitet, sondern es wird zwischen dem Hintergrund und den Objekten im Vordergrund unterschieden. Diese Objekte können Audio-, Video- oder Multimedia-Daten-Objekte (Text oder Animation) sein. Zwei Schwerpunte dieses Verfahrens sind: Die Möglichkeit der Manipulation von den Objekten ohne vorherige Dekodierung und die Robustheit beim Einsatz in fehleranfälligen Netzwerkumgebungen. Mehr Details zu MPEG-4 werden in Kapitel 2.6 beschrieben.

- WMV (Windows Media Video) wurde von der Firma Microsoft entwickelt.
- DivX ist ein MPEG-4 kompatibler Codec.
- Die von der ITU (International Telecommunications Union) entwickelten Verfahren: H.261, H.263, H.264 (auch unter MPEG-4 Part 10/AVC „Advanced Video Coding“ bekannt) werden zum Beispiel für die Videotelefonie verwendet.

An dieser Stelle folgt eine Aufzählung einiger Container-Formate für Video:

- AVI (.avi, Audio Video Interleave) wurde von Microsoft geschaffen und unterstützt eine Vielzahl an Audio- und Video-Kodierungsverfahren. Es hat aber einen Nachteil: Es ist nicht streamingfähig.
- QuickTime (.mov, .qt) ist Apples streamingfähiges Pendant zu Microsofts AVI .
- WMV und ASF (.wmv, .asf, Windows Media Video, Advanced Systems Format) sind Microsofts streamingfähige Videoformate, wobei WMV nur den Windows Media Video-Codec unterstützt, ASF aber beliebige Codecs.
- MPEG-Formate (.mp2, .mpv, .dat, .vob, .mp4, .dat) dabei werden .dat-Formate für Video-CDs und .vob-Formate für DVDs verwendet [Eidenberger03].

## 2.5 Streaming

Die Kommunikation mit Hilfe von Audio und Video ist für Menschen am natürlichsten (im direkten Gespräch werden audiovisuelle Nachrichten ausgetauscht), viel natürlicher als Bild- und Textnachrichten. In der ersten Dekade des Bestehens von Internet wurden fast ausschließlich die zweiten verwendet. Den Erfolg des

Fernsehens kann man als Vorbild nehmen.

Die ersten Multimedia-Anwendungen haben das Internet als Übertragungsmedium zum PC verwendet. Der Inhalt konnte erst nach dem vollständigen Herunterladen der Datei abgespielt werden. Beim Streaming wird der Inhalt in Realzeit zu dem abzuspielenden Programm gestreamt, von diesem direkt dargestellt und sofort gelöscht.

Mit den immer günstiger werdenden und an immer mehr Orten verfügbaren Hoch-Geschwindigkeits-Internet-Anschlüssen (wie z.B. DSL[2]) wurde der Grundstein für das Streaming über IP-Netzwerke gelegt. Zwischen vielen Streaming-Standards wurde der MPEG-4-Standard der Erfolgreichste. Zu den erfolgreichsten Herstellern von Produktion-, Server- und Abspiel-Software zählen RealNetworks und Apple. Beide unterstützen eine Vielzahl an Formaten, auch MPEG-4. MPEG-4 ermöglicht Interaktion auf der Empfänger- und Serverseite. Dieser Standard ist sehr skalierbar und deckt Auflösungen von sehr geringen, für mobile Endgeräte geeigneten, bis zu sehr hohen für das hochauflösende Fernsehen [Austerberry05].

Anwendungen, bei denen ursprünglich CD-ROMs zur Distribution von Video-Inhalten verwendet wurden, könnten heutzutage das Streaming als das Transport-Medium benutzen. Folgende Bereiche könnten diese Technologie verwenden: Schulung, Unternehmenskommunikation, Marketing und Unterhaltung (Abbildung 1).

2 Digital Subscriber Line, eine Reihe von übertragungsstandards der Bitübertragungsschicht

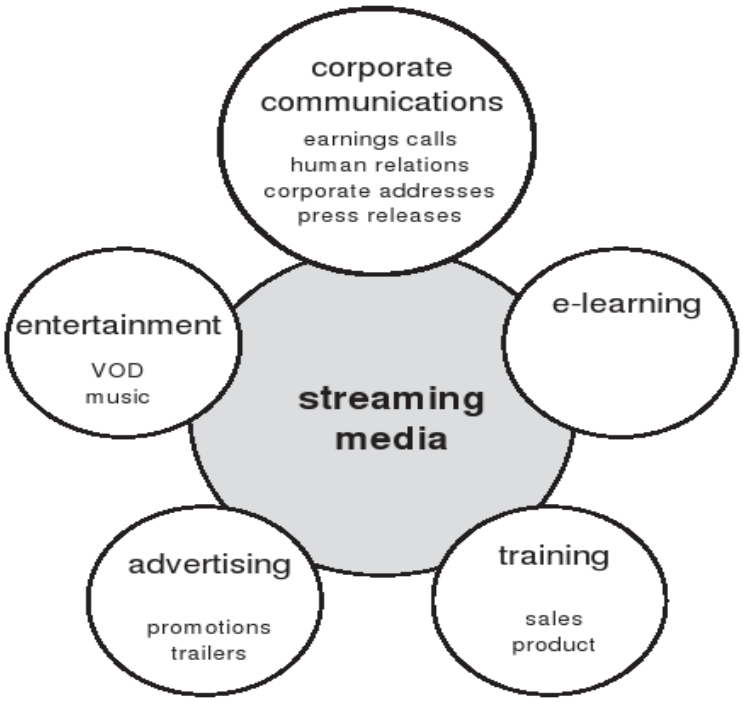


Abbildung 1: Streaming, Quelle: [Austerberry05]

Streaming kann auf zwei Arten benutzt werden: als Live-Streaming, bei dem das Medium kontinuierlich an den Empfänger geschickt (beispielsweise IPTV) und als On-Demand-Streaming (Abbildung 2), bei dem der Empfänger den Multimedia-Inhalt beliebig oft anfordern, spulen, pausieren, stoppen oder neustarten kann. Beim Multimedia-Streaming an mobilen Endgeräten kommt Video on Demand, kurz VoD am häufigsten vor. Die Dateien werden meistens im MPEG-4-Format kodiert. Mehr dazu in Kapitel 2.6.

Das Anfordern von WEB-Inhalten geschieht in der Regel mit Hilfe von HTTP und TCP. Wird eine Datei angefordet, dann wird sie schnellstmöglich heruntergeladen. TCP garantiert eine fehlerfreie Transmission der Inhalte durch wiederholtes Senden fehlerhafter Pakete. Die Transfer-Geschwindigkeit ist von der verfügbaren Bandbreite abhängig. Beim „Streamen“ ist aber die Ankunft der Daten in einer fest definierten Geschwindigkeit wichtiger als einige wenige Fehler im Datenstrom. Für den Videotransport wird daher UDP bevorzugt, mit oder ohne RTP. Das Anfordern und Steuern eines zu streamenden Videos geschieht HTML-ähnlich mit Hilfe von RTSP.

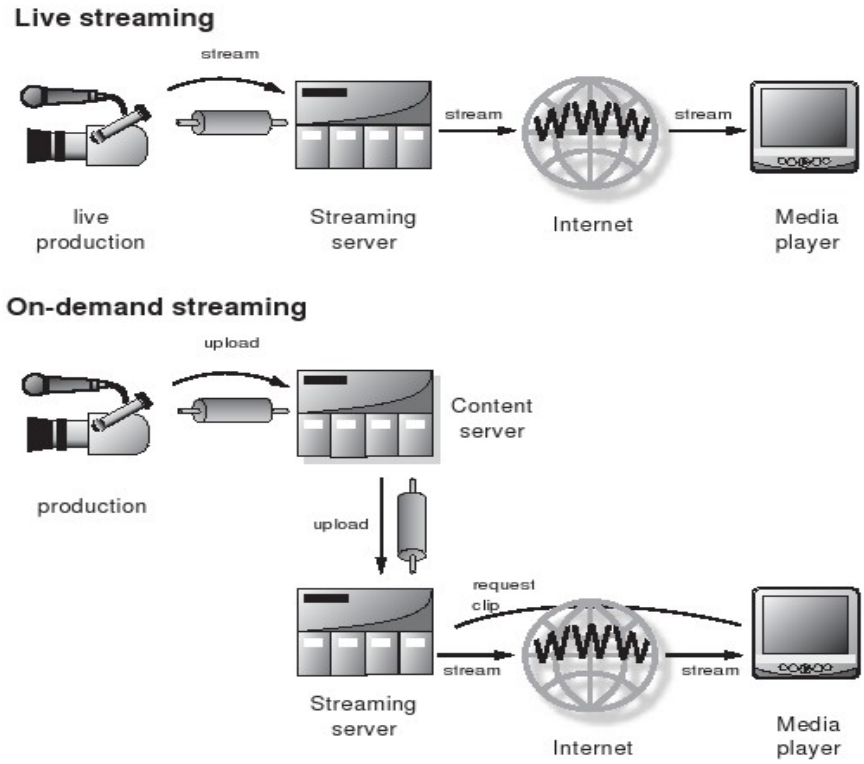


Abbildung 2: Video-Streaming, Quelle: [Austerberry05]

### 2.5.1 RTSP

RTSP ist ein Netzwerkprotokoll der Applikationsschicht zum Steuern von echtzeitintensiven Datenströmen, zum Beispiel von audiovisuellen Inhalt über IP-basierte Netzwerke. Die zu liefernden Daten können Echtzeit- oder on-Demand-Medien sein.

Bei RTSP handelt es sich um ein text-basiertes Protokoll. Dieses kann über TCP oder UDP übertragen werden und mehrere Sitzungen verwalten. Es ähnelt im Aufbau und Verhalten dem HTTP-Protokoll. RTSP kennt im Gegensatz zu HTTP Zustände und ist bidirektional, das heißt, dass sowohl Client als auch Server Anfragen absetzen können. Anfragen bezüglich eines Streams können von verschiedenen TCP-Verbindungen erfolgen.

Eine RTSP-URL hat folgenden Aufbau:

```
rtsp_URL=("rtsp:"|"rtspu:")"//"host[":"port][abs_path]
```

Die folgenden Methoden spielen in RTSP eine zentrale Rolle beim Lokalisieren und Benutzen von sich auf dem Streaming-Server befindenden Medien:

- DESCRIBE befielt dem Server, das Medium zu beschreiben.
- SETUP weist den Server an, die Resourcen vorzubereiten und die RTSP-Session zu starten.
- PLAY und RECORD starten die Übertragung.
- PAUSE hält die Übertragung temporär an.
- TEARDOWN gibt die Ressourcen frei, die Session wird auf dem Server gelöscht [RTSP_RFC].

Beispiel einer Client-Server Konversation (C: Client, S: Server [RTSP_RFC]):

```
C->S: DESCRIBE rtsp://foo/twister RTSP/1.0
      CSeq: 1

S->C: RTSP/1.0 200 OK
      CSeq: 1
      Content-Type: application/sdp
      Content-Length: 164
      v=0
      o=- 2890844256 2890842807 IN IP4 172.16.2.93
      s=RTSP Session
      i=An Example of RTSP Session Usage
      a=control:rtsp://foo/twister
      t=0 0
      m=audio 0 RTP/AVP 0
      a=control:rtsp://foo/twister/audio
      m=video 0 RTP/AVP 26
      a=control:rtsp://foo/twister/video

C->S: SETUP rtsp://foo/twister/audio RTSP/1.0
      CSeq: 2
      Transport: RTP/AVP;unicast;client_port=8000-8001
```

```
S->C: RTSP/1.0 200 OK
      CSeq: 2
      Transport: RTP/AVP;unicast;client_port=8000-
                 8001;server_port=9000-9001
      Session: 12345678

C->S: PLAY rtsp://foo/twister RTSP/1.0
      CSeq: 4
      Range: npt=0-
      Session: 12345678

S->C: RTSP/1.0 200 OK
      CSeq: 4
      Session: 12345678
      RTP-Info: url=rtsp://foo/twister/video;
                seq=9810092;rtptime=3450012

C->S TEARDOWN rtsp://foo/twister RTSP/1.0
     Cseq:7
     Session: 12345678

S->C: RTSP/1.0 200 OK
      CSeq: 7
      Session: 12345678
```

### 2.5.2 SDP

In der Antwort auf die DESCRIBE-Anfrage sendet der Server Eigenschaften des Multimediadatenstrom als SDP- (Session Description Protocol) Parameter. Das Session Description Protocol wird zur Initiierung und zum Austausch von Konfigurationsparametern bei Multimedia-Konferenzen, VOIP[3]-Anrufen, Videostreaming und vielen anderen Anwendungen genutzt [RFC2327].

3 VOIP: Voice-over-IP, IP-Telefonie

Die wichtigsten Parameter sind:

- v = Protokoll-Version
- o = Ersteller der Session und Session-Identifizierung
- s = Name der Session
- i = zusätzliche Session-Informationen
- u =URI mit der weiterführender Beschreibung
- m = Medienname und Adresse (IP-Adresse und Port)
- t = Zeit, in der die Session aktiv ist [RFC2327]

### 2.5.3 RTP

Real-Time Transport Protocol (RTP) definiert die Ende-zu-Ende Übertragung von audiovisuellen Daten mit Echtzeitanforderungen. Dieses Protokoll beinhaltet Dienste wie Nutzdaten-Identifikation, Sequenz-Nummerierung, das Markieren mit Zeitstempeln und die Zustellungs-Kontrolle. RTP wird im Allgemeinen über UDP betrieben, dieses ist aber nicht zwingend erforderlich. Zudem nutzt es seine Eigenschaften wie Multiplexing und Prüfsummen-Berechnung. Dieses Transport-Protokoll selbst sieht keine Mechanismen zur Sicherstellung der Quality-of-Service Bedingungen oder zur Transfer-Dauer der Daten vor. Es arbeitet aber mit RTCP (Real-Time Control Protocol) zusammen, das die Quality-of-Service-Parameter untersucht und sie an den Server und den Client meldet. Der Aufbau eines RTP-Headers ist in Abbildung 3 dargestellt.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td colspan="2">V=2</td><td>P</td><td>X</td><td colspan="4">CC</td><td>M</td><td colspan="7">PT</td><td colspan="16">sequence number</td></tr>
<tr><td colspan="32">timestamp</td></tr>
<tr><td colspan="32">synchronization source (SSRC) identifier</td></tr>
<tr><td colspan="32">contributing source (CSRC) identifiers</td></tr>
</table>

Abbildung 3: RTP Header

Version (V), 2 Bit
Die ersten zwei Bits stellen die Version des RTP-Protokolls dar.

Padding (P), 1 Bit

Das Füll-Bit ist gesetzt, wenn ein oder mehrere Füll-Oktets am Ende des Pakets angehängt sind und diese aber nicht zum eigentlichen Dateninhalt gehören. Das letzte gibt die Anzahl der hinzugefügten Füll-Oktets an.

Extension (X), 1 Bit
Das Erweiterungs-Bit ist gesetzt, wenn der Header um genau einen Erweiterungs-Header ergänzt wird.

CSRC Count (CC), 4 Bit
Der CSRC-Zähler gibt die Anzahl der CSRC-Identifier an, die dem konstanten Teil des Headers folgen.

Marker (M), 1 Bit
Das Marker-Bit ist anwendungsspezifisch.

Payload Type (PT), 7 Bit
Dieses Feld beschreibt das Format des zu transportierenden RTP-Inhalts.

Sequence Number, 16 Bit
Die Sequenznummer wird für jedes weitere RTP-Datenpaket erhöht. Die Startnummer wird zufällig ausgewählt. Der Empfänger kann mit Hilfe der Sequenznummer die Paket-Reihenfolge wiederherstellen und den Verlust von Paketen erkennen.

Timestamp, 32 Bit
Der Zeitstempel bezieht sich auf das erste Oktet des RTP-Datenpakets. Der Stempel muss sich an einem Takt orientieren, der kontinuierlich und linear ist, damit die Laufzeitunterschiede der Übertragungsstrecke (Jitter) ermittelt werden können und der Datenstrom synchron wiederhergestellt werden kann. Der Startwert sollte wieder ein zufälliger Wert sein. Aufeinanderfolgende Pakete können den gleichen Zeitstempel haben, wenn die transportierten Daten zum Beispiel zum selben Videopaket gehören. Pakete mit aufeinanderfolgenden Sequenznummern können aber auch nicht aufeinanderfolgende Zeitstempel enthalten.

SSRC, 32 Bit
Dieses Feld dient zur Identifikation der Synchronisationsquelle. Der Wert wird zufällig ermittelt, damit zwei Quellen innerhalb der RTP-Session immer verschiedene Identifikationsnummern besitzen.

CSRC List, 0 bis 15 Felder je 32 Bit
Die CSRC-Liste dient zur Identifikation der Quellen, die im RTP-Payload enthalten sind. Wenn mehr als 15 Quellen vorkommen, werden nur die ersten 15 identifiziert. Die Liste wird von Mixern eingefügt, die dazu den Inhalt des SSRC-Feldes der beteiligten Quellen einsetzen [RFC3550].

### 2.5.4 RTCP

Das RTCP (RTP Control Protocol) basiert auf periodischem Versenden von Reports über den aktuellen Zustand des Netzwerkes zwischen Sender und Empfänger. Es hat folgende Aufgaben:

- Das liefern des Feedbacks über die Qualität der Daten-Distribution. Dazu gehören: Fluss- und Staukontrolle. Dieses Feedback ermöglicht den Teilnehmern, die von ihnen generierte Datenrate zu begrenzen. Jeder Sender und Empfänger überträgt periodisch die so gennanten Sender Reports (SR) und Receiver Reports (RR).
- Für jeden Teilnehmer ist eine konstante, eindeutige Kennzeichnung (Canonical Name, CNAME) zu übertragen, da die SSRC sich ändern kann. Diese Kennzeichnung wird vom Empfänger auch dazu verwendet, um mehrere Datenströme eines Teilnehmers (z.B. Audio und Video), die in getrennten Sessions übertragen werden, wieder zusammenzubringen.
- Die Kontrolldaten müssen zwar periodisch von allen Teilnehmer übertragen werden, dürfen jedoch bei großen Teilnehmerzahlen nicht die Bandbreite für die Nutzdaten einschränken. Da aber jeder Teilnehmer alle RTCP-Pakete erhält, kennt RTCP die Gesamtzahl der Teilnehmer und kann daraus die Häufigkeit berechnen, mit der die Kontrollpakete gesendet werden.
- Optional kann jeder Teilnehmer via RTCP weitere Informationen über sich selbst versenden (Source Description, SDES), die z.B. in der

Benutzungsoberfläche der Anwendung angezeigt werden können.

Um diese Funktionalitäten anbieten zu können, werden folgende fünf Paketformate eingeführt:

- SR: Senderbericht
- RR: Empfängerbericht
- SDES: Quellenbeschreibung mit dem CNAME
- BYE: Zum Beenden der Sitzung
- APP: Anwendungsspezifische Funktionen

Von den Nachrichten werden mehrere (mindestens zwei) zu einem Paket zusammengefasst. Um die Funktionalitäten möglichst gut zu unterstützen, sollten in jedem zusammengefassten Paket (compound packet) Empfänger- (RR) oder Senderinformationen (SR) und ein Paket zur Quellenbeschreibung (SDES) enthalten sein. Ein RTCP Sender Report ist in Abbildung 4 dargestellt.

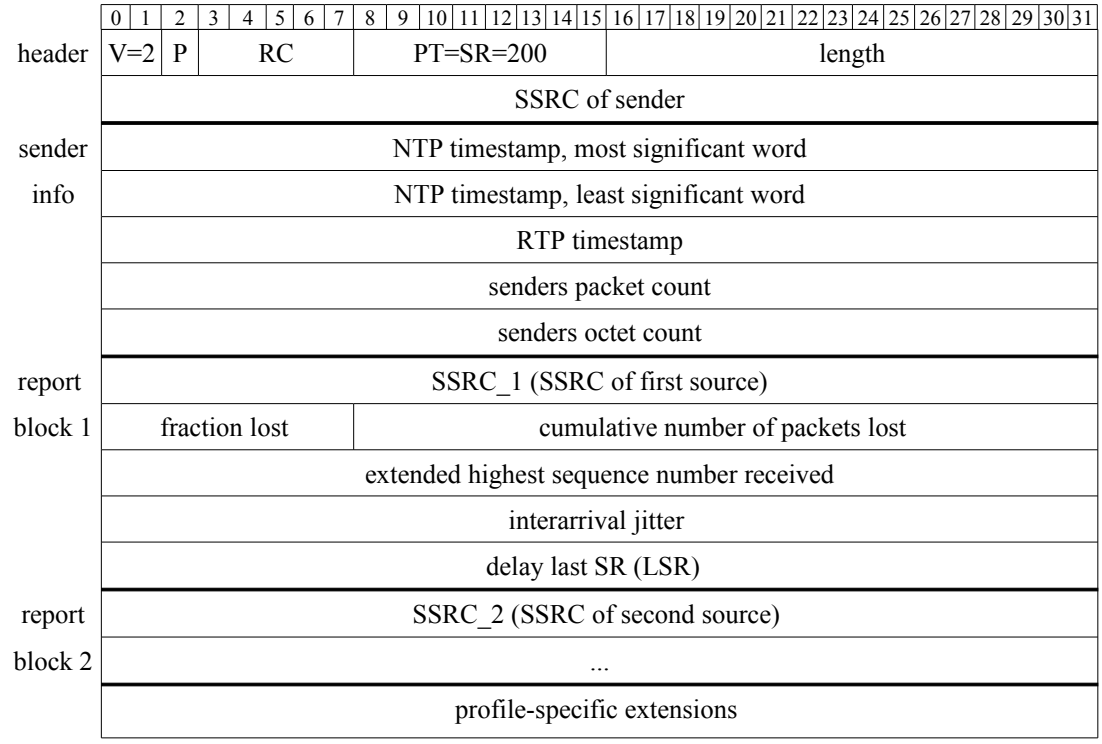


Abbildung 4: RTCP Sender Report. Quelle [RFC3550]

Das SR-Paket besteht aus drei Teilen:

- Im ersten Teil befinden sich Informationen über die RTP-Version, Länge des Paketes und der Identifikationsnummer des Senders.
- Im zweiten Teil ist ein Zeitstempel in zwei Formen enthalten: als NTP-Zeitstempel in den gleichen Einheiten und dem gleichen Offset wie bei den RTP-Paketen. Sie dienen der Inter- und Intra-Media-Synchronisation.
- Die Anzahl der Berichte im dritten Teil des Paketes hängt von der Anzahl der empfangenden Quellen ab. In diesem Teil wird die Zahl der verloren gegangenen Pakete, die Schwankungen des Zeitintervalls zwischen zwei Paketen (Jitter), der Zeitpunkt des Empfangs des letzten und das Zeitintervall zwischen zwei letzten SR-Paketen, übermittelt.

Ein wichtiger Aspekt ist hier der Begriff des Jitters (Interarrival Jitter). Dieser drückt die Schwankungen des Zeitintervalls zwischen den Ankunftszeiten der RTP-Pakete aus. Ist diese nicht konstant, kommt es zu Dekodierungs- und Synchronisationsproblemen beim Empfänger. Die Unterschiede können zum Beispiel durch Implementieren von Empfangs-Puffern ausgeglichen werden. Der Jitter wird folgendermaßen berechnet:

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

$$J(i) = J(i-1) + \frac{|D(i-1,i)| - J(i-1)}{16}$$

Wenn i und j RTP-Pakete sind, dann sind $R_i$ und $R_j$ ihre Ankunftszeiten, $S_i$ und $S_j$ ihre Zeitstempel.

Der Empfängerbericht ist im Aufbau dem Senderbericht ähnlich. Er wird in Abbildung 5 dargestellt.

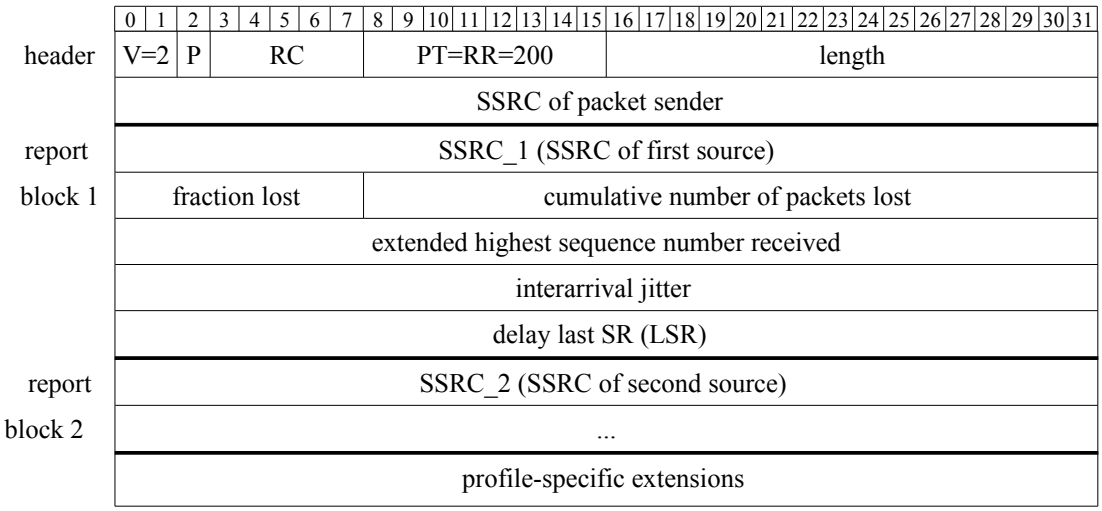


Abbildung 5: RTCP Receiver Report, Quelle: [RFC3550]

## 2.6 Der MPEG-4 Visual Standard

MPEG-4 Visual setzt sich aus einem Kern-Video-Kodierer-Dekodierer, der die Basisfunktionen umfasst, mehreren zusätzlichen Tools, die diese erweitern und Objekten zusammen. Applikationen nutzen üblich nur einen Teil der verfügbaren Werkzeuge. Deswegen werden für sie empfohlene Gruppen von Tools, in die so genannten Profile zusammengefügt. Die bekanntesten und an der Stelle am meisten relevanten Profile sind:

- „Simple“ stellt die minimale Menge der Werkzeuge für einfache Applikationen dar.
- „Core und Main“ wird für Kodierung von vielfachen beliebig gestalteten Video Objekten verwendet.
- „Advanced Real Time Simple“ bietet geringe Fehlerraten und Verzögerungen beim Transport an.
- „Advanced Simple“ ermöglicht bessere Qualität auf Kosten von größerer Komplexität.

Profile unterstützen die Zusammenarbeit von Codecs verschiedener Hersteller.

Ähnlich wie Profile Untermengen von Tools bilden, definieren Levels Einschränkungen in den Parametern des Videobitstroms (Tabelle 1):

| Profile | Level | Typical Resolution | Max. bitrate | Max. objects |
|---|---|---|---|---|
| Simple | L0 | 176 x 144 | 64 kbps | 1 simple |
| | L1 | 176 x 144 | 64 kbps | 4 simple |
| | L2 | 352 x 288 | 128 kbps | 4 simple |
| | L3 | | 384 kbps | 4 simple |
| Advanced Simple | L0 | 176 x 144 | 128 kbps | 1 AS or simple |
| (AS) | L1 | 176 x 144 | 128 kbps | 4 AS or simple |
| | L2 | 352 x 288 | 384 kbps | 4 AS or simple |
| | L3 | 352 x 288 | 768 kbps | 4 AS or simple |
| | L4 | 352 x 576 | 3 Mbps | 4 AS or simple |
| | L5 | 720 x 576 | 8 Mbps | 4 AS or simple |
| Advanced Real- | L1 | 176 x 144 | 64 kbps | 4 ARTS or simple |
| Time Simple | L2 | 352 x 288 | 128 kbps | 4 ARTS or simple |
| (ARTS) | L3 | 352 x 288 | 384 kbps | 4 ARTS or simple |
| | L4 | 352 x 288 | 2 Mbps | 16 ARTS or simple |

Tabelle 1: Level in MPEG-4 Visual. Quelle: [Richardson03]

Durch jedes Level wird die maximale, zur Dekodierung der MPEG-4-Videosequenz benötigte Leistung definiert. Die komplette Beschreibung des Bitstroms von dem zu dekodierenden Video ist zum Beispiel Simple Profile @ Level 0.

An dieser Stelle werden kurz Videoobjekte (video object VO) erläutert (Es gibt auch andere Objekte wie: Face and Body Animation Object, Mesh Object, 3D Mesh Object, die aber an der Stelle nicht relevant sind). MPEG-4 Visual erweitert das bisher eingesetzte Format des Videostroms als Folge von rechteckigen Videobildern um Objekte, die beliebige Formen annehmen können. Diese Darstellung ermöglicht dem Endbenutzer das Verändern der Video-Szene durch Ausschneiden, Hinzufügen oder Verschieben der Objekte. Sie existieren eine beliebige Zeit lang. Eine VO-Instanz zu einer bestimmten Zeit heißt Video Object Plane (VOP). VOPs können in einfachen Anwendungen auch die ganze Bildfläche umfassen, was ein zu den bisherigen Kompressionsverfahren äquivalentes Format ist [Richardson03]. Die beiden VOP-Arten sind in Abbildung 6 dargestellt.

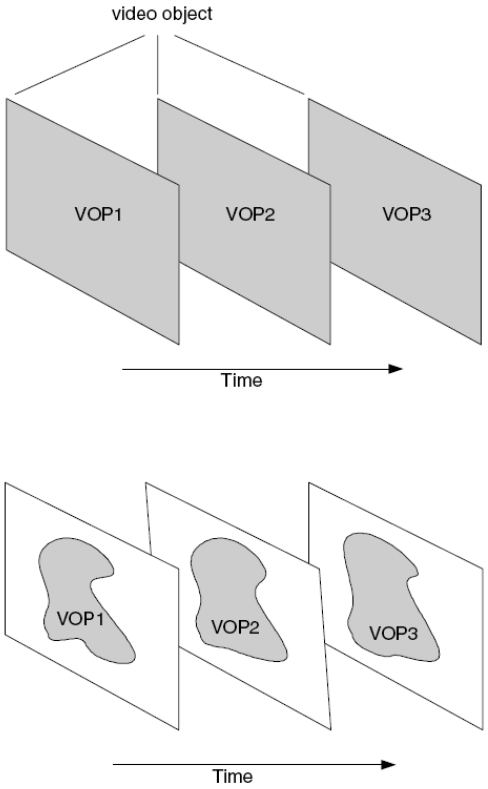


Abbildung 6: Video Object Planes. Quelle [Richardson03]

### 2.6.1 Visueller Bitstrom, Syntax und Semantik

Das Verständnis der binären Darstellung von kodierten Videodaten ist für Zwecke dieser Diplomarbeit von größter Bedeutung.

Kodierte Videodaten bilden in MPEG-4 Visual eine geordnete Zusammenstellung von Video-Bitströmen die „Layer“ (Schichten) genannt werden. Wenn es nur einen Layer gibt, dann spricht man von einem nicht skalierbaren, andernfalls von einem skalierbaren Bitstrom. Einer der Layer ist ein Basis-Layer. Dieser kann unabhängig dekodiert werden. Die anderen heißen Erweiterte-Layer (enhancement layers) und können zusammen mit den darunter liegenden Schichten dekodiert werden [MPEG-4_2]. Die logische Struktur des Videodatenstroms ist in Abbildung 7 dargestellt.

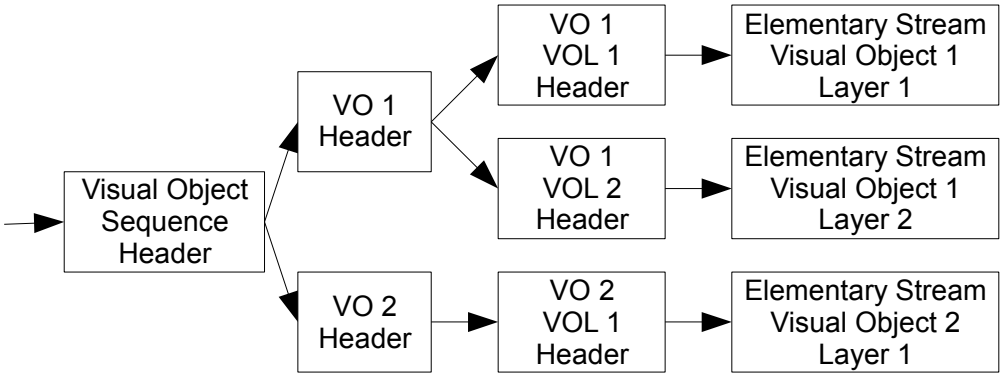


Abbildung 7: Videodatenstrom logisch strukturiert. Quelle: [MPEG-4_2]

In einem Videodatenstrom können Konfigurationsinformationen mit Elementarströmen kombiniert werden, wie in Abbildung 8 veranschaulicht.

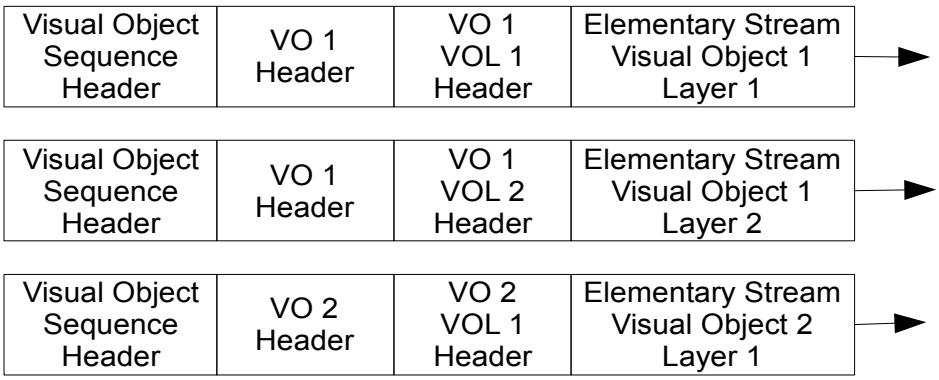


Abbildung 8: Konfigurationsinformationen kombiniert mit Elementarströmen, Quelle: [MPEG-4_2]

Konfigurationsinformationen und Elementarströme können aber auch separat übertragen werden (Abbildung 9).

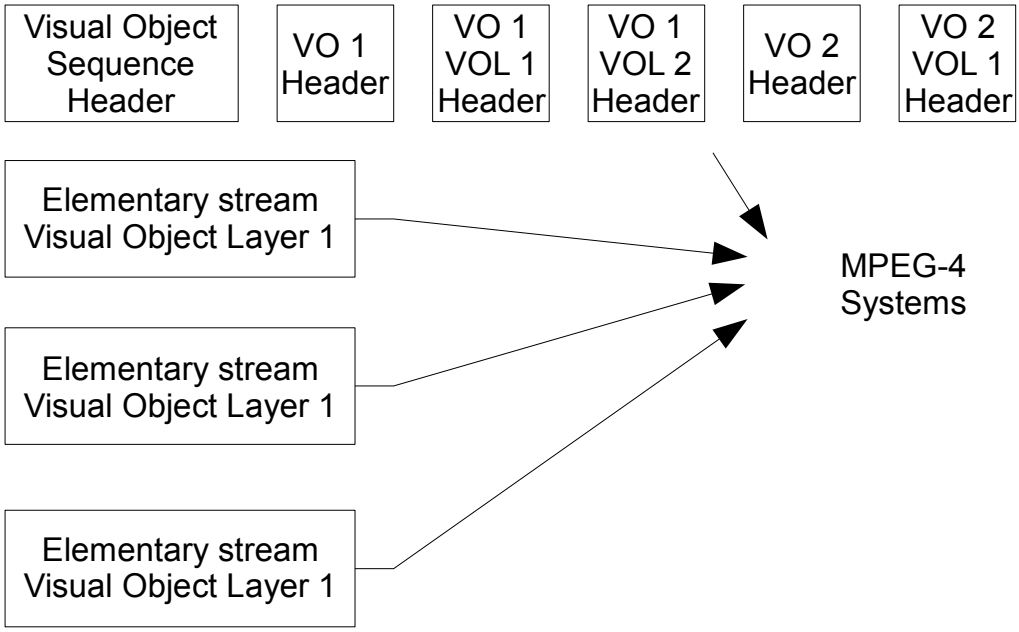


Abbildung 9: MPEG-4 Visual Separate Konfigurationsinformationen und Elementarströme

Die Syntax des Bitstroms besteht aus zwei Typen von Informationen:

- Konfigurationsinformationen; diese beinhalten:
  - Globale Informationen, die sich auf alle gleichzeitig von einem Decoder zu verarbeitenden Visuellen Objekten beziehen (VisualObjectSequence()),
  - Objekt-bezogene Informationen, die sich auf ein Visual Object beziehen (VisualObject()),
  - Objekt-Layer-bezogene Konfigurationsinformationen (VideoObjectLayer()).
- Elementarstrom mit Daten eines Single Layer und folgenden Einstiegsfunktionen:
  - Group_of_VideoObjectPlane()
  - VideoObjectPlane()
  - video_plane_with_short_header()
  - MeshObject()
  - fba_object()

Startcodes sind spezifische Bitmuster, die nur an bestimmten Stellen auftreten dürfen. Mit ihrer Hilfe wird der Anfang eines neuen Inhalts markiert. Jeder Startcode wird von einer bestimmten Bitfolge eingeleitet: '0000 0000 0000 0000 0000 0001'. Er wird durch eine acht Bit Integer-Zahl dargestellt (Tabelle 2):

| name | start code value (hexadecimal) |
|---|---|
| video_object_start_code | 00 through 1F |
| video_object_layer_start_code | 20 through 2F |
| reserved | 30 through AF |
| visual_object_sequence_start_code | B0 |
| visual_object_sequence_end_code | B1 |
| user_data_start_code | B2 |
| group_of_vop_start_code | B3 |
| video_session_error_code | B4 |
| visual_object_start_code | B5 |
| vop_start_code | B6 |
| reserved | B7-B9 |
| fba_object_start_code | BA |
| fba_object_plane_start_code | BB |
| mesh_object_start_code | BC |
| mesh_object_plane_start_code | BD |
| still_texture_object_start_code | BE |
| texture_spatial_layer_start_code | BF |
| texture_snr_layer_start_code | C0 |
| texture_tile_start_code | C1 |
| texture_shape_layer_start_code | C2 |
| stuffing_start_code | C3 |
| reserved | C4-C5 |
| System start codes (see note) | C6 through FF |
| NOTE System start codes are defined in ISO/IEC 14496-1:1999 | |

Tabelle 2: MPEG-4 Visual Startcodes, Quelle: [MPEG-4_2]

### 2.6.2 Transport

MPEG-4 kann aus auf zwei Arten transportiert werden: über den MPEG-2-Transport-Strom oder über IP meistens in RTP Pakete verpackt. Bei der zweiten Art können in RTP Paketen, entweder die Audio- und Video-Daten direkt ohne die

Verwendung von MPEG-4 Systems (RFC3016) oder die FlexMux-Daten (RFC3640) enthalten sein.

Wie oben schon erwähnt wird in RFC3016 eine Empfehlung zum RTP-Nutzdaten-Format für MPEG-4 Video- und Audio-Daten ohne die Benutzung von in MPEG-4-Part 1 beschriebenen Elementardatenströmen dargestellt. Die Daten werden direkt auf die RTP-Pakete abgebildet. Es entfallen dementsprechend die Synchronisations- und Steuerungs-Mechanismen des MPEG-4-Systems [MPEG-4_1]. Dieses Format kommt in Systemen mit eigener Datenstromverwaltung zur Verwendung. Von Vorteil ist es an der Stelle, dass er auf gleiche Weise handelt wie bei nicht im MPEG-4-Standard definierten Formaten. Ein Nachteil stellt die fehlende Interoperabilität mit MPEG-4-Systems benutzenden Systemen dar.

Es wird empfohlen ein VOP in ein RTP-Paket zu verpacken. Das Verpacken mehrerer VOPs in einem oder das Aufteilen eines VOP in mehrere RTP-Pakete sollte nur in Ausnahmefällen erfolgen (kleine VOPs können entstehen, wenn sie nicht ganze Frames, sondern kleine Objekte abbilden). Somit kann der RTP-Timestamp den Kodierungs-Zeitpunkt eines VOPs darstellen.

Der RTP-Timestamp ist also gleich dem Timestamp des in dem RTP-Paket enthaltenen VOPs. Wenn sich mehrere VOPs in dem Paket befinden, dann bezieht er sich auf den ersten VOP. Wenn das Paket nur Konfigurations-Informationen enthält, bezieht sich der Zeitstempel auf den ersten folgenden VOP. Die Auflösung des Timestamps ist standardmäßig 90kHz. Sie kann jedoch anders definiert werden, zum Beispiel durch entsprechende SDP- oder MIME-Einträge [RFC3016].

Die RTP-Nutzdaten können folgenden Aufbau haben (Abbildung 10):

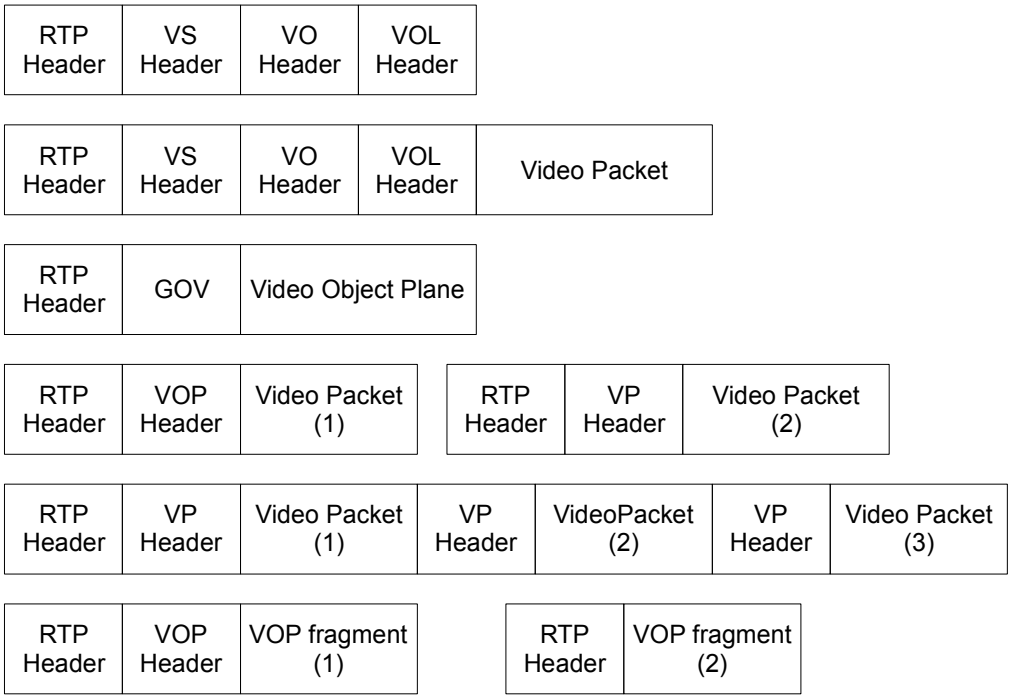


Abbildung 10: RTP Nutzdaten für MPEG-4 Visual, Quelle: [RFC3016]

Unzulässiger Aufbau des RTP-Payloads ist in Abbildung 11 dargestellt.

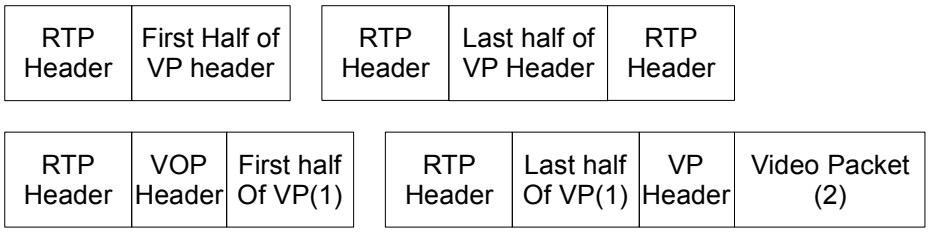


Abbildung 11: Unzulässige RTP Nutzdaten, Quelle: [RFC3016]

Einige Informationen über den Inhalt des Medienstroms werden in MIME[4]-Parameter verpackt. Im Folgenden werden die wichtigsten MIME-Parameter für MPEG-4 Video-Datenstrom beschrieben:

MIME-Media-Typ-Name: video

MIME-Subtyp-Name: MP4V-ES

Erforderliche Parameter: none

4 MIME: Multipurpose Internet Mail Extensions, ein Kodierstandard zur Festlegung der Struktur verschiedener Internetnachrichten [RFC2045]

Optionale Parameter:

- „rate“ stellt die Auflösung des RTP-Timestamps dar.
- „profile-level-id“ ist die dezimale Darstellung des MPEG-4-Profils und Levels.
- „config“ beinhaltet die Konfiguration des MPEG-4 Video-Datenstroms im gleichen Format wie die eigentlichen Videodaten.

Die MIME-Parameter werden auf entsprechenden Stellen im Session Description Protokoll (SDP [RFC2327]) eingetragen:

- MIME-Media-Typ-Name ist dann im SDP-Parameter „m=“,
- MIME-Subtyp-Name im „m=rtpmap“,
- der optionale Parameter „rate“ auch im „m=rtpmap“,
- der Parameter „profile-level-id“ und „config“ in dem Parameter „a=fmtp“ zu finden

Ein Beispiel für Simple Profile, Level 1, rate=90000(90kHz):

```
m=video 49170/2 RTP/AVP 98
a=rtpmap:98MP4V-ES/90000
a=fmtp:98 profile-level-id=1;config=000001B001000001B509
                    000001000000120008440FA282C2090A21F
```

## 2.7 Der H.264 Standard

Der H.264 Standard, auch Advanced Video Coding (AVC) genannt, ermöglicht bessere Qualität und höhere Kompressionsraten als frühere MPEG-4 und H.261 oder H.263 Standards. Er ist ein Teil des MPEG-4 Standards (14496 Part 10). Weil er gemeinsam mit der ITU-T[5] entwickelt wurde, ist er auch als H.264 Standard definiert.

Beim Kodieren des Videos erhält jedes Videobild eine eindeutige Nummer (picture order count), die die Reihenfolge, in der die weiteren Bilder dekodiert werden,

5 ITU: International Telecommunication Union, die Internationale Fernmeldeunion

definiert. Zudem wird eine weitere eindeutige Nummer (frame number) eingeführt, die aber nicht unbedingt die Dekodier-Reihenfolge darstellt. Referenz-Bilder können für die so genannte „inter prediction“[6] genutzt werden. Sie sind in zwei Listen (0 und 1) angeordnet.

Ein kodiertes Videobild besteht aus Makroblöcken (macroblocks). Ein Makroblock aus 16x16 „luma samples“[7] und dazu gehörigen 8x8 „chroma samples“[8] (Cb und Cr). Mehrere Makroblöcke werden zu Teilen (slices) zusammengefügt, die vom Typ I, P, B, SI oder SP sein können. Eine I-slice kann nur I-Makroblöcke enthalten, eine P-slice I- und P- und eine B-slice I- und B-Makroblöcke.

In AVC werden drei Profile mit einer bestimmten Menge an Funktionen, die vom Encoder und Decoder unterstützt werden, definiert (Abbildung 12). Baseline Profile wird meistens für Videotelefonie und Videokonferenzen verwendet. Für TV-Broadcast und -Speichern ist der „Main Profile“ die beste Wahl. Der Extended Profile ermöglicht effizientes Umschalten zwischen kodierten Bitströmen, Partitionierung der Daten und bessere Ausfallsicherheit. Er eignet sich deshalb besonders gut für das Streaming multimedialer Inhalte [Richardson03].

6 Nicht bewegte Teile eines Videobildes können vorhergesagt werden, da sie höchstwahrscheinlich gleich aussehend geblieben sind

7 Momentaner Wert der Helligkeit

8 Die Farbinformation.

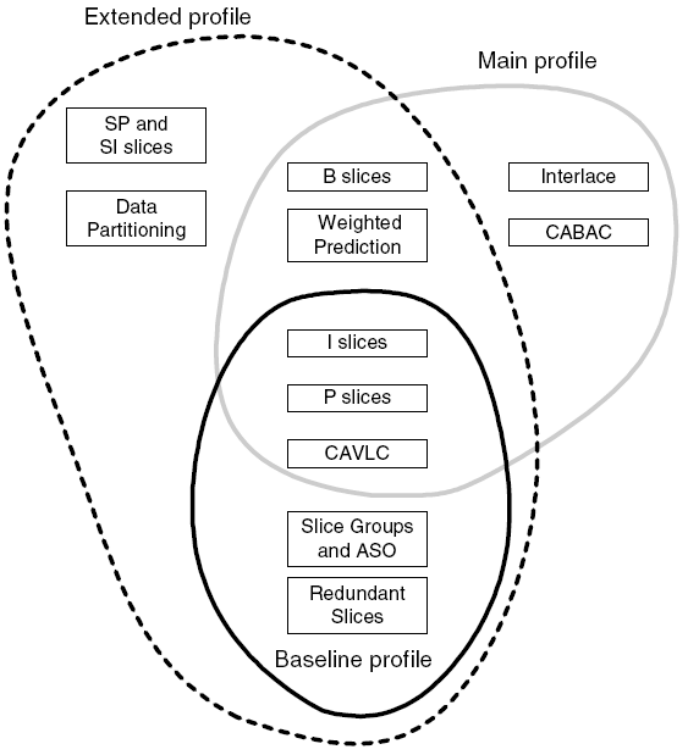


Abbildung 12: Profile in H.264, Quelle: [Richardson03]

### 2.7.1 Datenstruktur

H.264 unterscheidet zwischen den kodierten Videodaten (Video Coding Layer, VCL) und den Transportdaten (Network Abstraction Layer, NAL). Die VCL-Daten sind das Ergebnis der Videokodierung, die in NAL-Einheiten (NAL units, NALU) abgebildet werden. Die NALUs können entweder übertragen oder gespeichert werden. Jede beinhaltet ein „Raw Byte Sequence Payload“ (RBSP) mit kodierten Videodaten und Header-Informationen. Abbildung 13 zeigt eine Beispiel-RBSP-Sequenz an.

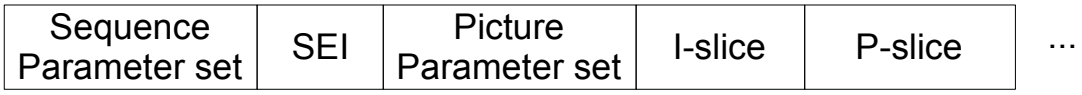


Abbildung 13: Sequenz von RBSP, Quelle: [Richardson03]

Jede RBSP-Einheit wird in einer separaten NAL-unit gespeichert. Der Aufbau der

Daten nimmt eine Gestalt an, die in Abbildung 14 dargestellt ist.

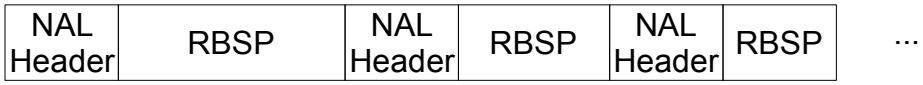


Abbildung 14: Aufbau von NAL-Einheiten. Quelle [Richardson03]

Der 8-Bit-lange NAL-Header (Abbildung 15) besteht aus:

- „forbidden_zero_bit“ gibt an, ob Fehler oder Syntax-Abweichungen im Header oder in Nutzdaten zu erwarten sind (sollte immer den Wert 0 besitzen).
- Die NRI-Nummer (nal_ref_idc) hat folgende Bedeutungen:
  - ein Wert ungleich 00 signalisiert, dass der Inhalt der NALU nicht zum dekodieren von Referenzbildern benötigt wird. Die „units“ können verworfen werden, ohne die Integrität des Referenzbildes zu gefährden. Die weiteren Werte kennzeichnen die relative Transport-Priorität.
  - Für „slices“ haben sie folgende Bedeutung:
    - Binär 10, für eine nicht-IDR slices oder slice-Daten in der Partition A
    - Binär 01, für slice-Daten in der Partition B oder C
- type (nal_unit_type), gibt den Inhalt der „NAL unit“ an


Abbildung 15: Der NAL-Header

In dem Type-Feld des Headers einer NAL-unit wird der Typ des RBSP-Inhalts angegeben [MPEG4_10]:

- „Sequence Parameter Set“ (type: 7) enthält globale Parameter für eine Sequenz kodierter Videobilder wie Bildgröße, Videoformat, Anordnung der Makroblöcke.
- „Picture Parameter Set“ (type: 8) bezieht sich auf einzelne Bilder.
- „Supplemental Enhancement Information (SEI)“ (Type: 6) beinhaltet

zusätzliche Informationen.

- „Picture Delimiter“ (type: 9) ist ein optionaler Parameter, eine Abgrenzung zwischen zwei Videobildern.
- „Coded slice Data Partition A, B, C“ hat den Type-Wert entsprechend: 2, 3 oder 4. Eine slice wird in drei verschiedenen Datenpartitionen platziert. Jede beinhaltet einen Teil der kodierten slice. Partition A enthält den Header der slice und Header von allen in ihr enthaltenen Makroblöcke. Partition B enthält alle intra-kodierten und die Partition C alle inter-kodierten makroblocks.
- „End of sequence“ (type: 10) signalisiert, dass das nächste Bild ein IDR-Bild ist.
- „End of Stream“ (type: 11) signalisiert, dass es in diesem Bitstrom keine Videobilder mehr gibt.
- „Filler data“ (type: 12) ist ein Dummy.

### 2.7.2 Transport

Die RFC[9]-Empfehlung für das RTP-Nutzdatenformat der H.264/AVC-Videoinhalte trägt die Identifikationsnummer 3984. Danach werden die RTP-Pakete mit einem oder mehreren NALUs gefüllt. Das RTP-Nutzdatenformat hat eine breite Einsetzbarkeit. Für Anwendungen mit geringer Bitrate, wie z.B. Videotelefonie, ist er genauso gut geeignet wie für „video-on-demand“ mit hohen Bitraten.
Es gibt drei grundlegende Datenstrukturen, die in diesem Dokument definiert werden. Der Empfänger kann sie anhand des darin enthaltenen ersten Byte (ersten NAL-Haeder) erkennen:

- „Single NAL Unit Packet“ (Abbildung 16) enthält nur eine NALU als Nutzdaten. Der Type-Wert im NAL-Header ist dabei gleich dem originalen Type-Wert der NALU (1-23).
- „Aggregation Packet“ ist eine Agregation von mehreren NALUs in einem RTP-Paket. Davon existieren vier Versionen (die type-Werte sind entsprechend 24, 25, 26 und 27):
  - „Single-Time Aggregation Packet type A“ (STAP-A) beinhaltet Pakete

9 Request For Comments (auf deutsch: Bitte um Kommentare). RFCs sind eine Serie von Dokumenten, Empfehlungen und Standards im Bereich Internet.

mit derselben NALU-Zeit (Abbildung 17).

- „Single-Time Aggregation Packet type B“ (STAP-B), wie oben, enthält aber zusätzlich die 16-Bit lange DON (Decoding Order Number, 0-65535). Sie definiert die Dekodier-Reihenfolge
- „Multi-Time Aggregation Packet with 16-bit offset“ (MTAP-16) vereinigt NAL-units mit verschiedenen NALU-Zeiten. Die DON wird dabei in zwei Werte zerlegt, in die 16 Bit lange DONB (Decoding Order Number Base) und die 8 Bit lange DOND (Decoding Order Number Difference), zusätzlich gibt es ein 16 Bit langes Zeitstempel-Offset zum RTP-Timestamp (TS Offset) (Abbildung 18).
- „Multi-Time Aggregation Packet with 24-bit offset“ (MTAP-24), wie oben, aber mit längerem Zeitstempel-Offset

■ „Fragmentation unit“ definiert die Art der Fragmentierung eines NAL in mehrere Einheiten (Abbildung 19). Es gibt zwei Versionen: FU-A und FU-B. Die NAL-Type-Werte sind 28 und 29. Der FU Indicator hat dieselbe Funktion wie die des STAP-Pakets. Neu ist hier der ebenso 1 Byte lange FU Header, der sich von dem FU Indicator darin unterscheidet, dass die ersten drei Bits (S, E, R) einzeln interpretiert werden. S steht für den Teil der NALU, E für den letzten und R für Reserved [RFC3984].

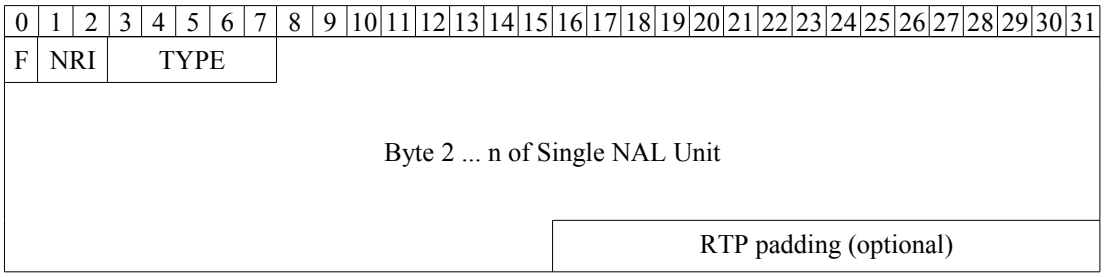


Abbildung 16: Single NAL-unit, Quelle: [RFC3984]

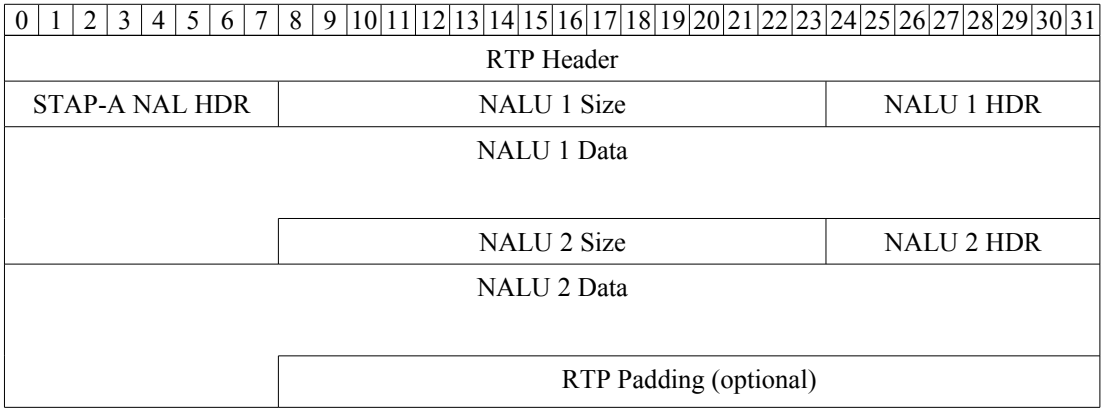


Abbildung 17: STAP-A Paketformat, Quelle: [RFC3984]

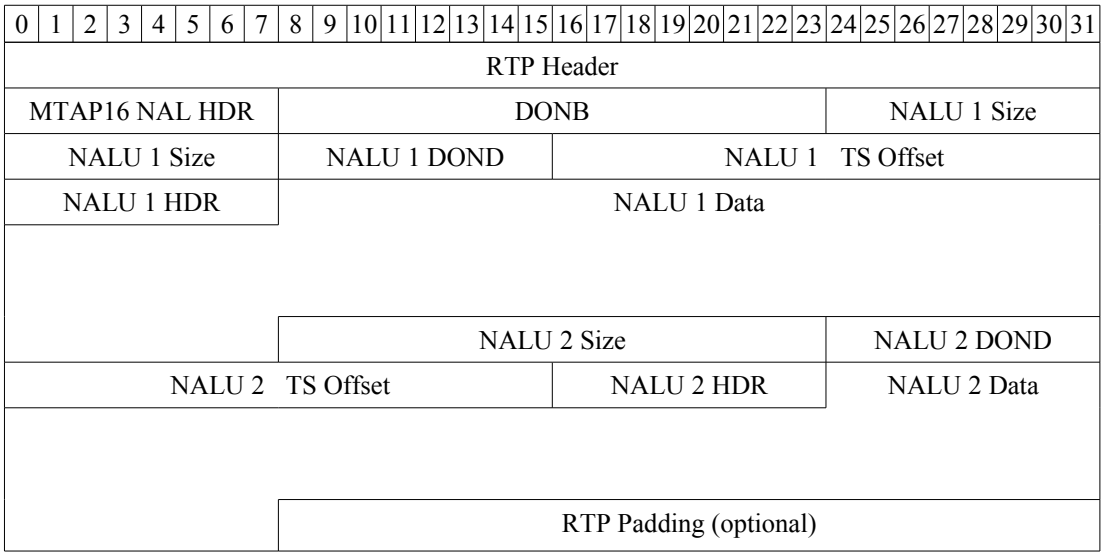


Abbildung 18: MTAP16 Paketformat, Quelle: [RFC3984]

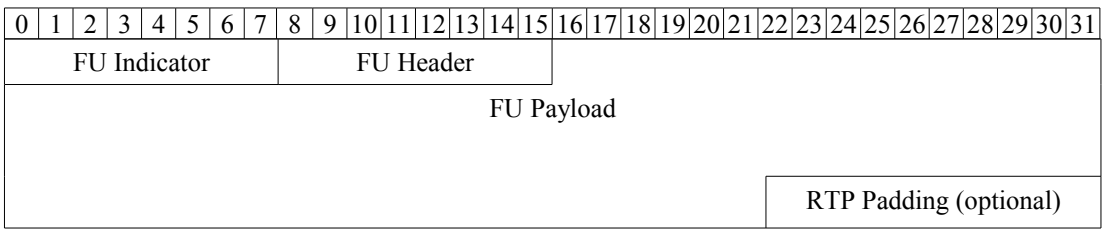


Abbildung 19: FU-A Paketformat, Quelle: [RFC3984]

Die Eigenschaften des Mediums werden als MIME-Parameter übermittelt. Die MIME-Parameter für H.264-Video sind:

Media-Typ-Name: video
Media-Subtyp-Name: H264
Erforderliche Parameter: keine
Optionale Parameter:

- „profile-level-id“ ist eine hexadezimale base16-Representation der folgenden, in H.264/AVC definierten Parameter: „profile_idc“, „profile-iop“ und „level_idc“. Dieses Parameter beschreibt die Eigenschaften des Datenstroms.
- „max-mbps“ gibt die maximale Verarbeitungsgeschwindigkeit von Makroblöcken gemessen in Mackroblöcken pro Sekunde an.
- „max-fs“ (maximum frame size) gibt die maximale Größe des Bildes als Anzahl der darin enthaltenen „macroblocks“ an.
- „max-cpb“ (maximum coded picture buffer size)
- „max-dpb“ (maximum decoded picture buffer size), die Einheit: 1024 Byte
- max-br“ (maximum video bit rate), die Einheit: 1000 Bit pro Sekunde
- „packetization-mode“ bedeutet „single NAL unit mode“, wenn der Wert 0 beträgt oder wenn er nicht gesetzt ist, „non-interleaved mode“, wenn der Wert 1 ist und „Interleaved mode“, wenn er mit einer 2 besetzt ist.
- Weitere Parameter sind: „redundant-pic-cap, sprop-parameter-sets, parameter-add, sprop-interleaving-depth, sprop-deint-buf-req, deint-buf-cap, sprop-init-buf-time, sprop-max-don-diff, max-rcmd-nalu-size“.

Die MIME-Parameter werden in die „a=fmtp:“ Zeile der SDP-Antwort auf die DESCRIBE-Anfrage des Streaming-Servers eingetragen [RFC3984].

## 2.8 Vergleich von MPEG-4 Visual und AVC

Das erfolgreichste Kompressionsverfahren der letzten 10 Jahre war MPEG-2, das in DVD-Video und digitalem Fernsehen eingesetzt wurde und wird. Es ist allerdings nicht mehr ganz aktuell. Es gibt zwei neuere, effizientere und innovative Verfahren, wie das MPEG-4 Visual und das H.264. Sie werden von der ITU (International

Telecommunications Union) und der MPEG (Moving Picture Experts Group) die Normen für die ISO (International Standards Organisation) entwickelt, standardisiert. MPEG-4 Visual ist in [MPEG-4_2] und MPEG-4 AVC (H.264) in [MPEG-4_10] beschrieben.

MPEG-4 Visual unterstützt eine Vielzahl an möglichen Videoobjekten. Er definiert auch 19 verschiedene Profile. Dieser Standard bietet aber lediglich eine mittlere Kodiereffizienz und eine minimale Bewegungskompensationsblockgröße[10] von 8x8 Bildpunkten. Die Genauigkeit des Bewegungsvektors von einem halben oder einem viertel des Bildpunktes.

H.264 (AVC) bietet die Möglichkeit ausschließlich rechteckige Einzelbider und Bildteile zu bilden. Es werden auch nur 3 Profile definiert. Dafür ist die Kodiereffizienz höher als bei MPEG-4 Visual. Die Bewegungskompensationsblockgröße beträgt 4x4 Bildpunkte und die Genauigkeit des Bewegungsvektors beträgt einen viertel Pixel.

## 2.9 Aufbau einer MP4-Datei

Da in dieser Arbeit Video-Transportstrom zentral ist, wird das Format einer lokalen Datei nur nebenbei erwähnt. Hier folgt eine ansatzweise Andeutung des Aufbaus eines MP4-Containers.
Eine Datei setzt sich aus einer Serie von Objekten, die sich „Boxes“ nennen, zusammen. Die Daten werden ausschließlich in diesen Objekten gespeichert, die wiederum ineinander verschachtelt werden können. So ist die größte Box die „moov“-Box (movie), die einige „trak“s (track) enthalten kann [MPEG-4_12].

Für diese Arbeit sind die so genannten „hint-tracks“ interessant. Sie beschreiben, wie die in der Datei gespeicherten, elementaren Datenströme in Streaming-Protokolle verpackt werden können. Für jedes Protokoll existiert ein „hint-track“. Diese Tracks werden dann in der Datei, neben den Video- und Audio-Spuren gespeichert. Dieser

10 Bewegungskompensationsblock (Motion compensation block) ist ein Teil des Videobildes, der mit dem nächsten Bild verglichen wird. Liegt kein Unterschied vor, dann wird nur ein Verschiebungsvektor gespeichert.

Aufbau ermöglicht, dass eine und dieselbe Datei sowohl lokal (die „hint tracks“ werden dann einfach ignoriert), als auch über das Netzwerk wiedergegeben werden kann. Der Server sucht vor dem Streamen der Datei nach allen darin enthaltenen „hint-tracks“ und wählt anschließend die passendsten aus. Es kann für ein Elementarstrom mehrere „hint tracks“ für unterschiedliche Streaming-Protokolle geben.

Die „hint tracks“ erstellen den zu streamenden Datenstrom durch Holen der Daten aus anderen Tracks. Diese können wiederum „hint tracks“ oder Elementarstrom-Tracks sein. Das geschieht mit Hilfe der in den „hint tracks“ gespeicherten Verweise auf die Elementarstrom-Daten. Diese bestehen aus vier Informationen: der Nummer des Tracks, auf den Verwiesen wird, der Nummer des Samples, einem Offset und der Länge [MPEG-4_12].

„Hint tracks“ können, wenn sie in der Media-Datei nicht vorhanden sind, zu dieser hinzugefügt werden, zum Beispiel mit Hilfe des Werkzeugs mp4creator, das ein Teil der MPEG4IP Anwendung ist. Ob sie vorhanden sind, kann mit dem Tool mp4info festgestellt werden (vgl. Kap. 3.1.2).

## 2.10 Anwendungen für mobile Endgeräte schreiben

### 2.10.1 Anforderungen an Programmiersprachen an mobilen Endgeräten

Die meisten Anwendungen für mobile Endgeräte sind in C/C++ oder in Java geschrieben. C wird nur in Ausnahmefällen als Programmiersprache genutzt, da sie die Objektorientierung nicht unterstützt.

Java ist aus folgenden Gründen vorteilhafter als ihr Konkurrent C++:

- Portabilität: Java Bytecode ist plattformunabhängig und funktioniert auf jedem Gerät mit installierter Java Virtual Maschine (JVM), die praktisch auf jedem mobilen Endgerät zu finden ist.
- Sicherheit: In Java gibt es ein API für sicherheitsrelevante Abläufe, wie z.B. für Authentifizierung, die Überprüfung von Signaturen und die Vergabe von Rechten.
- Akzeptanz: Java ist sehr weit verbreitet und hat eine große Community.
- Robustheit: Java Anwendungen werden in einer „Sandbox“ ausgeführt. Die Anwendung wird von der JVM überwacht, damit der eventuelle Absturz keine weiteren Programme stört. Desweiteren ist der Programmcode vor der Ausführung mit einem Prüfsummenverfahren verifiziert worden, um die Integrität des Bytecodes sicherzustellen. Seine eventuelle Veränderung, z.B. durch ein Virus, wird nahezu unmöglich.
- Garbage Collector: Java hat einen Garbage Collector, der mögliche Speicherlecks verhindert. In C++ muss der Programmierer sich selber darum kümmern [Breymann08].

### 2.10.2 Einschränkungen

Mobile Endgeräte unterliegen aufgrund ihrer kleinen Ausmaße bestimmten Einschränkungen:

- Die Batterietechnologie hat im Vergleich zu der Speicher- und CPU-Entwicklung geringere Fortschritte gemacht. Energie ist wegen der kleinen Größe der Geräte nur eingeschränkt verfügbar.
- Der Speicher ist aus Platz-, Gewichts-, und Energiespargründen begrenzt.

- Die Auflösung der Anzeige ist gering.
- Die Bedienelemente sind klein und müssen mit nur einer Hand bedienbar sein.
- Die Fehlerrate bei Netzwerk- und Funkzellenverbindung ist groß, da ein mobiles Endgerät beweglich ist und sich dadurch die Umgebung und damit die Verbindungsqualität oft ändern kann [Breymann08].

## 2.11 Java Micro Edition (JavaME)

Die Java Micro Edition ist nicht umfangreich. Trotzdem wirkt sie häufig als die am schwierigsten zu durchschauende Edition. Die Erfahrungen aus der JavaSE können nur in seltenen Fällen wiederverwendet werden. Die Netzwerkprogrammierung, die Gestaltung von Bedienoberflächen wird in der JavaME anders gehandhabt. Des weiterem fehlt eine einheitliche Spezifikation.

Die Java Micro Edition ist speziell für den Einsatz in kleinen Endgeräten ausgelegt. Es ist die „kleine Schwester“ der zwei Java-Editionen: der JavaEE, die für Server und Workstations konzipiert ist und der JavaSE für PCs. Als Folge der Ressourcenbeschränkung ist der Funktionsumfang, gemessen an der Anzahl der Programmierschnittstellen in den Klassenbibliotheken, kleiner als bei den anderen zwei Editionen. Zusätzlich ist auch die Leistung der „Java Virtual Machine“ eingeschränkt. Dies liegt z.B. daran, dass der „Just-in-time-Compiler“ fehlt. Auch der Sprachumfang beinhaltet nur einen Teil des aus JavaSE Bekannten[Schmatz07].

### 2.11.1 Konfigurationen, Profile und optionale Pakete in JavaME

Die Java Micro Edition wurde für eine große Zahl verschiedener Geräte konzipiert, z.B. für TV set-top Boxen, PDAs, Pager, Mobiltelefone, etc. Diese Endgeräte unterscheiden sich z.B. in der Prozessorleistung, der Bildschirm- und Hauptspeichergröße. Aus diesem Grund wird JavaME in zwei Konfigurationen unterteilt, CLDC und CDC.

CLDC ist für Geräte mit einfacher Bedienoberfläche, einem kleinen Hauptspeicher, einer geringen, drahtloser Verbindungsgeschwindigkeit und eingeschränkten

Batterieleistung gedacht.

CDC ist für Geräte mit etwas komplexerer Bedienoberfläche, einem größeren, für die KVM bereitstehenden Hauptspeicher vorgesehen.

CLDC findet man demnach hauptsächlich in Mobiltelefonen, Pagern, schwächeren PDAs, CDC in TV set-top Boxen sowie stärkeren PDAs (Abbildung 20) [Fitzek07].

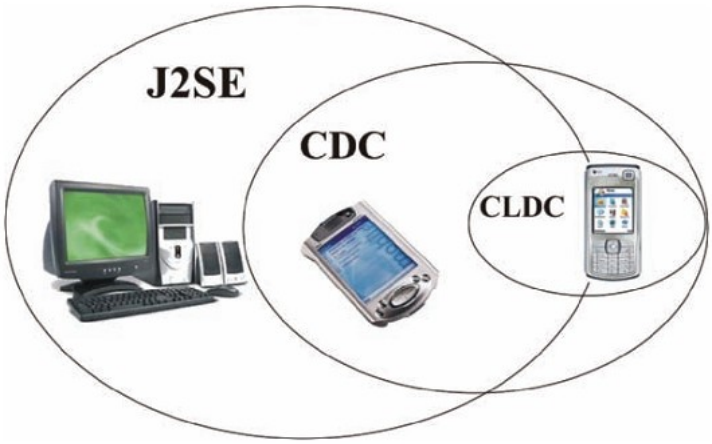


Abbildung 20: Vergleich zwischen den Konfigurationen, Quelle: [Fitzek07]

Die JavaME Platform setzt auf dem Betriebssystem des Endgeräts auf und besteht aus folgenden Bestandteilen (Abbildung 21):

- „Konfigurationen“ werden von den Basis-Bibliotheken (CDC oder „Connected Device Configuration“ und CLDC oder „Connected, Limited Device Configuration“) und der „Virtual Machine“ (KVM oder „Kilobyte Virtual Machine“) bei den leistungsschwächeren Geräten – eine abgespeckte JVM), gebildet. Die Basis-Bibliotheken sind meistens eine Teilmenge der JavaSE-API.
- „Profile“ ergänzen die Konfiguration um mehr „High-Level-APIs“. Sie beinhalten meistens: Bedienoberflächen, Speicherverwaltung, Netzwerkunterstützung. Das wichtigste Profil ist das „Mobile Information Device Profile“.
- „Optionale Pakete“ sind Klassenbibliotheken, die die Profile um bestimmte,

fortgeschrittene Funktionalitäten ergänzen. Es gibt mittlerweile eine große Zahl verschiedener Pakete, z.B. JSR135 „Mobile Media API“ MMAPI oder JSR82 „Java APIs for Bluetooth“ [Fitzek07].

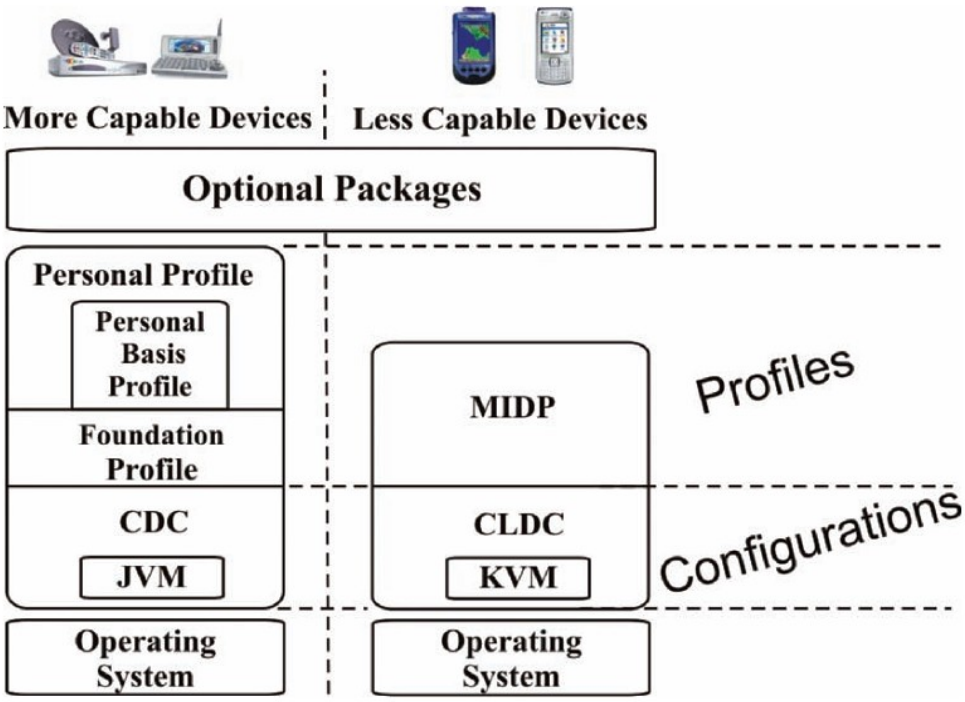


Abbildung 21: Profile der JavaME, Quelle: [Fitzek07]

Das wichtigste Profil für die CLDC ist, wie oben schon erwähnt, das MIDP. Es ist mittlerweile in nahezu jedem mobilen Telefon implementiert. Der Wichtigkeit wegen, wird hier die genaue Auflistung der in der MIDP-Klassenbibliothek vorhandenen Pakete dargestellt:

- `java.io` enthält Eingabe- und Ausgabestreams.
- `java.lang` enthält aus JavaSE bekannte Klassen. Sie sind fundamental in der Java-Programmiersprache.
- `java.util` stellt einige, auch aus JavaSE bekannte Klassen wie List, Array, Date, etc., zur Verfügung.
- `javax.microedition.io` ermöglicht die Netzwerkkommunikation.
- `javax.microedition.midlet` enthält nur die Klasse Midlet, die die Grundfunktionalität für Mobile Applikationen darstellt, darunter die Steuerung des Lebenszyklus eines MIDlets.

- `javax.microedition.lcdui` beinhaltet Bedienoberflächen.
- In `javax.microedition.rms` ist eine einfache Datenbankschnittstelle, das „Record-Management-System“, enthalten.

`java.io`, `java.lang`, `java.util` und `javax.microedition.io` sind auch schon in der CLDC vorhanden [MIDPAPI].

### 2.11.2 MIDlet-Suite

Die für das MIDP geschriebenen Anwendungen heißen MIDlets. Es sind Java-Klassen, die von der abstrakten Klasse MIDlet aus `javax.microedition.midlet` abgeleitet sind. Der Lebenszyklus eines MDLets wird von der so genannten AMS (Application Management Software) gesteuert. Ein MIDlet befindet sich zu jedem Zeitpunkt in einem definierten Zustand: gestartet, pausiert, beendet. Für jeden müssen entsprechende, in der Parent-Klasse MIDlet definierte Methoden implementiert werden. Der folgende, minimale Rahmen sieht dann so aus:

```
public class MIDletGrundrahmen extends MIDlet{
      public MIDletGrundrahmen(){ ... }
      public void startApp() throws
                  MIDletStateChangeException{ ... }
      public void pauseApp(){ ... }
      public void destroApp(boolean unconditional) throws
                        MIDletStateChangeException{ ... }
}
```

Die kleinste, installierbare Einheit heißt MIDlet-Suite und setzt sich aus einem oder mehreren MIDlets zusammen. In einer MIDlet-Suite besitzen alle MIDlets die gleichen Berechtigungen. Die von den MIDlets angelegten Datenbestände sind von allen anderen MIDlets lesbar, schreibbar und löschbar. Als statisch definierte Variablen sind innerhalb einer MIDlet-Suite, von allen Klassen sichtbar und änderbar.

Eine MIDlet-Suite wird dann zwecks der Installation, zu einem .jar-Archiv verpackt und setzt sich aus den präverifizierten .class-Dateien, weiteren Ressourcen (Icons,

Grafiken, etc.) und einem Manifest (Textdatei mit Metainformationen) zusammen [Schmatz].

### 2.11.3 Benutzungsschnittstellen

Einen weiteren, wichtigen Aspekt stellen die Benutzungschnittstellen (User-Interface) dar. An dieser Stelle wird die User-Interface-Bibliothek LCDUI (Liquid Crystal Display User Interface, Lowest Common Denominator User Interface oder Limited Connected Device User Interface) kurz beschrieben. Sie ist in dem Paket: `javax.microedition.lcdui` enthaltenen. Deren Funktionsumfang deckt die Möglichkeiten aller mobilen Endgeräte, unabhängig von der Bildschirmgröße oder der Tastenbelegung ab. Die LCDUI besteht aus zwei Teilen: dem High-Level-API und dem Low-Level-API. Das High-Level-API ist für Anwendungen, bei denen die Portabilität wichtig ist, die bessere Wahl. Das Low-Level-API bietet dafür ein sehr niedriges Abstraktionsniveau an und überlässt dem Programmierer die exakte Kontrolle über die Position und das Aussehen der Bedienelemente.

Der Portabilität wegen sind an der Stelle die Ansätze des High-Level-API von größerer Bedeutung. Folgend sind ein Paar wichtige Komponenten des APIs aufgelistet:

- Die softwaremäßige Darstellung des physischen Bildschirms stellt die Klasse Display dar. Die 1:1-Beziehung zwischen dem Display und dem MIDlet bleibt über den gesamten Programmablauf bestehen:

```
Display display = Display.getDisplay(this);
```

- Zum Eingeben und Editieren von Texten, kann die *TextBox* eingesetzt werden:

```
TextBox textBox = newTextBox("Test",null,100,TextField.ANY);
```

- Ein *Alert* bewirkt die Anzeige des Alert-Dialogs und anschließend des im zweiten Parameter übergebenen *Displayable:*

```
Alert alert = new Alert("Fehler");
display.setCurrent(alert,textBox);
```

- Ein Formular (Form) kann mehrere Elemente (Items) zusammenfassen und auf dem Bildschirm anzeigen:

```
Form form = new Form("Formular");
StringItem stringItem = new StrinItem("Alter","11");
form.append(stringItem);
display.setCurrent(strinItem);
```

Die wichtigsten Klassen der LCDUI sind im unten folgenden Klassendiagramm (Abbildung 22) dargestellt, wobei Screen und seine Unterklassen zu den High-Level-Komponenten gehören, Canvas gehört dagegen zu den Low-Level-Komponenten:

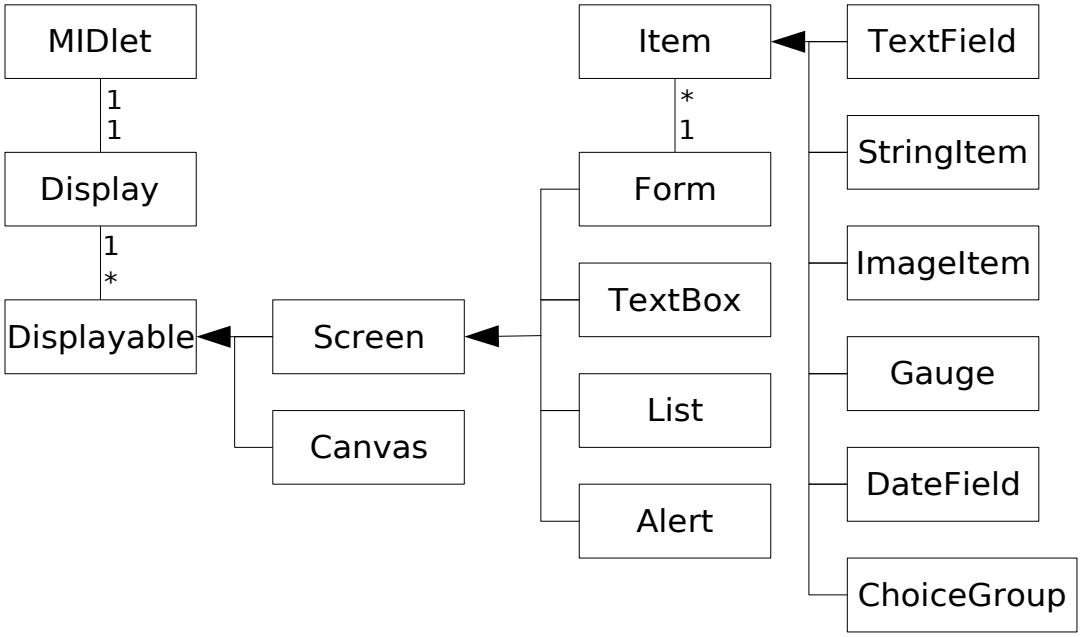


Abbildung 22: Benutzungsschnittstellen

Die Anwendung muss von dem Benutzer steuerbar werden. Dies geschieht dadurch, dass als Reaktionen auf Events, die von der Benutzerinteraktion ausgelöst wurden, eine entsprechende Callback-Methode eines zuvor definierten Listener-Objekts benachrichtigt wird. Die Events in der High-Level-Schnittstelle kann die Selektion

eines Kommandos für ein Displayable oder Item sein oder auch die Änderung des Zustands in einem Item.
Kommandos werden in LCDUI als Instanzen der Klasse Command repräsentiert und beinhalten lediglich die für die Darstellung in der Bedienoberfläche benötigten Informationen. Ein Command-Objekt kann folgende Attribute erhalten:

- Ein Label, das dem Benutzer die Identifikation des Kommandos ermöglicht.
- Ein Typ, der die Art von Aktion, die aus dem Kommando resultiert, ausdrückt. So könnten zum Beispiel die gleichen Kommandoarten nah beieinander platziert werden.
- Die Priorität. Kommandos mit hoher Priorität werden für den Benutzer leicht zugänglich platziert.

Ein Beispiel dafür ist folgend angeführt:

```
final TextBox textBox = new TextBox("Text:");
final Command cmdOK = new Command("OK", Command.OK, 1);
textBox.addCommand(cmdOK);
```

### 2.11.4 Low-Level-Netzwerkprogrammierung mit MIDP2

Für die Netzwerkkommunikation bietet JavaME das Generic Connection Framework an, das Verbindungen zu einem Kommunikationspartner durch das Interface `Connection` repräsentiert. Für jedes Protokoll gibt es ein von der `Connection` abgeleitetes Interface, das dessen Eigenschaften benutzbar macht.

Neben den Protokollen der Anwendungsschicht, wie HTTP und HTTPS, sind dank des Generic Connection Frameworks, auch die Protokolle der tiefer liegenden Schichten zur Netzwerkkommunikation im MIDlet anwendbar. Die wichtigsten an der Stelle sind:

- Das TCP-Protokoll und das dafür vorgesehene Interface `SocketConnection.`
- Das UDP-Protokoll mit der `DatagramConnection` und der `UDPDatagramConnection.`

Das Öffnen der Verbindung findet in der Klasse `Connector` statt. Die Klasse bietet

dazu folgende Methoden an, wobei `open()` eine bidirektionale Verbindung ermöglicht:

- `openInputStream()`
- `openDataInputStream()`
- `openOutputStream()`
- `openDataOutputStream()`
- `open()`

`openDateInputStream()` und `openDataOutputStream()` öffnen eine Verbindung zum Lesen und Schreiben von primitiven Java-Datentypen.

Zum Versenden von Daten wird bei einer TCP-Verbindung ein `InputStream()` und ein `OutputStream()` geöffnet. Die Daten können mit den Methoden `read()` und `write()` ausgetauscht werden.

Zum Austausch von Daten über eine UDP-Verbindung werden zunächst Datagramme gebildet:

```
DatagramConnection
dc=(DatagramConnection)Connector.open("datagram://:60000");
Datagram dg = dc.newDatagram(2048);
```

Mit `dg.setAddress()` kann die Empfängeradresse gesetzt werden und mit `dc.send(dg)` kann das Datagramm dann versendet werden. Durch den Aufruf der Methode `dc.receive(dg)` kann ein Paket empfangen werden. Sie verbleibt im wartenden Zustand, bis ein Datagramm ankommt [MIDPAPI].

### 2.11.5 Multimedia-Programierung mit der MMAPI

Das Mobile Media API (MMAPI, JSR135) ist ein optionales API, das erweiterte Multimedia-Unterstützung in Java-Programmen an mobilen Endgeräten bereitstellt.

In der MIDP ist bereits eine Untermenge des MMAPI, die einfache Audio-Unterstützung bietet, vorhanden. Sie erlaubt, Multimedia-Daten von verschiedenen Quellen, ohne tieferes Eindringen in die Übertragungsprotokolle zu behandeln. Das Verarbeiten der multimedialen Daten kann sehr komplex sein. MMAPI muss in der Lage sein die ankommenden Daten richtig zu interpretieren, sie zu dekodieren und zusätzlich Werkzeuge zur Manipulation des Mediums anbieten.

Die MMAPI trennt die oben genannten Aufgaben. Sie stellt zwei Objekte zur Verfügung die diese Aufgaben bearbeiten, die DataSource und den Player.

Die `DataSource` ist eine abstrakte Klasse, die als Aufgabe das Lokalisieren und Empfangen des Datenstroms hat. Der Player ist ein Interface, das für das Bearbeiten und Abspielen der Multimedia-Daten konzipiert wurde. Eine `DataSource` definiert einen oder mehrere Streams, die das Interface `SourceStream` implementieren. Ein `SourceStream` ist eine abstrakte Darstellung eines einzelnen Media-Streams.

Ein Player dient dem Abspielen und Steuern des Multimedia-Inhalts. Er bietet Methoden zum Wiedergeben, Steuern, Synchronisieren des Mediums mit anderen Playern und lauschen auf Ereignisse wie Starten, Stoppen oder Pausieren der Daten [Goyal06].

Einen Zugriff auf einen Player ermöglicht aber erst die `Manager`-Klasse. Sie ist eine Brücke zwischen der `DataSource` und dem entsprechenden `Player` (Abbildung 23).

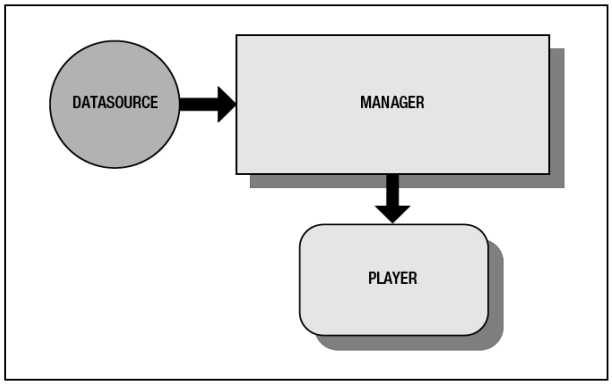


Abbildung 23: Das Erzeugen eines Players, Quelle: [Goyal06]

Ein Player kann eine DataSource selber erstellen, wenn das in der übergebenen URL angegebene Protokoll unterstützt wird. Ein einfacher Player kann folgendermaßen aufgebaut werden:

```
Player player = Manager.createPlayer("rtsp://192.168.1.1:554/test");
player.start();
```

Zum Erstellen einer eigenen DataSource muss die Klasse DataSource überschrieben werden:

```
public class StreamingDataSource extends DataSource{...}
```

Das Erzeugen eines Players sieht dann folgendermaßen aus:

```
StreamingDataSource sds = new StreamingDataSource();
Player player = Manager.createPlayer(sds);
```

# 3. Entwicklung

## 3.1 Vorbereitungen

### 3.1.1 Streaming Server

Zwei bekannte Streaming Server sind der VLC-Server und der Darwin Streaming Server.

Der VLC eignet sich sowohl zur Medien-Wiedergabe, als auch zum Streamen und Transkodieren von multimedialen Inhalten. Video on Demand gehört jedoch nicht zu seinen Stärken. Das VoD-Streamen auf mobile Endgeräte funktioniert nicht [VLC].

Der Darwin Streaming Server ist die Open Source Version von Apples QuickTime Streaming Server und ist in Versionen für Linux, Solaris und Windows verfügbar. Er kann Live- und VoD-Daten übertragen [DSS].

Die Konfiguration ist sehr einfach und kann über die Weboberfäche unter `http://<serverIP>:1220` getätigt werden. Für das Video-on-Demand-Streaming von z.B. MPEG-4-Medien reicht es die zu streamenden Dateien in den Ordner `/usr/local/movies` (Linux) zu kopieren. Diese können dann einfach mit `rtsp://<serverIP>:554/<datei.mp4>` angefordert werden.

### 3.1.2 Nützliche Werkzeuge

Die Videodateien können nicht in ihrem normalen Zustand gestreamt werden. Es müssen noch hint-Tracks hinzugefügt werden. Es gibt einige nützliche Werkzeuge, mit denen man Videodateien einfach manipulieren kann. Man kann mit ihnen z.B. einzelne Tracks hinzufügen oder entfernen, die Inhalte transkodieren oder detaillierte Informationen über das Medium anzeigen.

MPEG4IP

In Softwarepaket MPEG4IP werden einige Anwendungen als eine ultimative Lösung zum Streamen von multimedialen Inhalten bereitgestellt. MPEG4IP ist für das Debian Betriebssystem als Pakete: `mpeg4ip-utils` und `mpeg4ip-server` verfügbar. In dem Paket `mpeg4ip-utils` werden unter Anderem Werkzeuge zum Verarbeiten von Datei-Inhalten bereitgestellt: `mp4creator` ermöglicht das Extrahieren von MPEG-4-Elementardatenströmen und das Hinzufügen von hint-tracks. `mp4info` und `mp4videoinfo` liefern Informationen über die in einer Datei enthaltenen Tracks, `mp4dump` zeigt Dateistruktur an und `mp4trackdump` ermöglicht es den Aufbau eines Tracks zu Gesicht zu bekommen. MPEG4IP wird allerdings seit 2007 nicht mehr weiterentwickelt.

GPAC

Was GPAC ist, wird durch diesen Satz, der auf der ersten Seite der Homepage dieses Projekts zu lesen ist, am besten beschrieben:

> „GPAC is an Open Source multimedia framework for research and academic purposes in different aspects of multimedia, with a focus on presentation technologies (graphics, animation and interactivity).“ [GPAC]

Zum Verändern von Datei-Inhalten dient das GPAC-Tool MP4Box. Mit der Option -h können hint-tracks hinzugefügt werden:

```
MP4box -h=TRACKNUMMER datei.mp4
```

Welche Tracks schon vorhanden sind, kann man herausfinden mit:

```
MP4Box -info datei.mp4
```

MP4Box stellt Dumping-Befehle zur Verfügung, die die Dateistruktur oder, was wichtiger ist, die Abbildung der Videodaten auf RTP Pakete darstellt:

```
MP4Box -drtp -std datei.mp4
```

Alle Dumping-Befehle können mit folgendem Kommando aufgelistet werden:

```
MP4Box -h dump
```

MP4Box kann außerdem Multimedia-Dateien neu verpacken, Medien in ein anderes Format umkodieren und MPEG-4 Szenen-Beschreibung „BIFS“ dekodieren und enkodieren.

### 3.1.3 Die Entwicklungsumgebung

MIDlet-Suites können mit Hilfe des von Sun Microsystems entwickeltem Sun Wireless Toolkit (WTK) übersetzt, paketiert und ausgeführt werden. Das WTK bringt einen Emulator mit (Abbildung 24), der herstellerunabhängig ist. Er unterstützt die meisten APIs, so dass die meisten Programme getestet werden können [SunWTK].


Abbildung 24: Der WTK-Emulator, Screenshot

Das Fehlen eines Editors im WTK macht das Programmieren jedoch nicht unbedingt leicht. Wegen dieser Schwäche fällt die erste Wahl auf NetBeans. NetBeans ist ein im Juni 2000 von Sun Microsystems gegründetes Open Source Projekt mit einem großen Nutzerkreis. Die Entwicklungsumgebung unterstützt sehr viele Programmiersprachen, wie Java, C, C++, sogar PHP und JavaScript. Das wichtigste ist aber, dass NetBeans JavaME unterstützt. Das WTK ist samt Emulator in der Entwicklungsumgebung integriert. Die Besonderheit ist dabei, dass die MIDlets grafisch gestaltet werden können. Der so genannte Visual Mobile Designer ermöglicht den Programmablauf und das Aussehen der Bildschirmfläche grafisch zu gestalten [NetBeans] (Abbildung 25).

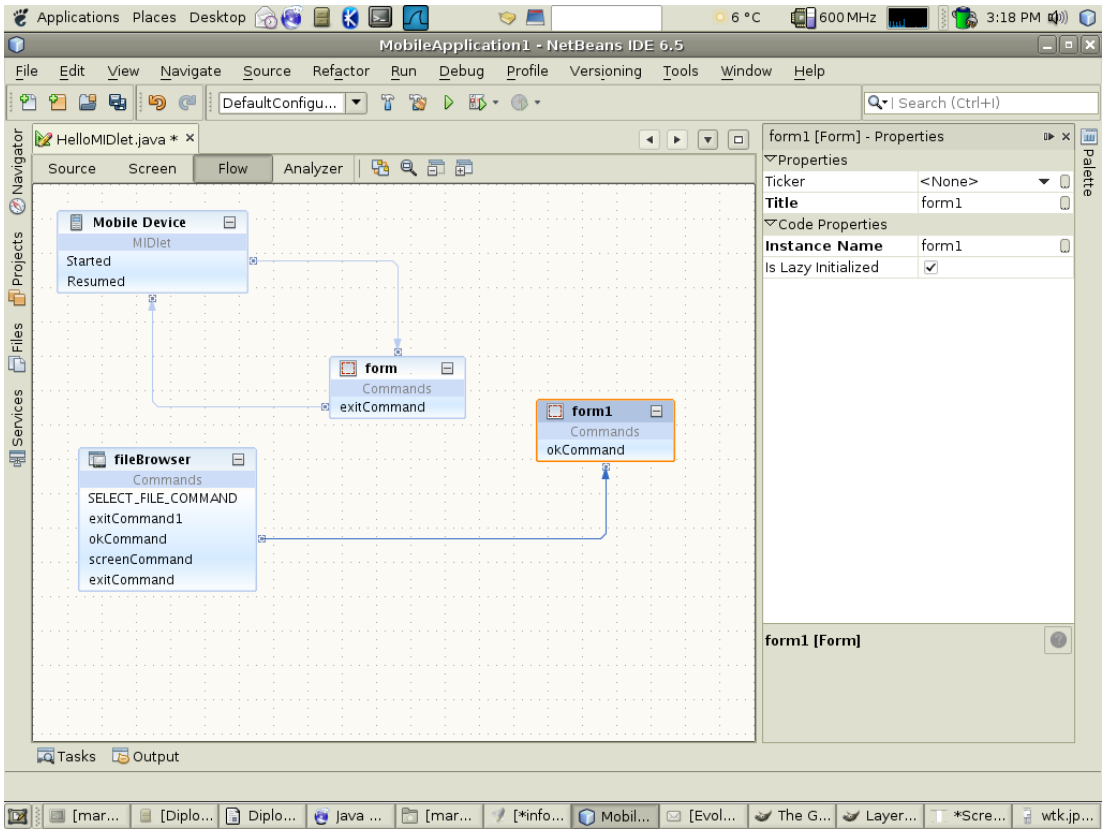


Abbildung 25: NetBeans, Screenshot

Die Installation ist sehr einfach. Sie beschränkt sich auf das Herunterladen der Installationsdatei sowie das Ändern der Benutzerrechte:

```
chmod +x ./<installationsdatei>
```

Ausführen der Datei und das Befolgen der Installationsanweisungen führt zum erfolgreichen Abschluss der Installation.

### 3.1.4 Das Nokia E51

Das Mobiltelefon Nokia E51 (Abbildung 26) ist ein leistungsstarkes Smartphone in sehr kompakter und vor allem schmaller Bauform. Im Inneren befindet sich ein 369MHz schneller ARM 11 Prozessor, 96MB Arbeitspeicher. Das Telefon unterstützt Micro-SD Karten mit bis zu 16GB Speicherkapazität. Als Betriebssystem dient Symbian S60 3rd Edition [InsideHandy].


Abbildung 26: Nokia E51. Quelle [Nokia]

Die Java-Platform ist standardmäßig installiert. Folgende optionale Pakete sind vorhanden [S60.com]:

- MIDP 2.0
- CLDC 1.1

- JSR 135 Mobile Media API
- JSR 172 Web Services API
- JSR 177 Security and Trust Services API
- JSR 179 Location API
- JSR 180 SIP API
- JSR 184 Mobile 3D Graphics API
- JSR 185 JTWI
- JSR 205 Wireless Messaging API
- JSR 226 Scalable 2D Vector Graphics API
- JSR 234 Advanced Multimedia Supplements
- JSR 75 FileConnection and PIM API
- JSR 82 Bluetooth API
- Nokia UI API

Das Telefon sollte für Zwecke dieser Diplomarbeit genügend ausgestattet sein.

## 3.2 Anwendung

Um eine Analyse des Videodatenstroms durchführen zu können, ist es notwendig jedes über das Netzwerk transportierte Videopaket auszuwerten. Das Video muss während der Auswertung auf dem Bildschirm dargestellt werden. Die Videodaten dürfen dabei nicht verändert und die von der Analyse verursachte Verzögerung im Transport muss so gering wie möglich gehalten werden.

### 3.2.1 Ansatz 1 mit Hilfe der MMAPI

Am elegantesten wäre die Anwendung der Mobile Media API. Die `DataSource` ist eine abstrakte Klasse, deren Implementierungen alle Aufgaben der Lokalisierung und des Empfangens kapseln. Somit sollte es möglich sein beliebige Protokolle zu implementieren [MMAPI]. Dazu müssen die Methoden der Klasse `DataSource` überschrieben werden und die Interfaces `SourceStream` implementiert werden [JavaToday_Goyal].

Eine DataSource hat folgenden Aufbau [MMAPI]:

```
public class StreamingDataSource extends DataSource {
  private SourceStream[] streams;
  public StreamingDataSource(String locator) {
      super(locator);
      setLocator(locator);
  }
  public String getLocator() { return locator; }
  public void connect() throws IOException {
        streams = new RTPSourceStream[2];
        streams[0] = new RTPSourceStream(locator,1);
        streams[1] = new RTPSourceStream(locator,2);
  }
  public void disconnect() {
     ((RTPSourceStream)streams[0]).close();
      ((RTPSourceStream)streams[0]).close();
  }
  public void start() throws IOException {
        ((RTPSourceStream)streams[1]).start();
        ((RTPSourceStream)streams[0]).start();
  }
  public void stop() throws IOException {
      ((RTPSourceStream)streams[0]).close();
      ((RTPSourceStream)streams[1]).close();
  }
  public String getContentType() { return "audio/mp4"; }
  public SourceStream[] getStreams(){ return streams; }
}
```

Zusätzlich muss noch das Interface SourceStream implementiert werden. Für jeden Track wird ein SourceStream gestartet. In der Start-Methode des Streams wird das Medium via RTSP vom Server angefordert. Es öffnet einen UDP-Port zum Empfangen der Audio- oder Videodaten und beinhaltet eine Methode, die vom Player für das Empfangen von jedem RTP-Paket aufgerufen wird:

```
public int read(byte[] buffer,int offset,intlength){}
```

Das Abspeichern der Video-Pakete zum späteren Auswerten kann mit Hilfe eines aus `read()` über eine synchronized-Methode steuerbaren Threads geschehen. Die Daten können nicht direkt ausgewertet werden, da dies zu nicht vernachlässigbaren Verzögerungen führen würde.

Wenn "video/mp4" durch die Methode `getContentType()` der Klasse `DataSource` zurückgegeben wird, dann bleibt der Player in der Methode `player.prefetch()` stehen. Wenn aber eine beliebige Zeichenkette den Rückgabewert darstellt, dann startet der Player eine neue, eigene RTSP-Session mit dem Server. Nach Einfügen von Kontrollausgaben in jeder Methode konnte man feststellen, dass die Methode `read()` des `SourceStreams` direkt, dass heißt ohne vorheriges Aufrufen der Methode `start()`, gestartet wurde. Es sollte aber erst in der Methode `start()` ein RTSP-PLAY-Befehl an den Server geleitet werden. Das Aufrufen der `start()`-Methode als letzter Befehl in der `connect()`-Methode (die `connect()`-Methode wird von dem Player korrekt aufgerufen), führt also zum Ausführen des RTSP-Play-Befehls und zum Empfangen von jedem RTP-Pakets (in `read()`). Der Player verlässt aber die Methode `prefetch()` nicht und das Video kann nicht angezeigt werden. Dieser Ansatz wird daher nicht weiter verfolgt.

### 3.2.2 Ansatz 2 mit Hilfe der Low-Level-Netzwerkprogrammierung

Dieser Ansatz basiert auf der Implementierung einer Brücke, die zwischen dem Player und dem Server platziert werden soll. Die Hauptkomponenten darin sind drei Threads. Jeder ist für ein anderes Protokoll zuständig. Den theoretischen Aufbau stellt Abbildung 27 dar.

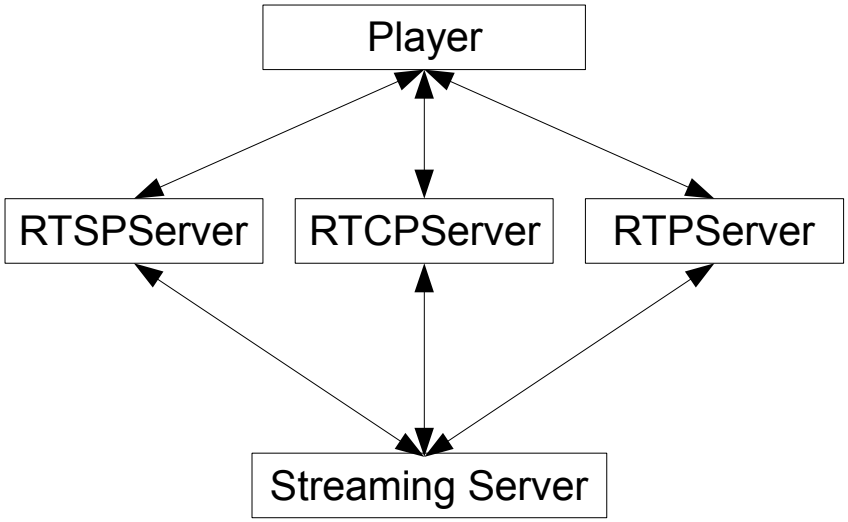


Abbildung 27: Aufbau der Anwendung

Das Klassendiagramm (unvollständig) stellt die Struktur der Anwendung dar (Abbildung 28). Die Einzelnen Klassen werden weiter unten erläutert.

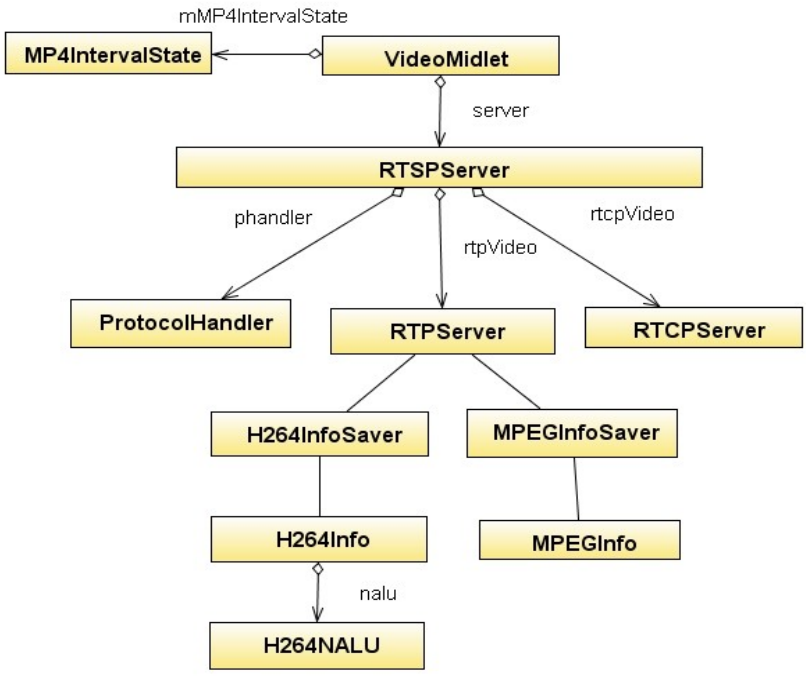


Abbildung 28: Klassendiagramm

RTSPServer

Es handelt sich um ein Thread, der lokal den Port 554 öffnet und auf Anfragen wartet. Alle Anfragen die an dem Port ankommen, werden an den eigentlichen Server weitergeleitet. Dabei ist es notwendig, die IP-Adressen und die Portnummern in den RTSP-Anfragen zu ersetzen.

Die folgende Methode (`getQuery()`) liest eine Anfrage ein. Die -1 als Kennzeichnung des Enden von dem Stream wird nicht gesetzt, da die gesamte Konversation noch nicht beendet ist. Die Methode

```
int anz = InputStream.read(byte[] buffer),
```

die ein „anz“ Bytes in ein Byte-Feld einliest funktioniert auf dem Mobiltelefon auch nicht.

Das Ende einer Anfrage muss daher anders erkannt werden. Das Ende einer RTSP-Nachricht ist an zweifach nacheinander folgenden „\r\n“-Zeichenketten zu erkennen. Folgendes Flussdiagramm erläutert die Funktionsweise dieser Methode (Abbildung 29).

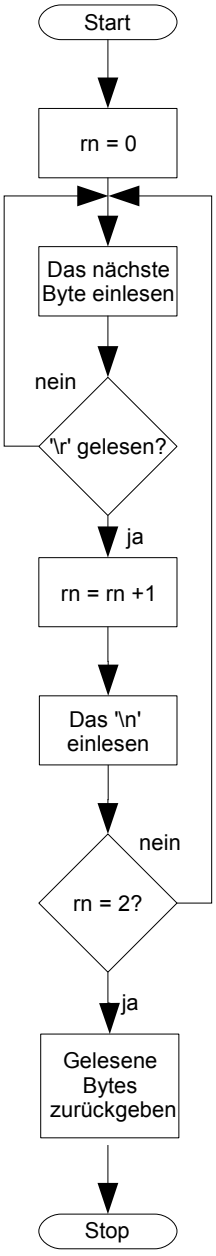


Abbildung 29: Funktoinsweise der Lesemethode des RTSPServer

Das Ersetzen der Port- und IP-Informationen geschieht mit Hilfe einfacher Zeichenketten-Operationen vor dem Weiterleiten an den Server und nach dem Empfangen von ihm in den Methoden: `replacePorts()`, `replaceClientPorts()`, `replaceIP()`.

Die Anfrage SETUP muss geparst werden, um die Portummern, auf denen der Player den Videostrom erwartet herauszufiltern. Dies erfolgt in der Methode

`parsePortInformation()`.

ProtocolHandler

Für die Kommunikation mit dem Streaming-Server kapselt die Klasse ProtocolHandler Methoden für jede Art von RTSP-Nachrichten. Die präparierten Anfragen des Players werden einfach mit Hilfe der `write()`−Methode an den Server-Port versendet (Methode `putResponse()`). Es ist jedoch zu beachten, dass ohne die Anwendung der `flush()`-Methode die Nachricht die Netzwerk-Schnittstelle nicht verlässt.

```
outputStream.write((command).getBytes());
outputStream.flush();
```

Die Antworten des Servers auf DESCRIBE- und SETUP-Anfragen müssen geparst werden. Aus der Antwort auf die SETUP-Anfrage werden mit `parsePortInformation()` die Portnummern herausgefiltert, von denen der Server den Videostrom senden wird. Die Antwort auf die DESCRIBE-Anfrage bedarf spezieller Behandlung (Abbildung 30). Der Grund dafür ist das Fehlen der „\r\n\r\n“-Zeichenfolge am Ende der Nachricht, wodurch das Ende der Antwort nicht eindeutig ist. Die Erkennung des Enden von dieser Antwort richtet sich daher nach dem Vorkommen von Semikolen in wenigen Zeilen: In der Beschreibung des Servers und in der „a=fmtp“-Zeile für jeden Track des Mediums. Dieses Flussdiagramm stellt das Einlesen der Antwort auf eine DESCRIBE-Anfrage für in einer Variablen `tracks` gespeicherten Anzahl von Tracks dar.

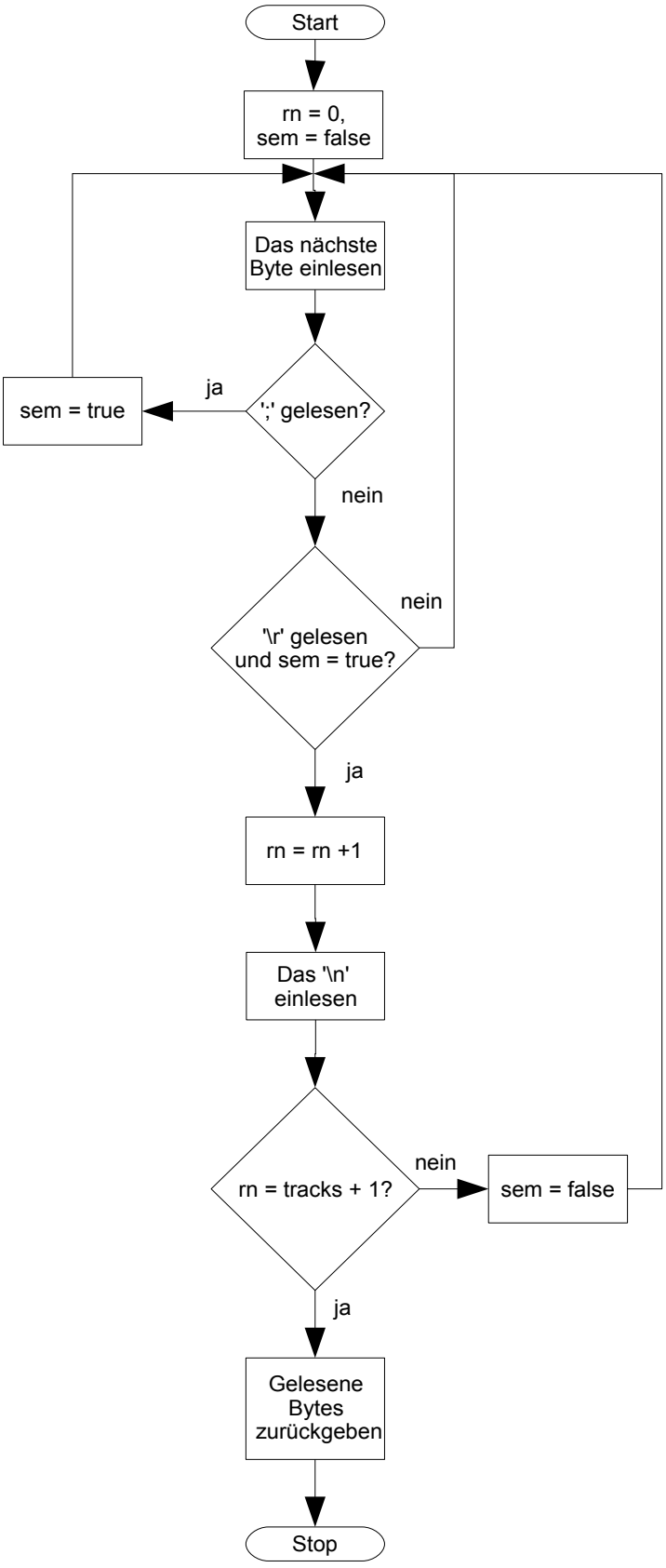


Abbildung 30: Funktionsweise der Lesemethode des ProtocolHandlers

Threads als Brücken

Nach einem erfolgreichen Parsen aller Parameter stehen nun alle Daten zur Verfügung, um sich bei der Videodaten-Übertragung zwischen den Server und den Player zu stellen. Diese Aufgabe haben die Threads `RTPServer` und `RTCPServer`, die im `RTSPServer` gestartet werden. Sie sind in ihrem Aufbau sehr ähnlich. Das Kernstück der `RTPServer`-Klasse ist eine Schleife, in der von der `DatagramConnection dc` ununterbrochen `Datagramme dg` empfangen und versendet werden. Die Ziel-Adresse wird gemäß der Herkunft des Pakets gesetzt.

```
while (stop == false) {
    dg = dc.newDatagram(2048);
    dc.receive(dg);
    if(dg.getAddress().equals(serverAddress)){
         dg.setAddress(playerAddress);
         dc.send(dg);
         if(!h264)mpegInfoSaver.save(dg.getData());
        else if(h264)h264InfoSaver.save(dg.getData());
    }
    else if(dg.getAddress().equals(playerAddress)){
          dg.setAddress(serverAddress);
    }
    dc.send(dg);
}
```

Das Empfangen und Weiterleiten der RTP- und RTCP-Pakete muss voneinander und von der restlichen Anwendung unabhängig sein. Um noch mehr Unabhängigkeit zu gewährleisten, sollte auch das Erfassen der Video-Informationen von dem Empfangen und Weiterleiten unabhängig sein. Dazu werden zwei zusätzliche Threads implementiert. Je nach Typ des Mediums wird ein entsprechender Thread gestartet (Abbildung 30).

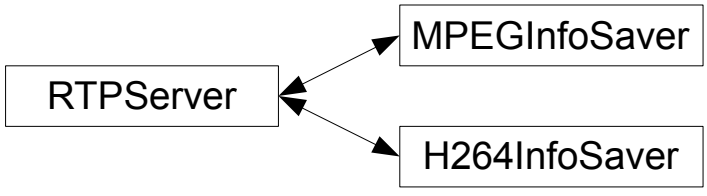


Abbildung 30: InfoSaver-Threads

Der Wert des h264-Flags gibt an, ob die Daten als H.264-Daten oder als MPEG-4-Visual-Daten zu speichern sind. `MPEGInfoSaver` und `H264InfoSaver` sind Threads, bei denen die `run()`-Methode mit der `save()`-Methode synchronisiert ist. Die Hauptkomponente ist hier wieder eine Schleife, die sich in der `run()`-Methode befindet und in der es nach jedem Durchlauf gewartet wird, (in der Methode `wait()`), bis Daten zu speichern sind (bis die Methode `save()` aufgerufen wird). Die Daten werden der RTP-Sequenznummer nach geordnet gespeichert, um spätere, rechenintensivere Sortierung der Daten zu sparen. Der synchronisierte Anweisungsblock (`synchronized(PlayerMidlet.obj){}`) dient als Taktgeber für die Auswertung der Daten. Diese dürfen nicht schneller ausgewertet als erfasst werden, da es zum Leerlaufen der Datenvorräte führen würde.

Die Klassen `MPEGInfoSaver` und `H264InfoSaver` sind fast gleich, jedoch um spätere kompressionsverfahrenspezifische Modifikationen zu erlauben, sind sie separat implementiert.

## 3.3 Deep Inspection

Wikipedia schreibt:

> „Deep Packet Inspection (DPI) (also called complete packet inspection and Information eXtraction - IX -) is a form of computer network packet filtering that examines the data and/or header part of a packet as it passes an inspection point, searching for protocol non-compliance, viruses, spam ...“ [Wikipedia]

Demnach handelt es sich um die Inspektion der Datenpakete durch Auswerten des Header- und Paketinhalts.

Diese Anwendung macht etwas Vergleichbares: Die RTP-Paket-Header werden in Einzelteile auseinander genommen und gespeichert, um den Jitter und Paketverlust zu errechnen. Ferner werden die Video-Daten in den RTP-Nutzdaten geparst und die einzelnen Inhalte des Videodatenstroms zugreifbar gemacht.

### 3.3.1 Datenerfassung

Die Erfassung der Daten beginnt in dem Thread `MPEGInfoSaver` oder `H264InfoSaver` mit dem Anlegen eines neuen Objekts: `MPEGInfo` oder `H264Info`. Diese Instanzen werden anschließend im Vektor `mpegData` gespeichert.

Die Klassen MPEGInfo und H264Info kapseln Informationen, die in genau einem RTP-Paket enthalten sind und stellen sie als Attribute zur Verfügung. Die Attribute werden in den Methoden `getInfo()` und evtl. `getSliceType()` in den `InfoSaver`-Klassen.

Das Extrahieren der RTP-Informationen geschieht durch Abspeichern der Bytes 2 bis 12 des Byte-Feldes und gegebenenfalls durch das Schieben der Bits:

```
int seqNo=(int)(((data[2] & 0xff) << 8) | (data[3] & 0xff));
long timeStamp = (long)(((data[4] & 0xff) << 24) | ((data[5] &0xff)
               << 16) | ((data[6] & 0xff) << 8) | (data[7] & 0xff));
long SSRC = (long)(((data[8] & 0xff) << 24) | ((data[9] & 0xff) <<
             16) | ((data[10] &  0xff) << 8) | (data[11] & 0xff));
```

Das Schieben ohne vorherige Und-Verknüpfung mit 0xff führt dabei zu unbestimmten Ergebnissen.

MPEG-4 Visual

Zusätzlich zu den RTP-Attributen werden noch Informationen über den Video-Datenstrom gespeichert. Im Falle des MPEG-4-Visual-Datenstroms werden alle Elemente des Videostroms durch eine bestimmte Bitfolge „angekündigt“, eine Folge aus 23 Nullen und einer Eins (hexadezimal: 0x000001). Die globalen Konfigurationsdaten werden als SDP-Parameter, während der RTSP-Parameter-Aushandlung übermittelt. Alle Weiteren folgen in den RTP-Nutzdaten. Die Aufgabe

ist es an der Stelle, alle Bitmuster: „0x000001“ zu finden und das darauf folgende Byte, das ein Startcode ist (vgl. Kap. 2.6.1) zu identifizieren. Kommen Visual Object Planes (Startcode 0xB6) in dem Bitstrom vor, werden deren Typen (I, P oder B) gespeichert. Auf eine Darstellung mit Namen wird jedoch aus Geschwindigkeitsgründen verzichtet.

H.264

Beim H.264-Video-Transport nach RFC 3984, ist das Erfassen der Teile des Datenstroms wesentlich komplizierter. Man muss zwischen verschiedenen Paketisierungsarten unterscheiden (Single NAL, STAP-A, STAP-B, etc., vgl. Kap. 2.7.2). Es gibt auch keine Startcodes.

Wenn die NAL-units einzeln transportiert werden, dann steht der 1-Byte große NALU-Header an erster Stelle im RTP-Payload. Beim Transport mehrerer oder eines Fragments eines NALU in dem RTP-Paket, steht an der ersten Stelle ein NAL-Header, der den Aufbau der Daten angibt. Gefolgt wird er von für diese Art spezifischen Parametern (bei STAP-A keine), der Angabe der Länge der nachfolgenden NALU und erst dann von dem eigentlichen NALU-Header sowie dem NALU-Inhalt.

Es kann also anhand des ersten Bytes zwischen den Paketisierungsarten unterschieden werden. Die weitere Verarbeitung wird in separaten Anweisungsblöcken durchgeführt.

Im Folgenden sollen die slice-Typen gespeichert werden. Die NAL-Typen 1 bis 5 beinhalten VCL-Daten. Aus diesen Daten werden Videobilder erzeugt. Die Typen 1, 2 und 5 enthalten slices und ihre Header. Die Information über den Typ der slice wird mit der Methode getSliceType() erfasst. Als etwas schwieriger hat sich dabei das Auslesen der Exp-Golomb[11] Variablen erwiesen, da die Werte der Variablen sich

11 Exp-Golomb code ermöglicht das Darstellen nichtnegativer Werte variabler Länge. Die Anzahl von Nullen vor dem ersten Auftreten einer Eins bestimmt die Länge einer Variablen in Anzahl Bits. Der Wert ergibt sich dann aus dem dargestellten Wert minus eins. Z.B.: 5 => 110 => 00110

über mehrere Bytes erstrecken können. Die Lösung dieses Problems wird anhand des folgenden Flussdiagramms präsentiert (Abbildung 31):

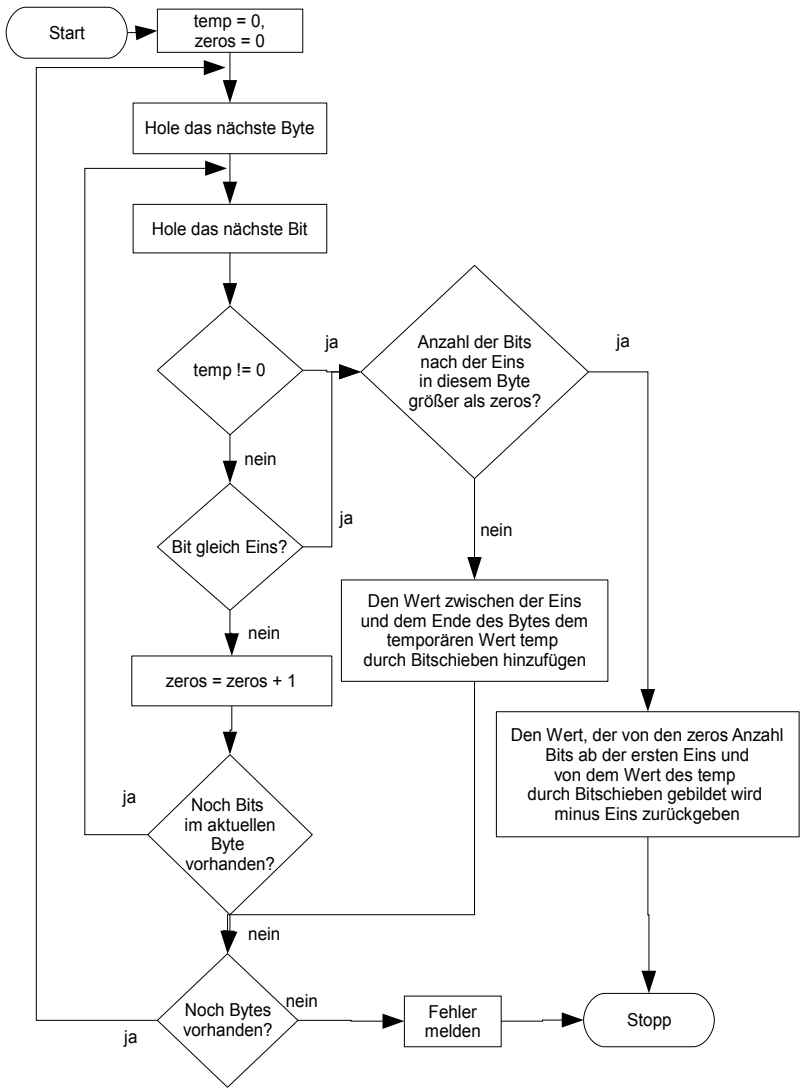


Abbildung 31: Flussdiagramm, Lesen von Exp-Golomb-Variablen

### 3.3.2 Datenauswertung

Die Auswertung der gespeicherten Daten beinhaltet Folgendes:

- Feststellung des Verlustes und Berechnung der Anzahl der verlorenen RTP-Pakete
- Berechnung des Interarrival-Jitters nach RFC3550 (vgl. Kap. 2.5.2)

[14496-10]

- Berechnung des mittleren (arithmetischer Mittelwert) und maximalen Wertes des Jitters
- Berechnung der in den Videodaten vorkommenden I-, P-, und B-VOPs oder -slices

Sie wird abhängig von dem aktuellen Videotyp von den Methoden `mpegBitstreamCheck()` und `h264BitstreamCheck()` durchgeführt. Der Haupt-Anweisungsblock jeder dieser Methoden ist mit Erfassung der Daten synchronisiert, um das Leerlaufen der Datenbestände zu vermeiden.

Das Feststellen, ob ein RTP-Paket verloren ging, ist dank der Speicherung in sortierter Reihenfolge an der Sequenznummer des nächsten Pakets erkennbar, wenn die nicht um eins größer als die des aktuellen ist. Es ist wichtig den Paketverlust vor der Berechnung des Jitters festzustellen, da das Ergebnis ansonsten verfälscht sein würde.

Die Berechnung des Interarrival Jitters erfolgt durch das Implementieren der im RFC3550 definierten Formel. Die dazu notwendigen Daten sind bereits in den gespeicherten Instanzen der Klassen `H264Info` und `MPEGInfo` vorhanden. Die RTP-Zeitstempel und die Paketankunftszeiten haben jedoch nicht dieselbe Auflösung. Die Ankunftszeit eines Pakets wird in Millisekunden erfasst. Die Einheit des RTP-Zeitstempels hat nicht immer den empfohlenen Wert 1/90000 einer Sekunde (90kHz). Sie wird aber in der SDP-Beschreibung des Mediums übermittelt. Die Methode `parseRateInformation()` aus der Klasse `ProtocolHandler` extrahiert diesen Wert.

Ein Beispiel für die Auflösung des Zeitstempels gleich 1000Hz für ein MPEG-4 Visual Datenstrom :

```
a=rtpmap:96 MP4V-ES/1000
```

Die in die Berechnung des Interarrival Jitters eingehenden RTP-Zeitstempel werden mit dem Kehrwert ihrer Auflösung multipliziert. Da die Einheit der lokal gemessenen Zeit eine Millisekunde ist, müssen sie noch mit dem Faktor 1000 multipliziert werden.

$$Zeitstempel\ neu = \frac{Zeitstempel\ alt * 1000}{Auflösung\ des\ Zeitstempels}$$

Alle VOPs und slices sind für jedes RTP-Paket bei dessen Ankunft erfasst worden. Hier gibt es Unterschiede in den vorliegenden Daten zwischen H.264 und MPEG-4 Visual. Beim Ersten wurden NAL-units mit mehreren Attributen erfasst und beim Zweiten die primitiveren VOPs. Aus diesem Grund unterscheiden sich die Anweisungen für das Zusammenzählen dieser Daten.

Die Ergebnisse werden in von dem Benutzer bestimmbaren Intervallen als Objekte der Klasse MP4IntervalState in einem Vektor gespeichert.

# 4. Bedienung

Nach dem Starten der Anwendung wird der Benutzer mehrmalig zum Bestätigen des Zugriffs auf das Dateisystem angefordert. Das hängt mit dem Sicherheitkonzept der Java-Umgebung zusammen, das ausschließlich zertifizierte Anwendungen sicherheitskritische Operationen ohne Bestätigung des Benutzers ausführen lässt. Ein selbst erstelltes Zertifikat reicht nicht aus. Die Anforderungen an ein solches Zertifikat sind „Handy"-herstellerspezifisch. Es gibt aber ein, von den größten Endgeräte-Hersteller ins eben gerufenes Programm: Das sogenannte Java-Verified-Programm [javaverified], das Anwendungen, die das Programm erfolgreich durchlaufen haben, zertifiziert. Die Zertifikate gelten dann auf allen Geräten der teilnehmenden Hersteller. Ein Teil des Programms ist aber eine manuelle Überprüfung der Anwendung durch ein akreditiertes Testhaus, die kostenpflichtig und zeitaufwändig ist [Schmatz]. Auf eine Zertifizierung wird deshalb verzichtet.

Nach dem mehrmaligen Bestätigen des Dateizugriffs, wird eine Eingabemaske sichtbar. In dieser Maske gibt es die Möglichkeit alle zum erfolgreichen Videotest erforderlichen Parameter einzugeben (Abbildung 32).

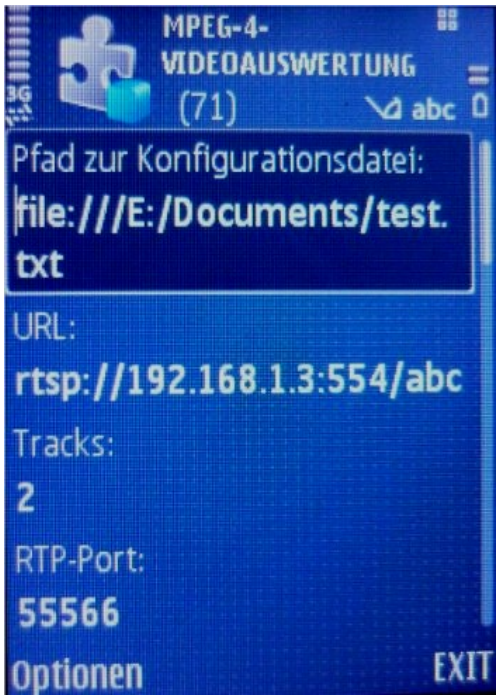


Abbildung 32: Eingabemaske, Screenshot

Das erste Eingabefeld ist mit dem Pfad zu der Standard-Konfigurationsdatei vorbelegt. Sowohl der Speicherort der Konfigurationsdatei als auch die Konfigurationsdatei selbst können verändert werden. Die Eingaben sind beliebig änderbar. Wird ein Parameter nicht gesetzt, dann wird der aus der Konfigurationsdatei verwendet. Sie hat folgenden Aufbau:

```
url=rtsp://192.168.1.10:554/watchmen_gross.mp4
rtp_port=4444
rtcp_port=4445
codec=h264
interval=3000
tracks=1
```

Unter der Eingabemaske, durch das Scrollen der Display-Anzeige erreichbar, befindet sich eine Liste mit den unterstützten Formaten und der vorhandenen Verzeichnisstruktur (Abbildung 33).

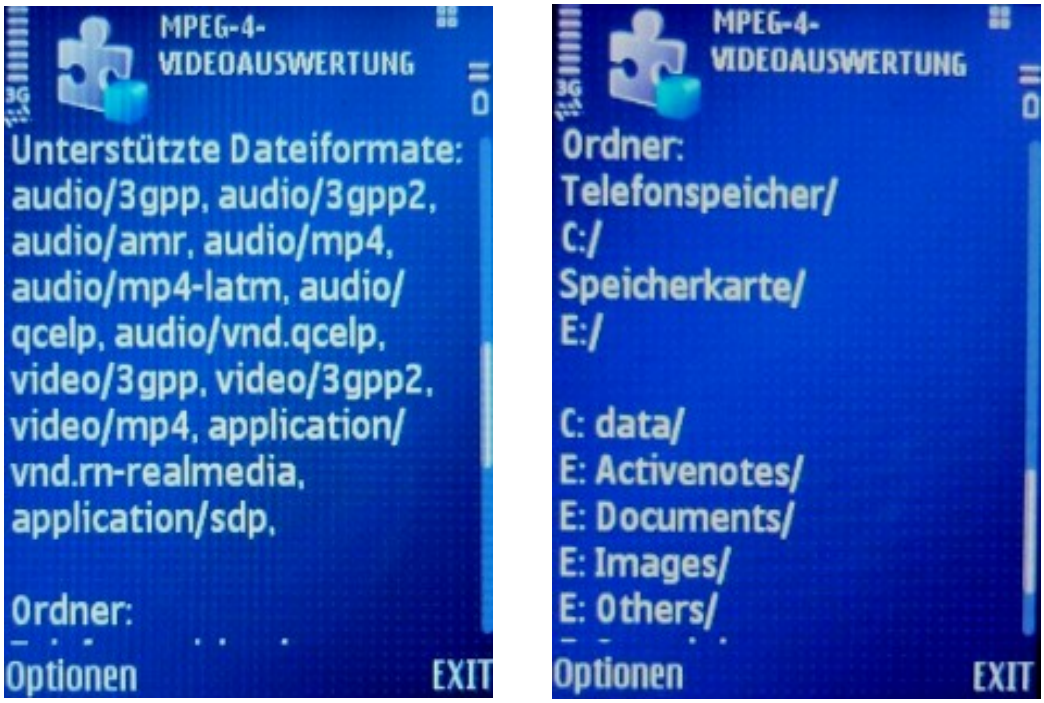


Abbildung 33: Formate und Verzeichnisse, Screenshot

Das Menü der Anwendung beinhaltet sechs Einträge. Die ersten zwei dienen dem Starten und Stoppen des Tests. Die nächsten beiden zeigen die RTSP-Nachrichten, die zwischen dem Player und der Anwendung sowie zwischen dem Streaming-Server und der Anwendung ausgetauscht wurden. Die vorletzte Position stellt die ausgewerteten Daten dar. Die letzte gibt alle Ausnahmen, die während der Laufzeit der Auswertung aufgetreten sind, an (Abbildung 34).


Abbildung 34: Menü, Screenshot

Wenn der Player erfolgreich gestartet und das Medium korrekt vom Server angefordert wurde, dann wird ein neues Formular angezeigt. In diesem ist die fortlaufende Nummer des Auswertungsintervalls und die Anzahl der darin vorkommenden I-, P- und B-VOPs bzw. -slices dargestellt. Weiter unten wird die Anzahl der in dem Intervall verloren gegangenen RTP-Pakete, die Jitter-Werte als Mittel- und Maximalwert des Intervalls und der Interarrival Jitter, der kontinuierlich über den gesamten Auswertungszeitraum bestimmt wird, angezeigt. Der dargestellte Wert des Interarrival Jitters ist der Momentanwert (Abbildung 35).


Abbildung 35: Ergebnismaske, Screenshot

Die Auswertung kann jederzeit beendet werden. Nach dem die Stopp-Taste getätigt wurde gelangt man zur Startseite. Es besteht jetzt die Möglichkeit, die erfassten Informationen über den Menüeintrag „Ausgewertete Daten“ auszugeben.

Bis zu diesem Moment wurden der Portabilität wegen alle Elemente der Benutzungsschnittstelle ausschließlich mit Hilfe des High-Level-API der Bibliothek „LCDUI“ implementiert. Sie lassen sich jedoch nur Zeilenweise platzieren, sodass eine horizontale Ausrichtung der Inhalte schwierig ist. Von den Steuerungszeichen kann nur „\n“ verwendet werden.

Aufgrund der vielen Ergebniswerte, die auf dem Bildschirm dargestellt werden sollen, wird an dieser Stelle auf die Elemente des Low-Level-API zurückgegriffen. Es wird ein `TableItem` aus dem Paket `org.netbeans.microedition.lcdui` verwendet. Die Ergebnisse einer Beispiel-Auswertung sind in Abbildung 36 dargestellt.

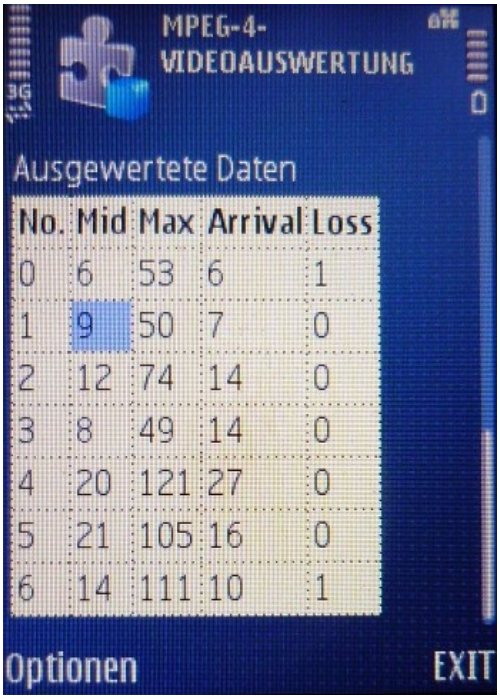


Abbildung 36: Ausgewertete Daten. Screenshot

Nach dem Betätigen der Taste „EXIT“ werden alle erfassten Paketinformationen und alle gemessenen Werte in zwei Dateien geschrieben. Die Namen der Dateien werden aus dem Namen des angeforderten Mediums und einem entsprechenden Zusatz „RTP“ oder „Results“ gebildet.

# 5. Funktionstest

## 5.1 Leistung des Programms

Die Funktionen des Programms wurden auf ihre Leistung überprüft. Der Test stützte sich dabei auf zwei Videodateien [mobile9], die jeweils in drei Bitraten verwendet wurden. Alle haben die Auflösung 176x144 Bildpunkte und alle sind im Profil: „Simple @ L1“ kodiert. Die Ergebnisse für MPEG-4 Visual sind in Tabelle 3 dargestellt.

| Datei | Bitrate in kbps | Paketverlust in % | Gemessener Interarrival Jitter | In RTCP berichteter Interarrival Jitter | Bildqualität |
|---|---|---|---|---|---|
| simpsons_klein.mp4 | 106 | < 1 | 9 ~ 30 | 3 ~ 10 | gut |
| simpsons_mittel.mp4 | 204 | 10 | 19 ~ 40 | 10 ~ 28 | mittelmäßig, gelegentliches Einfrieren des Bildes |
| simpsons_gross.mp4 | 265 | 21 | 33 ~ 46 | 12 ~ 28 | schlecht, gelegentliches Fehlen des Bildes |
| nespresso_klein.mp4 | 142 | 6 | 18 ~ 70 | 6 ~ 33 | gut |
| nespresso_mittel.mp4 | 202 | 22 | 21 ~ 29 | 17 ~ 29 | mittelmäßig, gelegentliches Einfrieren des Bildes |
| nespresso_gross.mp4 | 259 | 24 | 23 ~ 81 | 12 ~ 38 | schlecht, gelegentliches Fehlen des Bildes |

Tabelle 3: Funktionstest MPEG-4 Visual

Die Dateien wurden mit dem VLC-Player in die unterschiedlichen Datenraten umkodiert. Bei den H.264-Dateien haben sich die Profilinformationen von

„Baseline@1.1“ (bei den „_gross“-Dateien, die original belassen sind) in „Unknown 40@5.1“ durch die Umkodierung verändert. Auch die Auflösung wurde von 480x200 in 176x144 verkleinert.

Es werden auch zwei Dateien [mp4point] in drei Datenraten für H.264 verwendet. Das Bild konnte aber von dem Player nicht angezeigt werden. Auch das Benutzen des externen Players führte zum erfolgreichen Anfordern des Mediums und zum Übermitteln der Daten durch den Server, jedoch ohne, dass ein Bild dargestellt wurde. Die Ergebnisse des Tests können Tabelle 4 entnommen werden.

| Datei | Bitrate in kbps | Paketverlust in % | Gemessener Interarrival Jitter | In RTCP berichteter Interarrival Jitter |
|---|---|---|---|---|
| monster_klein.mp4 | 125 | 2 | 12 ~ 20 | 3 ~ 10 |
| monster_mittel.mp4 | 247 | 12 | 9 ~ 11 | 8 ~ 20 |
| monster_gross.mp4 | 501 | 50 | 17 ~ 77 | 8 ~ 78 |
| watchmen_klein.mp4 | 125 | 12 | 6 ~ 8 | 6 ~ 17 |
| watchmen_mittel.mp4 | 243 | 11 | 8 ~ 15 | 6 ~ 14 |
| watchmen_gross.mp4 | 467 | 27 | 11 ~ 34 | 6 ~ 27 |

Tabelle 4: Funktionstest H.264

Den Ergebnissen zufolge kann das Messsystem für Bitraten bis zu 100kbps angewendet werden. Die Anzahl der verlorenen Pakete ist dann meistens Null.

Die berichteten Jitter-Werte wurden mit dem Netzwerk-Sniffer *Wireshark* ausgelesen. Die gemessenen Werte unterscheiden sich geringfügig von den durch den Player berichteten Jitter-Werten. Das liegt zum einen daran, dass die Zeitpunkte des Ermittelns der Werte nicht identisch sein können. Des Weiteren verursacht das Programm eine Verzögerung im Pakettransport, die den Jitter vergrößern, aber auch vermindern kann.

## 5.2 Korrektheit der Funktionen

Die Korrektheit der Arbeitsweise konnte anhand der im vorherigen Kapitel genannten Dateien festgestellt werden. Die Inhalte der Datei mit den Ergebniswerten konnten anhand der Inhalte der Datei mit allen erfassten RTP-Paketen und der mittels de„Wireshark“ ermittelten Daten, überprüft werden.

# 5. Schlussbetrachtungen und Ausblick

Die Java Mobile Edition bietet Möglichkeiten, um die über das IP-Netzwerk übermittelten Daten auf Paketebene zu empfangen, zu versenden und zu verarbeiten. Das Programm konnte erfolgreich als Vermittler zwischen dem Abspielprogramm und dem Server eingesetzt werden. Dabei reichte die Leistung und der Speicherplatz des Mobiltelefons aus, um die Informationen parallel zu vermitteln, zu extrahieren, speichern und auszuwerten. Bei Bitraten über 100kbps stößt das Gerät auf seine Grenzen und schafft es nicht mehr alle Pakete zu bearbeiten.

Das Implementieren des Programmteils für das Extrahieren der MPEG-4 Videoinformationen war im Falle des H.264-Formats etwas schwieriger als bei dem MPEG-4 Visual Format. Das lag zum einen daran, dass es beim RTP-Transport des H.264-Inhaltes mehrere Möglichkeiten zum Verpacken der Videodaten gibt. Zum anderen lag es daran, dass in der Datenstruktur einige Informationen als Exp-Golomb-Code gespeichert sind, deren Inhalte beliebiger Länge sein können.

Das Programm lässt sich in seiner Funktionalität erweitern. Vor allem können mehr Informationen über den Videostrom gesammelt und errechnet werden. Die Einstiegspunkte für die MPEG-4 Visual- und AVC-Videodaten wurden implementiert (für die Übertragung nach RFC3016 und RFC3984). Die weitere Detaillierung der gesammelten Videoinformationen kann mit der Kenntnis von in ISO-14496 definierten Datenstrukturen in den Klassen `H264Info` und `MPEGInfo` stattfinden.

Diese Anwendung kann nach kleinen Änderungen für andere Streaming-Konzepte genutzt werden. Das Steuerungsprotokoll kann geändert werden, in dem man die Implementierung des Kommunikationsablaufs in der Klasse RTSPServer ändert. Solange die Daten über das UDP-Protokoll versendet werden, können die Klassen RTPServer und RTCPServer verwendet werden und nur die Kapselungsklassen MPEGInfo und H264Info an die neuen Daten angepasst werden.

# 6. Literaturverzeichnis


[Austerberry05] David Austerberry, The Technology of Video and Audio Streaming, 2005

[Breymann08] Ulrich Breymann Heiko Mosemann Java ME. Anwendungsentwicklung für Handys, PDA und Co.: Anwendungsentwicklung für Handys, PDA und Co, 2008

[Brockhaus03] Der Brockhaus, Computer und Informatiostechnologie, 2003

[Eidenberger03] Horst M. Eidenberger, Roman Divotkey, Medienverarbeitung in Java, 2003

[Fitzek07] Frank H.P. Fitzek, Frank Reichert, Mobile Phone Programming and its Application to Wireless Networking, 2007

[Goyal] Vikram Goyal, Pro Java ME MMAPI, Mobile Media API for Java Micro Edition, 2006

[MPEG-4_1] ISO/IEC 14496-1, Information technology-Coding of audio-visual objects, Part 1: Systems, 2001

[MPEG-4_2] ISO/IEC 14496-2 Information technology-Coding of audio-visual objects, Part 2: Visual, 2001

[MPEG-4_10] ISO/IEC 14496-10, Information technology-Coding of audio-visual objects, Part 10: Advanced Video Coding, 2004

[MPEG-4_12] ISO/IEC 14496-10, Information technology-Coding of audio-visual objects, Part 12: ISO base media file format, 2005

[Richardson03] Iain E.G. Richardson, H.264 and MPEG-4 Video Compression, 2003

[Schmatz07] Klaus-Dieter Schmatz, Java Micro Edition, Entwicklung mobiler JavaME-Anwendungen mit CLDC und MIDP, 2007

**RFCs**

[RFC2045] Network Working Group, Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, 1996

[RFC2327] Network Working Group, SDP: Session Description Protocol, 1998

[RFC3016] Network Working Group, RTP Payload Format for MPEG-4 Audio/Visual Streams, 2000

[RFC3550] Network Working Group, RTP: A Transport Protocol for Real-Time Applications, 2003

[RFC3984] Network Working Group RTP Payload Format for H.264 Video, 2005

**Internetquellen:**

[DSS] http://dss.macosforge.org

[InsideHandy] http://www.inside-handy.de

[JavaToday_Goyal] http://today.java.net/pub/a/today/2006/08/22/experiments-in-streaming-java-me.html

[javaverified] www.javaverified.com

[NetBeans] http://www.netbeans.org

[Nokia] http://www.nokia.de

[MIDPAPI] http://java.sun.com/javame/reference/apis/jsr118

[mobile9] http://gallery.mobile9.com/f/207340/, (Trailer zum Film „Simpsons“) ,
http://gallery.mobile9.com/f/611688/ (Werbung-„Nespresso“)

[mp4point] http://www.mp4point.com/watchmen-trailer (Trailer zum Film „Watchmen“)
http://www.mp4point.com/monsters-vs-aliens-trailer (Trailer zum Film „Monster vs. Aliens“)

[S60.com] http://www.s60.com

[SunWTK] http://java.sun.com/products/sjwtoolkit

[VLC] http://www.videolan.org

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Anhang

## VideoMidlet.java

```
import javax.microedition.lcdui.*;
import javax.microedition.io.file.*;
import javax.microedition.midlet.*;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.*;
import javax.microedition.io.Connector;
import javax.microedition.media.Player;
import javax.microedition.media.Manager;
import javax.microedition.media.PlayerListener;
import javax.microedition.media.control.VideoControl;
import org.netbeans.microedition.lcdui.SimpleTableModel;
import org.netbeans.microedition.lcdui.TableItem;

public class VideoMidlet extends MIDlet implements Runnable, CommandListener, PlayerListener {

    private Display display;
    //Elemente der Bedienoberfaeche
    private Form form;
    private Form playerForm;
    private TextField pathToConfig;
    private TextField url;
    private TextField port;
    private TextField tracks;
    private TextField interval;
    private ChoiceGroup videoType;
    private StringItem packetLossItem;
    private StringItem jitterItem;
    private StringItem vopCountItem;
    private StringItem sliceCountItem;
    private TableItem tableItem;
    private SimpleTableModel simpleTableModel;
    //Unterstuetzte Formate und vorhandene Verzeichnisse
    private String[] contentTypes;
    private Enumeration roots;
    private Player player;
    //Statische Variablen
    private static final Object obj = new Object();
    private static int rtpRate;
    private static String localIP;
    private static String serverIP;
    /* configParameter:
     * 0: URL
     * 1: RTP-Port
     * 2: RTCP-Port
     * 3: Medientyp
     * 4: Intervall
     * 5: Tracks
     * */
    private static Vector configParameter;
    private static Vector RTSPData;
    private static Vector RTSPServerData;
    private static Vector errorData;
    private static Vector mpegData; //alle gelesenen Videopakete
    private Vector intervals; //Messwerte innerhalb eines Intervalls
    private RTSPServer server;
    private Command start;
    private Command stop;
    private Command showRTSPData;
    private Command showRTSPServerData;
    private Command showInspectedData;
    private Command showErrorData;
    private Command exit;
    private boolean stopCheck = false;
    private Thread t;

    public VideoMidlet() {
        try {
            form = new Form("MPEG-4-VIDEOAUSWERTUNG");
            playerForm = new Form("AUSWERTEND");
            pathToConfig = new TextField("Pfad zur Konfigurationsdatei:",
                    "file:///E:/Documents/test.conf", 100, TextField.URL);
            url = new TextField("URL:", "", 100, TextField.URL);
            port = new TextField("RTP-Port:", "", 6, TextField.DECIMAL);
            tracks = new TextField("Tracks:", "", 1, TextField.DECIMAL);
```

```
            interval = new TextField("Intervall:", "", 10, TextField.DECIMAL);
            videoType = new ChoiceGroup("Videotyp:", Choice.EXCLUSIVE);
            vopCountItem = new StringItem("Nr.     I-VOPs     P-VOPs    B-VOPs", "");
            packetLossItem = new StringItem("Paketverlust: ", "");
            jitterItem = new StringItem("jitter: mid       max      interarrival", "");
            sliceCountItem = new StringItem("Nr.    I-slices     P-slices     B-slices", "");
            //Commands
            start = new Command("Start", Command.SCREEN, 1);
            stop = new Command("Stop", Command.SCREEN, 2);
            showRTSPData = new Command("RTSP-Daten des Players", Command.SCREEN, 3);
            showRTSPServerData = new Command("RTSP-Daten des Servers", Command.SCREEN, 3);
            showInspectedData = new Command("Ausgewertete Daten", Command.SCREEN, 4);
            showErrorData = new Command("Fehler", Command.SCREEN, 5);
            exit = new Command("EXIT", Command.EXIT, 1);
            errorData = new Vector();
        } catch (Exception e) {
            errorData.addElement("\nException in: PlayerMidlet()" + e.toString());
        }
    }

    //Start des Programms
    public void startApp() {
        if (display == null) {
            display = Display.getDisplay(this);
            form.append(pathToConfig);
            form.append(url);
            form.append(tracks);
            form.append(port);
            form.append(interval);
            videoType.append("---------", null);
            videoType.append("H.264", null);
            videoType.append("MPEG-4 Visual", null);
            form.append(videoType);
            form.addCommand(start);
            form.addCommand(stop);
            form.addCommand(showRTSPData);
            form.addCommand(showRTSPServerData);
            form.addCommand(showInspectedData);
            form.addCommand(showErrorData);
            form.addCommand(exit);
            form.setCommandListener(this);
            display.setCurrent(form);
            getSupportetData();
        }
    }

    //Starten und Initialisieren,
    //Bei jedem Teststart ausgeführt
    public void start() {
        try {
            configParameter = new Vector();
            RTSPData = new Vector();
            RTSPServerData = new Vector();
            errorData = new Vector();
            mpegData = new Vector();
            intervals = new Vector();
            stopCheck = false;

            String path = pathToConfig.getString();
            if(!path.equals("")){
                readConfigFile(path);
            }
            //Lesen der Eingabefelder
            //Wenn manuell Parameter eingegeben, dann diese benutzen
            if (!url.getString().equals("")) {
                configParameter.removeElementAt(0);
                configParameter.insertElementAt(url.getString(), 0);
            }
            //Die Pornummer
            if (!port.getString().equals("")) {
                String rtp = port.getString();
                int rtcp = Integer.parseInt(rtp) + 1;
                configParameter.removeElementAt(1);
                configParameter.insertElementAt(rtp, 1);
                //RTCP-Port automatisch setzen
                configParameter.removeElementAt(2);
                configParameter.insertElementAt("" + rtcp, 2);
            }
            //Videotyp
            if (videoType.isSelected(1)) {
                configParameter.removeElementAt(3);
                configParameter.insertElementAt("h264", 3);
            } else if (videoType.isSelected(2)){
                configParameter.removeElementAt(3);
                configParameter.insertElementAt("MPEG-4", 3);
            }
            //Auswertungsintervall
```

```
        String interv = interval.getString();
        if(!interv.equals("")){
            configParameter.removeElementAt(4);
            configParameter.insertElementAt(interv, 4);
        }
        //Die Anzahl der Tracks
        if (!tracks.getString().equals("")) {
             configParameter.removeElementAt(5);
             configParameter.insertElementAt(tracks.getString(), 5);
        }
        String address = (String) configParameter.elementAt(0);
        //Extrahieren der IP
        serverIP = address.substring(address.indexOf("//") + 2,
                    address.indexOf(':', 8));
        server = new RTSPServer();
        server.start();
        t = new Thread(this);
        t.start();
    } catch (Exception e) {
        errorData.addElement("Exception in PlayerMidlet.start()" + e.toString());
    }
}

//Stoppen des Programms
public void stop() {
    try {
        stopCheck = true;
        player.close();
        player = null;
        server.close();
    } catch (Exception e) {
        errorData.addElement("\nException in stop(): " + e.toString());
    }
}

//Anweisungen, die als Thread durchgeführt werden
public void run() {
    try {
        String sckt = ((String)configParameter.elementAt(0)).substring(
                ((String)configParameter.elementAt(0)).indexOf(':', 8));

        String res = "rtsp://" + localIP + sckt;//127.0.0.1 wird von SDP nicht vertragen
        player = Manager.createPlayer(res);
        // Steuerung des Players
        player.addPlayerListener(this);
        player.prefetch();
        player.realize();
        player.start();
        //Da run() im neuen Thread ausgeführt wird und jetzt sowieso beendet wäre
        //kann hier die DeepInspection hier stattfinden
        if (((String) configParameter.elementAt(3)).equals("h264")) {
            h264BitstreamCheck(Integer.parseInt((String) configParameter.elementAt(4)));
        }
        else {
            mpegBitstreamCheck(Integer.parseInt((String) configParameter.elementAt(4)));
            errorData.addElement("\nNACH BITSTREAMCHECK");
        }
    } catch (Throwable ioe) {
        errorData.addElement("\nException in run(): " + ioe.toString());
    }
}

//Auswertungsmethode für MPEG-4 Visual
private void mpegBitstreamCheck(long intervalTime) {
    int i = 0, loop = 0;
    int pvop = 0, ivop = 0, bvop = 0;
    int pvop_all = 0, ivop_all = 0, bvop_all = 0;
    long intervalStartTime = 0;
    MPEGInfo mpegAtom1;
    MPEGInfo mpegAtom2;
    MP4IntervalState mp4IntervalState; //Zustandsdaten in einem Intervall
    int allMissedCount = 0; //Alle Verloneren RTPPakete
    int intervalMissedCount = 0;//In einem Intervall verlorene Pakete
    int allPackets = 0;
    double jitter = 0;
    long mid_jitter = 0;
    long max_jitter = 0;
    long d = 0;
    long d_sum = 0;
    int size = 0;
    Vector d_sorted = new Vector();
    Integer IVOP = new Integer(0);
    Integer PVOP = new Integer(1);
    Integer BVOP = new Integer(2);
    try {
        synchronized (obj) {//synchronisiert mit dem Abspeichern der Pakete
            do {            //das Speichern ermöglicht eine Auswertung zu beliebiger Zeit;
```

```
            obj.wait();//warten auf obj.notify()
            mpegAtom1 = (MPEGInfo) mpegData.elementAt(i);
            mpegAtom2 = (MPEGInfo) mpegData.elementAt(++i);
            allPackets++;
            //Beeim ersten Durchlauf muss hier die Startzeit gesetzt werden
            if (loop == 0) {
                intervalStartTime = mpegAtom1.getIncomeTime();
                loop++;
            }
            //P- B- und I-VOPs zusammenzählen
            if (mpegAtom1.getVops() != null) {
                //mehrere VOPs in einem Paket möglich
                for (int l = 0; l < mpegAtom1.getVops().size(); l++) {
                    if (mpegAtom1.getVops().elementAt(l).equals(IVOP)) {
                        ivop++;
                        ivop_all++;
                    } else if (mpegAtom1.getVops().elementAt(l).equals(PVOP)) {
                        pvop++;
                        pvop_all++;
                    } else if (mpegAtom1.getVops().elementAt(l).equals(BVOP)) {
                        bvop++;
                        bvop_all++;
                    }
                }
            }
            //Packet loss
            if (mpegAtom2.getSeqNo() != (mpegAtom1.getSeqNo() + 1)) {
                intervalMissedCount++;
                allMissedCount++;
                //im Falle eines Paketverlusts liefert die Jitterberechnung
                //falsche Ergebnisse
                continue;
            }

            //alle Jiiter-Berechnungen nur wenn ein neues
            //Videopaket
            if (mpegAtom2.getTimeStamp() != mpegAtom1.getTimeStamp()) {

                //Interarrival Jitter
                double rate = (double) 1000 / (double) getRTPRate();
                d = (mpegAtom2.getIncomeTime() - mpegAtom1.getIncomeTime());
                d=d-((long)((mpegAtom2.getTimeStamp()-mpegAtom1.getTimeStamp()) * rate));
                //addErrorData("rate: " + rate + "d: " + d);
                d = Math.abs(d);
                d_sum += d;
                jitter = jitter + ((d - jitter) / 16);
                //Jitter
                size = d_sorted.size();
                if (size == 0) {
                    d_sorted.addElement(new Long(d));
                } else {
                    while(size>0 && ((Long)d_sorted.elementAt(size - 1)).longValue()>d) {
                        size--;
                    }
                    d_sorted.insertElementAt(new Long(d), size);
                }
            }
            //für jedes Intervall: Ausgabe und Abspeichern
            if ((mpegAtom1.getIncomeTime() - intervalStartTime) > intervalTime) {
                intervalStartTime = mpegAtom1.getIncomeTime();
                //Jitter-gemittelt ermittlen
                mid_jitter = d_sum / d_sorted.size();
                //maximaler Jitter
                max_jitter = ((Long) d_sorted.elementAt(d_sorted.size() -
                                                                1)).longValue();
                //Ausgabe
                packetLossItem.setText("" + intervalMissedCount);
                jitterItem.setText("            " + mid_jitter + "         "
                                   + max_jitter + "          " + (long)jitter);
                vopCountItem.setText(loop++ + "          " + ivop + "          "
                                              + pvop + "          " + bvop);
                //Abspeichern der Ergebnisse in einer Klasse gekapselt
               double percentMissed=((double)allMissedCount / (double)allPackets) * 100;
               mp4IntervalState = new MP4IntervalState(intervalStartTime,
                                  intervalMissedCount, allMissedCount,
                                    (int)percentMissed, mid_jitter, max_jitter,
                                       (long)jitter, ivop, pvop, bvop);
                intervals.addElement(mp4IntervalState);
                //Intervallspezifische Werte zurücksetzen
                intervalMissedCount = 0;
                pvop = 0;
                ivop = 0;
                bvop = 0;
                d_sum = 0;
                d_sorted = new Vector();
            }
        } while (stopCheck == false);
```

```
            }
        } catch (Exception e) {
            errorData.addElement("\n\n!!!EXCEPTION in mpegBitstreamCheck(): " + e.toString());
            return;
        }
    }

    //Die Auswertungsmethode für H.264
    private void h264BitstreamCheck(long intervalTime) {
        int i = 0, loop = 0;
        int pslice = 0, islice = 0, bslice = 0;
        int pslice_all = 0, islice_all = 0, bslice_all = 0;
        long intervalStartTime = 0;
        long mid_jitter = 0;
        long max_jitter = 0;
        int size;
        Vector d_sorted = new Vector();
        H264Info mpegAtom1;
        H264Info mpegAtom2;
        H264NALU h264Nalu;
        MP4IntervalState mp4IntervalState; //Zustandsdaten in einem Intervall
        int allMissedCount = 0; //Alle Verloneren RTPPakete
        int allPackets = 0;
        int intervalMissedCount = 0;//In einem Intervall verlorene Pakete
        double jitter = 0;
        long d = 0;
        long d_sum = 0;

        try {
            synchronized (obj) {//synchronisiert mit dem Abspeichern der Pakete
                do {            //das Speichern ermöglicht eine Auswertung zu beliebiger Zeit
                    obj.wait();//warten auf obj.notify()
                    mpegAtom1 = (H264Info) mpegData.elementAt(i);
                    mpegAtom2 = (H264Info) mpegData.elementAt(++i);
                    allPackets ++;
                    if (loop == 0) {
                        intervalStartTime = mpegAtom1.getIncomeTime();
                        loop++;
                    }
                    //P- B- und I-slices zusammenzählen
                    if (mpegAtom1.getNalus() != null) {
                        //mehrere NALUs in einem Paket möglich
                        for (int l = 0; l < mpegAtom1.getNalus().size(); l++) {
                            h264Nalu = (H264NALU) mpegAtom1.getNalus().elementAt(l);
                            if (h264Nalu.getSliceType() == 2 || h264Nalu.getSliceType() == 7) {
                                islice++;
                                islice_all++;
                             } else if (h264Nalu.getSliceType()==0 || h264Nalu.getSliceType()==5)
{
                                pslice++;
                                pslice_all++;
                             } else if (h264Nalu.getSliceType()==1 || h264Nalu.getSliceType()==6)
{
                                bslice++;
                                bslice_all++;
                            }
                        }
                    }
                    //Packet loss
                    if (mpegAtom2.getSeqNo() != (mpegAtom1.getSeqNo() + 1)) {
                        intervalMissedCount++;
                        allMissedCount++;
                        //im Falle eines Paketverlusts liefert die Jitterberechnung
                        //falsche Ergebnisse
                        continue;
                    }
                   //alle Jiiter-Berechnungen nur wenn ein neues
                    //Videopaket
                    if (mpegAtom2.getTimeStamp() != mpegAtom1.getTimeStamp()) {
                        //Interarrival Jitter
                        double rate = (double) 1000 / (double) getRTPRate();
                        d = (mpegAtom2.getIncomeTime() - mpegAtom1.getIncomeTime());
                        d = d - ((long) ((mpegAtom2.getTimeStamp() -
                                          mpegAtom1.getTimeStamp()) * rate));
                        //addErrorData("rate: " + rate + "d: " + d);
                        d = Math.abs(d);
                        d_sum += d;
                        jitter = jitter + ((d - jitter) / 16);
                        //Jitter
                        size = d_sorted.size();
                        if (size == 0) {
                            d_sorted.addElement(new Long(d));
                        } else {
                            while(size>0 && ((Long)d_sorted.elementAt(size - 1)).longValue()>d) {
                                size--;
                            }
                            d_sorted.insertElementAt(new Long(d), size);
```

```
                    }
                }
                //für jedes Intervall: Ausgabe und Abspeichern
                if ((mpegAtom1.getIncomeTime() - intervalStartTime) > intervalTime) {
                    intervalStartTime = mpegAtom1.getIncomeTime();
                    //Jitter-gemittelt ermittlen
                    mid_jitter = d_sum / d_sorted.size();
                    //maximaler Jitter
                    max_jitter = ((Long) d_sorted.elementAt(d_sorted.size()-1)).longValue();
                    //Ausgabe
                    packetLossItem.setText("" + intervalMissedCount);
                    jitterItem.setText("              " + mid_jitter +
                            "       " + max_jitter + "       " + (long)jitter);
                    sliceCountItem.setText(loop++ + "       " + islice +
                            "          " + pslice + "          " + bslice);
                    //Abspeichern der Ergebnisse in einer Klasse gekapselt
                    double percentMissed = ((double)allMissedCount /
                                            (double)allPackets) * 100;
                    mp4IntervalState = new MP4IntervalState(intervalStartTime,
                                        intervalMissedCount,allMissedCount,
                                        (int)percentMissed, mid_jitter, max_jitter,
                                        (long)jitter, islice, pslice, bslice);
                    intervals.addElement(mp4IntervalState);
                    //Intervallspezifische Werte zurücksetzen
                    intervalMissedCount = 0;
                    pslice = 0;
                    islice = 0;
                    bslice = 0;
                    d_sum = 0;
                    d_sorted = new Vector();
                }
            } while (stopCheck == false);
        }
    } catch (Exception e) {
        errorData.addElement("\n\n!!!EXCEPTION in h264BitstreamCheck(): " + e.toString());
        return;
    }
}

//Auslesen der Konfigurationsinformationen
private void readConfigFile(String path){
    FileConnection configFileConnection = null;
    InputStream is = null;
    StringBuffer fileContentBuffer = new StringBuffer();
    int index;
    try {
            configFileConnection = (FileConnection) Connector.open(path);
            if (configFileConnection.exists()) {
                is = configFileConnection.openInputStream();
                int ch;
                while ((ch = is.read()) != -1) {//Solange bis Dateiende nicht erreicht
                        fileContentBuffer.append((char) ch);
                }
                is.close();
                String fileContent = fileContentBuffer.toString();
                index = fileContent.indexOf("url=") + "url=".length();
                configParameter.insertElementAt(fileContent.substring(index,
                                fileContent.indexOf("\n",index)).trim(), 0);
                index = fileContent.indexOf("rtp_port=") + "rtp_port=".length();
                configParameter.insertElementAt(fileContent.substring(index,
                                fileContent.indexOf("\n",index)).trim(), 1);
                index = fileContent.indexOf("rtcp_port=") + "rtcp_port=".length();
                configParameter.insertElementAt(fileContent.substring(index,
                                fileContent.indexOf("\n",index)).trim(), 2);
                index = fileContent.indexOf("codec=") + "codec=".length();
                configParameter.insertElementAt(fileContent.substring(index,
                                fileContent.indexOf("\n",index)).trim(), 3);
                index = fileContent.indexOf("interval=") + "interval=".length();
                configParameter.insertElementAt(fileContent.substring(index,
                                fileContent.indexOf("\n",index)).trim(), 4);
                index = fileContent.indexOf("tracks=") + "tracks=".length();
                configParameter.insertElementAt(fileContent.substring(index).trim(), 5);
            }else {
                //Alert
            }

        } catch (Exception e) {
            errorData.addElement("\n!!!Exception beim Öffnen der Datei: " +
                                e.toString());
            if (configFileConnection != null) {
                try {
                    configFileConnection.close();
                } catch (IOException ioe) {
                    errorData.addElement("Exception beim Schliessen der Datei: "
                                + ioe.toString());
                }
            }
```

```
            }
    }

    //Ermitteln der unterstützen Datentypen
    //und der vorhandenen Verzeichnisse
    private void getSupportetData(){
            FileConnection fcC = null;
            FileConnection fcE = null;
            String s = "Unterstützte Dateiformate: \n";
            try {
                contentTypes = Manager.getSupportedContentTypes("rtsp");
                s += "\n";
                int length = contentTypes.length;
                for (int i = 0; i < length; i++) {
                    s += contentTypes[i] + ", ";
                }
                roots = FileSystemRegistry.listRoots();
                s += "\n\nOrdner:\n";
                while (roots.hasMoreElements()) {
                    s += roots.nextElement() + "\n";
                }

                fcC = (FileConnection) Connector.open("file:///C:/");
                fcE = (FileConnection) Connector.open("file:///E:/");
                Enumeration filesC = fcC.list();
                Enumeration filesE = fcE.list();
                while (filesC.hasMoreElements()) {
                    s += "\nC: " + filesC.nextElement();
                }
                while (filesE.hasMoreElements()) {
                    s += "\nE: " + filesE.nextElement();
                }
                form.append(s);
                fcC.close();
                fcE.close();
            } catch (Exception e) {
                errorData.addElement("\n!!!Exception beim Auslesen des Wurzelbaums: "
                                + e.toString());
                if (fcC != null) {
                    try {
                        fcC.close();
                    } catch (IOException ioe) {
                        errorData.addElement("\n!!!Exception beim Auslesen des Wurzelbaums"
                                + ioe.toString());
                    }
                }
                if (fcE != null) {
                    try {
                        fcE.close();
                    } catch (IOException ioe) {
                        errorData.addElement("\n!!!Exception beim Auslesen des Wurzelbaums"
                                + ioe.toString());
                    }
                }
            }
    }

    //Abspeichern der Daten
    private void writeData() {
        FileConnection fcRTP = null;
        FileConnection fcResults = null;
        OutputStream osRTP = null;
        OutputStream osResults = null;
        String path = (String)configParameter.elementAt(0);
        String fileName = path.substring(path.indexOf("/", 10) + 1);
        String s = "SeqNo.\tIncome time\t\tTimestamp\tVideodaten\n";
        try {
            fcRTP = (FileConnection)Connector.open("file:///E:/Documents/rtpData_" +
                    fileName + ".txt");
            fcResults = (FileConnection)Connector.open("file:///E:/Documents/results_" +
                    fileName + ".txt");
            if (!fcRTP.exists()){
                fcRTP.create();
            }
            if (!fcResults.exists()){
                fcResults.create();
            }
            osRTP = fcRTP.openOutputStream();
            osResults = fcResults.openOutputStream();
            for (int i = 0; i < mpegData.size(); i++) {
                if (((String) configParameter.elementAt(3)).equals("h264")) {
                    s += "" + ((H264Info) mpegData.elementAt(i)).getSeqNo() + "\t" +
                            ((H264Info) mpegData.elementAt(i)).getIncomeTime() + "\t" +
                            ((H264Info) mpegData.elementAt(i)).getTimeStamp() + "\t";
                     for (int l = 0; l < ((H264Info) mpegData.elementAt(i)).getNalus().size(); l+
+){
                        s += " " + ((H264NALU)((H264Info) mpegData.elementAt(i)).
```

```
                                        getNalus().elementAt(l)).getSliceType();
                    }
                    s += "\n";
                } else {
                    s += "" + ((MPEGInfo) mpegData.elementAt(i)).getSeqNo() + "\t" +
                            ((MPEGInfo) mpegData.elementAt(i)).getIncomeTime() + "\t" +
                            ((MPEGInfo) mpegData.elementAt(i)).getTimeStamp() + "\t";
                     for (int l = 0; l < ((MPEGInfo) mpegData.elementAt(i)).getVops().size(); l++)
{
                        s += " " + ((MPEGInfo) mpegData.elementAt(i)).getVops().elementAt(l);
                    }
                    s += "\n";
                }
            }
            osRTP.write(s.getBytes());
            osRTP.close();
            fcRTP.close();

            s = "missed\tall missed\t%missed\tmid\tmax\tinter arrival\n";
            for (int i = 0; i < intervals.size(); i++) {
                if (((String) configParameter.elementAt(3)).equals("h264")) {
                   s+=""+((MP4IntervalState) intervals.elementAt(i)).getMissedPackets() + "\t" +
                   ((MP4IntervalState) intervals.elementAt(i)).getAllMissedPackets() + "\t\t" +
                   ((MP4IntervalState) intervals.elementAt(i)).getMissedPercent() + "\t\t" +
                   ((MP4IntervalState) intervals.elementAt(i)).getMidJitter() + "\t" +
                   ((MP4IntervalState) intervals.elementAt(i)).getMaxJitter() + "\t" +
                   ((MP4IntervalState) intervals.elementAt(i)).getJitter() + "\n";
                } else {
                    s+=""+((MP4IntervalState) intervals.elementAt(i)).getMissedPackets() + "\t" +
                    ((MP4IntervalState) intervals.elementAt(i)).getAllMissedPackets() + "\t\t" +
                    ((MP4IntervalState) intervals.elementAt(i)).getMissedPercent() + "\t\t" +
                    ((MP4IntervalState) intervals.elementAt(i)).getMidJitter() + "\t" +
                    ((MP4IntervalState) intervals.elementAt(i)).getMaxJitter() + "\t" +
                    ((MP4IntervalState) intervals.elementAt(i)).getJitter() + "\n";
                }
            }
            osResults.write(s.getBytes());
            osResults.close();
            fcResults.close();
        } catch (Exception e) {
            errorData.addElement("\n!!!EXCEPTION in writeData(): " + e.toString());
            if (fcRTP != null) {
                try {
                    osRTP.close();
                    fcRTP.close();
                } catch (IOException ioe) {
                    errorData.addElement("\n!!!EXCEPTION in writeData(): " + ioe.toString());
                }
            }
            if (fcResults != null) {
                try {
                    osResults.close();
                    fcResults.close();
                } catch (IOException ioe) {
                    errorData.addElement("\n!!!EXCEPTION in writeData(): " + ioe.toString());
                }
            }
        }
    }

    //Reaktionen auf Zustandsänderungnen des Players
    public void playerUpdate(Player player, String event, Object eventData) {
        try {
            if (event.equals(PlayerListener.STARTED)) {
                VideoControl vc = null;
                if ((vc = (VideoControl) player.getControl("VideoControl")) != null) {
                    Item videoDisp = (Item) vc.initDisplayMode(vc.USE_GUI_PRIMITIVE, null);
                    playerForm.append(videoDisp);
                    vc.setDisplaySize(130, 100);
                    if (((String) configParameter.elementAt(3)).equals("h264")) {
                        playerForm.append(sliceCountItem);
                    } else {
                        playerForm.append(vopCountItem);
                    }
                    playerForm.append(packetLossItem);
                    playerForm.append(jitterItem);
                    playerForm.addCommand(stop);
                    playerForm.addCommand(exit);
                    playerForm.setCommandListener(this);
                    display.setCurrent(playerForm);
                }
            } else if (event.equals(PlayerListener.CLOSED)) {
                playerForm.deleteAll();
                display.setCurrent(form);
            }
        } catch (Exception e) {
            errorData.addElement("\nEXCEPTION in playerUpdate(): " + e.toString());
```

```
    }
}

//Benutzerinteraktion
public void commandAction(Command command, Displayable disp) {
    try {
        if (command == showRTSPData) {
            showReceivedRTSPData();
        } else if (command == showRTSPServerData) {
            showReceivedRTSPServerData();
        } else if (command == showInspectedData) {
            showInspectedVideoData();
        } else if (command == showErrorData) {
            showErrorData();
        } else if (command == start) {
            start();
        } else if (command == stop) {
            stop();
        } else if (command == exit) {
            stop();
            //Programm schliessen
            destroyApp(false);
            notifyDestroyed();
        }
    } catch (Exception e) {
        errorData.addElement("\n!!!EXCEPTION in commandAction()" + e.toString());
    }
}
//RTSP Nachrichten des Players anzeigen
public void showReceivedRTSPData() {
    form.delete(6);
    form.append(RTSPData.toString());
}
//RTSP Nachrichten des Servers anzeigen
public void showReceivedRTSPServerData() {
    form.delete(6);
    form.append(RTSPServerData.toString());
}
//Ausgewertete Informationen anzeigen
public void showInspectedVideoData() {
    String values[][] = new String[intervals.size()][5];
    int i;
    simpleTableModel = new SimpleTableModel(new String[][]{
    new String[]{"", "", "", ""}}, new String[]{"No.", "Mid", "Max", "Arrival", "Loss"});

    for (i = 0; i < intervals.size(); i++) {
        values[i][0] = "" + i;
        values[i][1] = "" + ((MP4IntervalState) intervals.elementAt(i)).getMidJitter();
        values[i][2] = "" + ((MP4IntervalState) intervals.elementAt(i)).getMaxJitter();
        values[i][3] = "" + ((MP4IntervalState) intervals.elementAt(i)).getJitter();
        values[i][4] = "" + ((MP4IntervalState) intervals.elementAt(i)).getMissedPackets();
    }
    simpleTableModel.setValues(values);
    tableItem = new TableItem(Display.getDisplay(this), "Ausgewertete Daten");
    tableItem.setModel(simpleTableModel);
    form.delete(6);
    form.append(tableItem);
}
//Fehler anzeigen
public void showErrorData() {
    form.delete(6);
    form.append(errorData.toString());
}

public static Object getObj(){
    return obj;
}

public static  Vector getConfigParameter(){
    return configParameter;
}

public static String getLocalIP(){
    return localIP;
}

public static String getServerIP(){
    return serverIP;
}

public static Vector getRTSPData(){
    return RTSPData;
}

public static Vector getRTSPServerData(){
    return RTSPServerData;
}
```

```
    public static Vector getErrorData(){
        return errorData;
    }
    public static Vector getMpegData(){
        return mpegData;
    }

    public static void notifyObj(){
        obj.notify();
    }

    public static  void setConfigParameter(Vector v){
        configParameter = v;
    }

    public static void setLocalIP(String ip){
        localIP = ip;
    }

    public static void setServerIP(String ip){
        serverIP = ip;
    }

    public static void setRTSPData(Vector v){
        RTSPData = v;
    }

    public static void setRTSPServerData(Vector v){
        RTSPServerData = v;
    }

    public static void setErrorData(Vector v){
        errorData = v;
    }
    public static void setMpegData(Vector v){
        mpegData = v;
    }

    public static void addRTSPData(String s){
        RTSPData.addElement(s);
    }

    public static void addRTSPServerData(String s){
        RTSPServerData.addElement(s);
    }

    public static void addErrorData(String s){
        errorData.addElement(s);
    }

    public static void addMpegData(Object o){
        mpegData.addElement(o);
    }

    public static void insertMpegData(Object o, int pos){
        mpegData.insertElementAt(o, pos);
    }

    public static void setRTPRate(int rate){
        rtpRate = rate;
    }

    public static int getRTPRate(){
        return rtpRate;
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
        writeData();
    }
}
```

# RTSPServer.java

```
import java.util.Vector;
import javax.microedition.io.*;
import java.io.*;

public class RTSPServer implements Runnable {

    private ServerSocketConnection server;
```

```
private SocketConnection client;
private InputStream dis;
private OutputStream dos;
private Vector ports = new Vector(4);
private RTCPServer   rtcpVideo;
private RTPServer   rtpVideo;
private Thread t;
private ProtocolHandler phandler;
private boolean h264;
private boolean stop = false;

public RTSPServer() {
    try {
        phandler =
                  new ProtocolHandler((String)VideoMidlet.getConfigParameter().elementAt(0));
        server = (ServerSocketConnection) Connector.open("socket://:554");
        VideoMidlet.setLocalIP(server.getLocalAddress());
        if(((String)VideoMidlet.getConfigParameter().elementAt(3)).equals("h264"))h264= true;
        else h264 = false;
    } catch (Exception e) {
         VideoMidlet.addErrorData("\n!!!EXCEPTION at creation of the new DatagramConnections:"
                                                                              + e.toString());
    }
}

//Den Thread starten
public void start() {
    t = new Thread(this);
    t.start();
}

//Die Serverfunktionalität nach dem Starten des Threads
public void run() {
    try {
        String query;
        client = (SocketConnection) server.acceptAndOpen();
        dis = client.openInputStream();
        dos = client.openOutputStream();
        //OPTIONS
        query = getQuery();
        query = replaceIP(query,1);
        putResponse("RTSP/1.0 200 OK\r\nServer: DSS/5.5.5 (Build/489.16; " +
                "Platform/S60; Release/Darwin; state/beta; )\r\nCseq: 1\r\n" +
                "Public: DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE, OPTIONS, " +
                "ANNOUNCE, RECORD\r\n\r\n");
        //DESCRIBE
        query = getQuery();
        query = replaceIP(query,1);
        query = phandler.describe(query);
        putResponse(query);
        //SETUP
        query = getQuery();
        query = replaceIP(query,1);
        parsePortInformation(query);//Die Ports des Players extrahieren
        query = replaceClientPorts(query, 1);//Die PlayerPorts gegen eigene tauschen
        query = phandler.setup(query);
        //Die PlayerPorts wieder zurück- und ServerPorts gegen eigene tauschen
        query = replacePorts(query, 1);
        putResponse(query);
        //SETUP Ev. auch für Audio
        for (int l = 0; l < Integer.parseInt((String)VideoMidlet.
                          getConfigParameter().elementAt(5)); l++) {
            query = getQuery();
            query = replaceIP(query,1);
            parsePortInformation(query);//Die Ports des Players extrahieren
            //query = replaceClientPorts(query, 1);//Die PlayerPorts gegen eigene tauschen
            query = phandler.setup(query);
            //query = replacePorts(query, 1);//Die PlayerPorts wieder zurück-
            //und ServerPorts gegen eigene tauschen
            putResponse(query);
        }
        //RTCP-Meldungen empfangen und senden
        rtcpVideo = new RTCPServer("datagram://" + VideoMidlet.getServerIP() +
                    ":" + phandler.darwinPorts.elementAt(1),"datagram://" +
                    VideoMidlet.getLocalIP() + ":" + ports.elementAt(1));
        rtcpVideo.start();

        rtpVideo = new RTPServer("datagram://" + VideoMidlet.getLocalIP() +
                ":" + ports.elementAt(0),"datagram://" + VideoMidlet.
                getServerIP() + ":" + phandler.darwinPorts.elementAt(0),h264);
        rtpVideo.start();

        //Alle weiteren Befehle an den Server
        //dazwischen mehrmaliges PAUSE und PLAY
        //automatisch vom Player ausgeführt
        while (!stop){
            query = getQuery();
```

```
            query = replaceIP(query,1);
            query = phandler.uniCommand(query);
            putResponse(query);
        }
    } catch (Exception ioe) {
        VideoMidlet.addErrorData("\n!!!Exception im Server: " + ioe.toString());
    }
}

//Eine RTSP-Anfrage des Players lesen
private String getQuery() throws IOException {
    String request;
    byte b[] = new byte[2048];
    int c, i = 0;
    int mark = 0;

    while ((c = dis.read()) != -1) {
        b[i++] = (byte) c;
        if (mark == 0x0a && c == 0x0d) {//wenn zwei Mal \r\n dann fertig
            b[i++] = (byte) dis.read();//letztes \n einlesen
            break;
        }
        mark = c;
    }
    request = new String(b, 0, i); //Eine Timeout-Abfrage einbauen!!!!
    b = null;
    VideoMidlet.addRTSPData("\nQUERY: \n" + request);

    return request;
}

//Auf eine Anfrage antworten
private void putResponse(String response) {
    try {
        VideoMidlet.addRTSPData("\nRESPONSE: \n" + response);
        dos.write(response.getBytes());
        dos.flush(); //Ohne Flush werden die Daten nicht verschickt
    } catch (IOException ioe) {
        VideoMidlet.addErrorData("\n!!!EXCEPTION in put Response:" + ioe.toString());
    }
}

//Resourcen freigeben
public void close() throws IOException {
    try {
        server.close();
        server = null;
        stop = true;
        dis.close();
        dos.close();
        client.close();
        rtcpVideo.stop();
        rtpVideo.stop();
        t.interrupt();
    } catch (Exception e) {
        VideoMidlet.addErrorData("\n!!!EXCEPTION in RTSPServer.close" + e.toString());
    }

}

//Portnummern extrahieren
private void parsePortInformation(String response) {
    String temp = response;
    int index = temp.indexOf("client_port=");
    int baseIdx1 = index + "client_port=".length();
    int baseIdx2 = temp.indexOf('-', baseIdx1);
    ports.addElement(temp.substring(baseIdx1, baseIdx2));
    ports.addElement(temp.substring(baseIdx2 + 1, temp.indexOf(';', baseIdx2)));
}

//Portnummern unseres Servers müssen eingetragen werden
private String replacePorts(String res, int firstOrSecond) {
    String response = res;
    String source = "source=";
    String ss = "ssrc=";
    int index = response.indexOf(source);
    String replaced = response.substring(0, index + source.length());
    index = response.indexOf(ss);
    String ssrc = response.substring(index);
    ssrc = ssrc.substring(5, 13);
    if (firstOrSecond == 1) {
        replaced = replaced + VideoMidlet.getLocalIP() + ";client_port=" +
                ports.elementAt(0) + "-" + ports.elementAt(1) + ";server_port=" +
                VideoMidlet.getConfigParameter().elementAt(1) + "-" +
                VideoMidlet.getConfigParameter().elementAt(2) + ";ssrc=" +
                ssrc + "\r\n\r\n";//element(3,4)
    }
```

```
        if (firstOrSecond == 2) {
            replaced = replaced + VideoMidlet.getLocalIP() + ";client_port=" +
                    ports.elementAt(2) + "-" + ports.elementAt(3) + ";server_port=" +
                    VideoMidlet.getConfigParameter().elementAt(2) + "-" +
                    VideoMidlet.getConfigParameter().elementAt(3) + ";ssrc=" +
                    ssrc + "\r\n\r\n";
        }
        return replaced;
    }

    //Portnummern des Clients müssen eingetragen werden
    private String replaceClientPorts(String res, int firstOrSecond) {
        int pos1 = res.indexOf("client_port=");
        int pos2 = res.indexOf(";", pos1);
                        return   res.substring(0,   pos1)   +   "client_port="   +   (String)
         VideoMidlet.getConfigParameter().elementAt(1) +
        "-" + (String) VideoMidlet.getConfigParameter().elementAt(2) + res.substring(pos2);
    }

    //IP ersetzen
    private String replaceIP(String toReplace, int serverOrPlayer) {
        int pos;
        if (serverOrPlayer == 0) {
            //das Ersetzen auf 127.0.0.1 im sdp im DESCRIBE führt zum Aufhängen der Kommunikation
            while ((pos = toReplace.indexOf(VideoMidlet.getServerIP())) != -1) {
                toReplace = toReplace.substring(0, pos) + VideoMidlet.getLocalIP() +
                        toReplace.substring(pos + VideoMidlet.getServerIP().length());
            }
        } else {
            //das Ersetzen auf 127.0.0.1 im sdp im DESCRIBE führt zum Aufhängen der Kommunikation
            while ((pos = toReplace.indexOf(VideoMidlet.getLocalIP())) != -1) {
                toReplace = toReplace.substring(0, pos) + VideoMidlet.getServerIP() +
                        toReplace.substring(pos + VideoMidlet.getLocalIP().length());
            }
        }
        return toReplace;
    }
}
```

# ProtocolHandler.java

```
import java.util.Vector;
import java.io.*;
import java.io.IOException;
import java.io.OutputStream;
import javax.microedition.io.*;

public class ProtocolHandler {

    private String address;
    private InputStream is;
    private OutputStream os;
    Vector tracks = new Vector(2);
    boolean stopped = false;
    Vector darwinPorts = new Vector();

    public ProtocolHandler(String address) throws IOException {
        this.address = address;//Die Adresse des Mediums
        String sckt = "socket://" + VideoMidlet.getServerIP() + ":554";
        SocketConnection sc = (SocketConnection) Connector.open(sckt);
        //Die erste Möglichkeit dei lokale IP auszulesen
        VideoMidlet.setLocalIP(sc.getLocalAddress());
        is = sc.openInputStream();
        os = sc.openOutputStream();
    }

    public String describe(String query) throws IOException {
        String response = answerQuery(query, 1);
        parseRateInformation(response);
        parseTrackInformation(response); //extrahiere die Tracks
        return response;
    }


    public String setup(String query) throws IOException {
        String response = answerQuery(query, 2);
        parsePortInformation(response);
        return response;
    }

    public String uniCommand(String query) throws IOException {
        String response = answerQuery(query, 4);
        return response;
    }
```

```
    private void parseTrackInformation(String response) {
        String localRef = response;
        String trackId = "";
        int index = localRef.indexOf("a=control:trackID=");
        while (index != -1) { //Finde alle Trackinformationen
            int baseIdx = index + "a=control:trackID=".length();
            trackId = localRef.substring(baseIdx, baseIdx + 1);
            localRef = localRef.substring(baseIdx + 1, localRef.length());
            index = localRef.indexOf("a=control:trackID=");
            tracks.addElement(trackId);
        }
    }

    //Portnummern extrahieren
    private void parsePortInformation(String response) {
        String temp = response;
        int index = temp.indexOf("server_port=");
        int baseIdx1 = index + ("server_port=").length();
        int baseIdx2 = temp.indexOf('-', baseIdx1);
        darwinPorts.addElement(temp.substring(baseIdx1, baseIdx2));
        darwinPorts.addElement(temp.substring(baseIdx2 + 1, temp.indexOf(';', baseIdx2)));
    }

    private void parseRateInformation(String response) {
        //Es wird vorausgesetzt, dass Video über RTP-Payload
        //Type 96 übertragen wird
        int index = response.indexOf("a=rtpmap:96");
        int index1 = response.indexOf("/",index);
        int index2 = response.indexOf("\r\n", index1);
        String rate = response.substring(index1+1, index2);
        VideoMidlet.setRTPRate(Integer.parseInt(rate));
    }

    private String answerQuery(String command, int what) throws IOException {
        String response;
        VideoMidlet.addRTSPServerData("\nQUERY: " + command);
        os.write((command).getBytes());
        os.flush(); //Ohne Flush werden die Daten nicht verschickt

        byte b[] = new byte[2048];
        int c, i = 0, end = 0;
        boolean semikolon = false;
        int mark = 0;
        while ((c = is.read()) != -1) {
            b[i++] = (byte) c;
            if (mark == 0x0a && c == 0x0d && what != 1) {//wenn zwei Mal \r\n dann fertig
                b[i++] = (byte) is.read();//letztes \n einlesen
                break;
            }
            if (what == 1 && mark == 0x3b) {//wenn ";"
                semikolon = true;
            }
            if (semikolon && mark == 0x0d && c == 0x0a) {//wenn "\r\n" nach ';'
                end++;
                semikolon = false;
            }
             //Das zweite Mal wenn "\r\n" nach ';'
            if(end==Integer.parseInt((String)VideoMidlet.getConfigParameter().elementAt(5))+ 1){
                break;
            }
            mark = c;
        }

        response = new String(b, 0, i);
        VideoMidlet.addRTSPServerData("\nRESPONSE: " + response);
        b = null;
        if (!response.startsWith("RTSP/1.0 200 OK")) {
            throw new IOException("Server returned invalid code: " + response);
        }
        return response;
    }
}
```

# RTPServer.java

```
import java.io.IOException;
import javax.microedition.io.*;

public class RTPServer implements Runnable {

    private String playerAddress;
    private String serverAddress;
    private boolean stop;
    private boolean h264;
```

```
    private DatagramConnection dcRTP;
    private Datagram dg;
    private Thread t;

    public RTPServer(String playerAddress, String serverAddress, boolean h264) {
        this.playerAddress = playerAddress;
        this.serverAddress = serverAddress;
        this.h264 = h264;
        try {
            dcRTP = (DatagramConnection) Connector.open("datagram://:" +
                    (String) VideoMidlet.getConfigParameter().elementAt(1));
        } catch (IOException ex) {
            VideoMidlet.addErrorData("\nEXCEPTION in RTPServer()" + ex.toString());
        }
    }

    public void start() {
        stop = false;
        t = new Thread(this);
        t.start();
    }

    public void run() {
        MPEGInfoSaver mpegInfoSaver = null;
        H264InfoSaver h264InfoSaver = null;
        if (!h264) {
            mpegInfoSaver = new MPEGInfoSaver();
        } else if (h264) {
            h264InfoSaver = new H264InfoSaver();
        }
        try {
            while (!stop) {
                dg = dcRTP.newDatagram(2048);
                dcRTP.receive(dg);
                //Adresse der Gegenstelle setzen
                if (dg.getAddress().equals(serverAddress)) {
                    dg.setAddress(playerAddress);
                    dcRTP.send(dg);
                    if (!h264) {
                        //Informationen speichern
                        mpegInfoSaver.save(dg.getData());
                    } else if (h264) {
                        h264InfoSaver.save(dg.getData(), dg.getLength());
                    }
                } else if (dg.getAddress().equals(playerAddress)) {
                    dg.setAddress(serverAddress);
                }
                dcRTP.send(dg);
            }
        } catch (Exception ioe) {
            VideoMidlet.addErrorData("\n!!!EXCEPTION in RTPServer.run(): " + ioe.toString());
        }
    }

    public void stop() {
        stop = true;
        try {
            dcRTP.close();
            dg = null;
            t.interrupt();
        } catch (IOException ex) {
            VideoMidlet.addErrorData("\n!!!EXCEPTION in RTPServer.stop():" + ex.toString());
        }
    }
}
```

# RTCPServer.java

```
import java.io.IOException;
import javax.microedition.io.*;

public class RTCPServer extends Thread {
    private String serverAddress;
    private String playerAddress;
    private boolean stop;
    private Thread t;
    private DatagramConnection dcRTCP;
    private Datagram dg = null;

    public RTCPServer(String serverAddress, String playerAddress) {
        this.serverAddress = serverAddress;
        this.playerAddress = playerAddress;
        try {
            dcRTCP = (DatagramConnection) Connector.open("datagram://:" +
                    (String) VideoMidlet.getConfigParameter().elementAt(2));
```

```
        } catch (IOException ex) {
            VideoMidlet.addErrorData("\n!!!EXCEPTION in RTCPServer()" + ex.toString());
        }
    }

    public void start() {
        stop = false;
        t = new Thread(this);
        t.start();
    }

    public void run() {
        try {
            while (!stop) {
                dg = dcRTCP.newDatagram(2048);
                dcRTCP.receive(dg);
                if (dg.getAddress().equals(serverAddress)) {
                    dg.setAddress(playerAddress);
                } else if (dg.getAddress().equals(playerAddress)) {
                    dg.setAddress(serverAddress);
                }
                dcRTCP.send(dg);
            }
        } catch (Exception ioe) {
            VideoMidlet.addErrorData("!!!Exception in RTCPServer.run(): "  + ioe.toString());
        }
    }

    public void stop() {
        stop = true;
        try {
            dcRTCP.close();
            dg = null;
            t.interrupt();
        } catch (Exception e) {
            VideoMidlet.addErrorData("!!!EXCEPTION in RTCPServer.stop():"+ e.toString());
        }

    }
}
```

# MPEGInfoSaver.java

```
import java.util.Date;
import java.util.Vector;

public class MPEGInfoSaver extends Thread {

    private byte[] b;
    private boolean stop = false;

    public MPEGInfoSaver() {
        b = null;
        start();
    }

    public synchronized void save(byte[] b) {
        this.b = b;
        notify();
    }

    public synchronized void run() {
        int size;
        while (stop == false) {
            if (b == null) {
                try {
                    wait();
                } catch (Exception e) {
                    VideoMidlet.addErrorData("\nException im DataSender.run(): 1"+e.toString());
                }
            }

            try {
                synchronized(VideoMidlet.getObj()){
                //Speichern in korrekter Reihenfolge
                MPEGInfo mpegInfo = getInfo();
                size = VideoMidlet.getMpegData().size();
                if(size == 0)VideoMidlet.addMpegData(mpegInfo);
                else{
                    while(size > 0 && ((MPEGInfo)VideoMidlet.getMpegData().elementAt(size-
                                                1)).getSeqNo() > mpegInfo.getSeqNo())
                        size--;
                    VideoMidlet.insertMpegData(mpegInfo, size);
                    VideoMidlet.notifyObj();
```

```
                }
              }
            } catch (Exception e) {
                VideoMidlet.addErrorData("\nException im DataSender.run(): 2" + e.toString());
            }
            b = null;
        }
    }

    //Extrahieren der Daten
    private MPEGInfo getInfo() {
        MPEGInfo mpegInfo = new MPEGInfo();
        try {
            mpegInfo.setIncomeTime(new Date().getTime());
            mpegInfo.setSeqNo((int) (((b[2] & 0xff) << 8) | (b[3] & 0xff)));
            mpegInfo.setTimeStamp((long) (((b[4] & 0xff) << 24) | ((b[5] & 0xff) << 16) | ((b[6]
                                                                & 0xff) << 8) | (b[7] & 0xff)));
             mpegInfo.setSSRC((long) (((b[8] & 0xff) << 24) | ((b[9] & 0xff) << 16) | ((b[10] &
                                                                  0xff) << 8) | (b[11] & 0xff)));
            Vector vops = new Vector();
            Vector objects = new Vector();
            for (int i = 12; i < (b.length - 5); i++) {
                //Startsequenz
                 if((((b[i] & 0xff) << 16) | ((b[i + 1] & 0xff) << 8) | (b[i + 2] & 0xff))== 1) {
                    //Alle Objekte im Paket. Keine Unterscheidung an der Stelle (performance)
                    objects.addElement(new Integer(b[i + 3] & 0xff));
                    if ((b[i + 3] & 0xff) == 0xb6) { //vop
                        //Keine Unterscheidung
                        vops.addElement(new Integer((b[i + 4] & 0xff) >> 6));
                    }
                }
            }
            mpegInfo.setVops(vops);
            mpegInfo.setObjects(objects);
        } catch (Exception e) {
            VideoMidlet.addErrorData("!!!\nException in getInfo()" + e.toString());
        }
        return mpegInfo;
    }

    public void stop() {
        stop = true;
        save(null);// Damit der Thread aus dem wait() rauskommt
    }
}
```

# H264InfoSaver.java

```
import java.util.Date;
import java.util.Vector;

public class H264InfoSaver extends Thread {

    private byte[] b;
    private int length;
    private boolean stop = false;

    public H264InfoSaver() {
        b = null;
        start();
    }

    public synchronized void save(byte[] b, int length) {
        this.b = b;
        this.length = length;
        notify();
    }

    public synchronized void run() {
        H264Info h264Info;
        int size;
        while (stop == false) {
            if (b == null) {
                try {
                    wait();
                } catch (Exception e) {
                   VideoMidlet.addErrorData("\n!!!Exception im DataSender.run():1"+e.toString());
                }
            }

            try {
                synchronized (VideoMidlet.getObj()) {
                    //Speichern in korrekter Reihenfolge
                    h264Info = getInfo();
                    size = VideoMidlet.getMpegData().size();
```

```
                if (size == 0) {
                    VideoMidlet.addMpegData(h264Info);
                } else {
                    while (size > 0 && ((H264Info) VideoMidlet.getMpegData().
                            elementAt(size - 1)).getSeqNo() > h264Info.getSeqNo()) {
                        size--;
                    }
                    VideoMidlet.insertMpegData(h264Info, size);
                    VideoMidlet.notifyObj();
                }
            }
        } catch (Exception e) {
            VideoMidlet.addErrorData("\n!!!Exception im DataSender.run(): 2" +
                                     e.toString());
        }
        b = null;
    }
}

//Extrahieren der Daten
public H264Info getInfo() {
    H264Info h264Info = new H264Info();
    try {
        H264NALU nalu = new H264NALU();
        Vector nalus = new Vector();
        h264Info.setIncomeTime(new Date().getTime());
        h264Info.setSeqNo((int) (((b[2] & 0xff) << 8) | (b[3] & 0xff)));
        h264Info.setTimeStamp((long) (((b[4] & 0xff) << 24) | ((b[5] & 0xff)
                            << 16) | ((b[6] & 0xff) << 8) | (b[7] & 0xff)));
        h264Info.setSSRC((long) (((b[8] & 0xff) << 24) | ((b[9] & 0xff) <<
                            16) | ((b[10] & 0xff) << 8) | (b[11] & 0xff)));
        //H264 Auswertung
        //An erster Stelle im RTP-Payload muss ein NAL-Header stehen
        //Er ist 1Byte groß (b[12])
        nalu.setFBit((b[12] & 0xff) >> 7);
        nalu.setNri((b[12] & 0xff) >> 5);
        nalu.setType(b[12] & 0x1f);
        //wenn Elemente des VCL mit slice-header()
        if (nalu.getType() == 1 || nalu.getType() == 2 || nalu.getType() == 5) {
            nalu.setSliceType(getSliceType(13));
        }
        nalus.addElement(nalu);
        //STAP-A
        int naluSize = 0;
        int i = 13;
        if (nalu.getType() == 24) {
            while (i < length) {
                nalu = new H264NALU();
                //die Nächten 2 Byte sind die Länge der nächsten
                //NALU einschliesslich dem Header
                naluSize = (((b[i] & 0xff) << 8) | (b[i + 1] & 0xff));
                i = i + 2;
                //NALU-Header
                nalu.setFBit((b[i] & 0xff) >> 7);
                nalu.setNri((b[i] & 0xff) >> 5);
                nalu.setType(b[i] & 0x1f);
                //wenn Elemente des VCL mit slice-header()
                if (nalu.getType() == 1 || nalu.getType() == 2 || nalu.getType() == 5) {
                    nalu.setSliceType(getSliceType(i + 1));
                }
                nalus.addElement(nalu);
                //Die nächste NALU
                i = i + naluSize;
            }
        } //STAP-B
        else if (nalu.getType() == 25) {
            boolean packetBeginn = true;//Erforderlich für STAP-B
            nalu = new H264NALU();
            nalu.setDon(((b[i] & 0xff) << 8) | (b[i + 1] & 0xff));
            if (packetBeginn) {//Die Länge der DON überspringen
                i = i + 2;
                packetBeginn = false;
            }
            naluSize = ((b[i] & 0xff) << 8) | (b[i + 1] & 0xff);
            nalu.setFBit((b[i + naluSize] & 0xff) >> 7);
            nalu.setNri((b[i + naluSize] & 0xff) >> 5);
            nalu.setType(b[i + naluSize] & 0x1f);
            nalus.addElement(nalu);
            i = i + naluSize;//zur Nächsten NALU springen
        } //MTAP16
        else if (nalu.getType() == 26) {
            //To do
        } //MTAP24
        else if (nalu.getType() == 27) {
            //To do
        } //FU-A
        else if (nalu.getType() == 28) {
```

```
            int startBit = (b[13] & 0xff) >> 7;
            //Wenn es der Anfang der NALU ist, speichern
            if (startBit == 1) {
                nalu = new H264NALU();
                nalu.setSliceType(b[13] & 0x1f);
                if (nalu.getType() == 1 || nalu.getType() == 2 || nalu.getType() == 5) {
                    nalu.setSliceType(getSliceType(14));
                }
            }
        } //FU-B
        else if (nalu.getType() == 29) {
            int startBit = (b[13] & 0xff) >> 7;
            //Wenn es der Anfang der NALU ist, speichern
            if (startBit == 1) {
                nalu = new H264NALU();
                nalu.setDon((((b[14] & 0xff) << 8) | (b[15] & 0xff)));
                nalu.setType(b[13] & 0x1f);
                if (nalu.getType() == 1 || nalu.getType() == 2 || nalu.getType() == 5) {
                    nalu.setSliceType(getSliceType(16));
                }
            }
        }
        h264Info.setNalus(nalus);
    } catch (Exception e) {
        VideoMidlet.addErrorData("Exception in MPEGInfo" + e.toString());
    }
    return h264Info;
}

//Holen des Typs der Slice
private int getSliceType(int offset) {
    int count = 0;
    int zeros = 0;
    int value = 0;
    byte tempValue = 0x00;
    final byte masks[] = {0x00, 0x01, 0x03, 0x07, 0x0f, 0x1f, 0x3f};
    try {
        for (int k = offset; k < b.length; k++) {
            for (byte i = 7; i >= 0; i--) {//Ein Byte nach einer 1 durchsuchen
                byte l = (byte) (((b[k] & 0xff) >> i) & 0x01);
                //in diesen Block nur eintreten wenn noch
                //keine 1 vorgekommen ist und es nicht der
                //nächste Teil der tempValue ist
                if (l == 0x01 && tempValue == 0) {
                    if (zeros <= i) {
                        value = (((b[k] & 0xff) >> (i - zeros)) & masks[zeros + 1]) - 1;
                        count++;
                        zeros = 0;
                        //die Methode verlassen wenn der zweite
                        //ue(v) Wert eingelesen wurde (slice_type)
                        if (count == 2) {
                            return value;
                        }
                        continue;//erster Wert erfasst
                    } else if (zeros > i) {
                        //die noch zu lesenden Bits
                        zeros = zeros - i;
                        //in diesem Byte die restlichen Bits einlesen
                        tempValue = (byte) ((b[k] & masks[i + 1]) - 1);
                        //in diesem Byte ist nichts mehr zum Holen
                        break;
                    }
                }

                if (tempValue != 0) {
                    //wenn  alle noch benöntigten Bits
                    //in diesem Byte vorhanden sind
                    if (zeros < 8) {
                        //Platz schaffen für die restlichen Bits:
                        value = (tempValue & 0xff) << zeros;
                        //Den Platz füllen
                        value = (value | ((b[k] & 0xff) >> (8 - zeros))) - 1;
                        VideoMidlet.addErrorData("tempvalue" + value);
                        count++;
                        if (count == 2) {
                            return value;
                        }
                        //von vorne an anfangen
                        tempValue = 0;
                        zeros = 0;
                        //nächstes Byte holen
                        break;
                    }
                    if (zeros > 8) {
                    }
                }
```

```
                    //Anzahl der Nullen muss bekannt sein um die Anzahl der
                    //Stellen nach der 1 auszuwerten (Exp-Golomb code)
                    //Inkrementiert solange wie
                    if (tempValue == 0) {
                        zeros++;
                    }
                }

            }
        } catch (Exception e) {
            VideoMidlet.addErrorData("\n!!!Exception in getSliceType(): " + e.toString());
        }
        return value;
    }

    public void stop() {
        stop = true;
        save(null, 0);// Damit der Thread aus dem wait() rauskommt
    }
}
```

# MPEGInfo.java

```
import java.util.Vector;

public class MPEGInfo extends Thread {

    private int seqNo;
    private long timeStamp; //long um negative Zahlen zu vermeiden
    private long SSRC;
    private short payloadType;
    private Vector vops;
    private Vector objects;
    private long incomeTime;

    public MPEGInfo() {
    }

    public int getSeqNo(){
        return seqNo;
    }

    public long getTimeStamp(){
        return timeStamp;
    }

    public long getSSRC(){
        return SSRC;
    }

    public short getPayloadType(){
        return payloadType;
    }

    public Vector getVops(){
        return vops;
    }

    public Vector getObjects(){
        return objects;
    }

    public long getIncomeTime(){
        return incomeTime;
    }

    public void setSeqNo(int seq){
        seqNo = seq;
    }

    public void setTimeStamp(long time){
        timeStamp = time;
    }

    public void setSSRC(long s){
        SSRC = s;
    }

    public void setPayloadType(short t){
        payloadType = t;
    }

    public void setVops(Vector v){
        vops = v;
    }
```

```
    public void setObjects(Vector v){
        objects = v;
    }

    public void setIncomeTime(long time){
        incomeTime = time;
    }
}
```

# H264Info.java

```
import java.util.Vector;

public class H264Info extends Thread {

    private int seqNo;
    private long timeStamp; //long um negative Zahlen zu vermeiden
    private long SSRC;
    private short payloadType;
    private H264NALU nalu;
    private Vector nalus;
    private long incomeTime;

    public int getSeqNo(){
        return seqNo;
    }

    public long getTimeStamp(){
        return timeStamp;
    }

    public long getSSRC(){
        return SSRC;
    }

    public short getPayloadType(){
        return payloadType;
    }

    public Vector getNalus(){
        return nalus;
    }

    public H264NALU getNalu(){
        return nalu;
    }

    public long getIncomeTime(){
        return incomeTime;
    }

    public void setSeqNo(int seq){
        seqNo = seq;
    }

    public void setTimeStamp(long time){
        timeStamp = time;
    }

    public void setSSRC(long s){
        SSRC = s;
    }

    public void setPayloadType(short t){
        payloadType = t;
    }

    public void setNalus(Vector v){
        nalus = v;
    }

    public void setNalu(H264NALU n){
        nalu = n;
    }

    public void setIncomeTime(long time){
        incomeTime = time;
    }
}
```

## H264NALU.java

```
    public class H264NALU {
        private int fBit;
        private int nri;
        private int type;
        private int don;
        private int sliceType;

        public int getFBit(){
            return fBit;
        }

        public int getNri(){
            return nri;
        }

        public int getType(){
            return type;
        }

        public int getDon(){
            return don;
        }

        public int getSliceType(){
            return sliceType;
        }

        public void setFBit(int b){
            fBit = b;
        }

        public void setNri(int n){
            nri = n;
        }

        public void setType(int t){
            type = t;
        }

        public void setDon(int d){
            don = d;
        }

        public void setSliceType(int st){
            sliceType = st;
        }
    }
```

## MP4IntervalState.java

```
public class MP4IntervalState {

    private long intervalStart;

    private int missedPackets;

    private int allMissedPackets;

    private int missedPercent;

    private long mid_jitter;

    private long max_jitter;

    private long jitter;

    private int  i;

    private int p;

    private int b;


    public MP4IntervalState(long intervalStart, int missedPackets,

             int allMissedPackets, int missedPercent, long mid_jitter,

            long max_jitter, long jitter, int i, int p, int b){

        this.intervalStart = intervalStart;

        this.missedPackets = missedPackets;
```

```
        this.allMissedPackets = allMissedPackets;
        this.missedPercent = missedPercent;
        this.mid_jitter = mid_jitter;
        this.max_jitter = max_jitter;
        this.jitter = jitter;
        this.i = i;
        this.p = p;
        this.b = b;
    }
    public long getIntervalStart(){
        return intervalStart;
    }
    public int getMissedPackets(){
        return missedPackets;
    }
    public int getAllMissedPackets(){
        return allMissedPackets;
    }
    public long getMissedPercent(){
        return missedPercent;
    }
    public long getMidJitter(){
        return mid_jitter;
    }
    public long getMaxJitter(){
        return max_jitter;
    }
    public long getJitter(){
        return jitter;
    }
    public long getIUnits(){
        return i;
    }
    public long getPUnits(){
        return p;
    }
    public long getBUnits(){
        return b;
    }
}
```