Simulation des Real-Time Streaming Protocol (RTSP) mit OMNeT++

Arndt Buschmann

6. Oktober 2002

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Studienarbeit

Simulation des Real-Time Streaming Protocol (RTSP) mit OMNeT++

von
cand. inform. Arndt Buschmann

Aufgabenstellung und Betreuung:
Prof. Dr. L. Wolf und Dipl.-Inform. V. Kahmann

Braunschweig, den 6. Oktober 2002

Erklärung

Ich versichere, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Braunschweig, den 6. Oktober 2002

Kurzfassung

In dieser Arbeit wird ein Simulationsmodell des Real-time Streaming Protocol (RTSP) für die Simulationsumgebung OMNeT++ entwickelt. Das Modell soll als Basismodell für eine Proxy-Architektur, welche Streaming-Anwendungen in kooperativen Umgebungen ermöglichen soll, dienen.

Das zugrundeliegende Protokoll RTSP beschrieben und die OMNeT++-Simulationsumgebung vorgestellt. Aufbauend auf diesen Erkenntnissen wird anschließend eine Modulstruktur für dieses Simulationsmodell entwickelt und mit Hilfe der Programmiersprache C++ implementiert.

[Hier wird später die Aufgabenstellung eingefügt.]

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Multimediale Anwendungen im Internet gewinnen immer stärker an Bedeutung, insbesondere die Übertragung von Datenströmen unterschiedlichster Art in Echtzeit. Neue Anwendungen, wie Media-on-Demand, Videokonferenzen, Präsentationen oder auch Audioübertragungen, benötigen eine kontinuierliche Datenübertragung über das Netz. Dies ist nur durch die Kombination verschiedener Internetprotokolle möglich. Eine mögliche Kombination von Protokollen wäre das Resource Reservation Protocol (RSVP) für die Reservierung der Netzwerkressourcen des Datenstroms, das Real-time Transport Protocol (RTP) für den Transport der Daten und das Real-time Streaming Protocol (RTSP) als Signalisierungsprotokoll. [1]
RTSP ist in der Anwendungsschicht angesiedelt und steuert die Übertragung kontinuierlicher Medien wie Audio- oder Videodateien. Das Protokoll bietet Funktionen wie Play, Stop, Pause und ermöglicht so Anwendungen eine direkte Kontrolle der Medien auf dem Server.
Die eigentliche Übertragung der Nutzdaten wird nicht direkt von RTSP übernommen, nur in Ausnahmefällen ist es möglich Nutzdatenpakete in RTSP Datenpakete zu kapseln. Normalerweise benutzt RTSP andere Protokolle um die Nutzdaten zu übertragen. Dabei ist RTSP nicht auf ein Protokoll festgelegt, sondern erlaubt die Übertragung der Daten über verschiedene Protokolle vorzunehmen, beispielsweise UDP oder das bereits erwähnte RTP.

Um das Verhalten von Kommunikationsprotokollen in verteilten Systemen zu prüfen, bietet es sich an, Protokolle mittels einer Simulationsumgebung zu evaluieren. Durch die Simulation eines Protokolls kann sein Verhalten unter veränderbaren Bedingungen geprüft und die Ergebnisse können als Ausgangspunkt für mögliche Erweiterungen verwendet werden.

1.1 Zielsetzung der Arbeit

Im Rahmen dieser Studienarbeit wird ein RTSP-Simulationsmodell für die Simulationsumgebung OMNeT++ entwickelt. Das Modell soll als Basismodell für eine Proxy-Architektur, welche Streaming-Anwendungen in kooperativen Umgebungen ermöglichen soll, dienen. Die wesentlichen Aufgaben dieser Arbeit bestehen darin

- die Protokollabläufe bei Client und Server zu modellieren
- und ein vorhandenes Simulationsmodell für die RTP-Simulation einzubetten

Das Simulationsmodell baut dabei auf die in [2] entwickelte RTPLayer- und die in [3] erstellte SocketLayer-Bibliothek auf.

1.2 Gliederung der Arbeit

Zunächst wird in Kapitel 2 das Protokoll RTSP entsprechend RFC 2326 [4] vorgestellt. Dabei wird auf dem auch im Simulationsmodell implementierten Zustandsautomaten besonderen Wert gelegt.

In Kapitel 3 wird ein Überblick über die Simulationsumgebung OMNeT++ und ihre Funktionsweise gegeben. In den folgenden beiden Kapiteln 4 und 5 wird die Modulstruktur der OMNeT++-Module und deren Implementation in C++ beschrieben. Die Beschreibung der Implementation beinhaltet einen grundlegenden Überblick über die Klassenstruktur.

Das abschließende 6. Kapitel fasst die wesentlichen Aspekte der Arbeit noch einmal zusammen. Ebenfalls werden Anregungen zur weiteren Ergänzung des Modells gegeben.

2 Das Real-time Streaming Protocol

Das Real-time Streaming Protocol (RTSP) wurde gemeinsam von RealNetworks, Netscape und der Universität von Columbia entwickelt. Dabei wurde auf die Erfahrungen der Produkte von RealNetworks RealAudio und Netscapes LiveMedia zurückgegriffen. Der erste Entwurf des Protokolls wurde 1996 der Internet Engineering Task Force (IETF) vorgestellt und schließlich im April 1998 im RFC 2326 [4] veröffentlicht.

2.1 Eigenschaften

RTSP ist ein Protokoll der Anwendungsschicht. Die Aufgabe des Protokolls besteht in der Steuerung einer Übertragung kontinuierlicher Daten. Dabei kann es sich um Liveübertragungen oder um die Übertragung von gespeicherten Audio- oder Videodaten handeln, zur Aufnahme oder Wiedergabe. Das Protokoll ist darauf ausgelegt mehrere Sitzungen (Sessions) zu steuern und auf verschiedene Protokolle aufzusetzen, die die eigentliche Übertragung der Daten übernehmen. Dies können UDP oder Mulicast-UDP, TCP oder auch RTP (RFC 1889) [5] sein.

Jede Präsentation und jeder Medienstrom kann anhand einer RTSP-URL identifiziert werden. Bevor eine Verbindung per RTSP aufgebaut werden kann, müssen dem Client einige Informationen, wie die Adresse des Servers, die Kodierung der Daten, Inhaltsbeschreibungen usw., bekannt sein. Diese Informationen sind in Präsentationsbeschreibungen (Presentation descriptions) abgelegt und können mit dem Hyper Text Transport Protocol (HTTP) oder einem ähnlichen Protokoll abgefragt werden. RTSP ist nicht auf ein spezielles Format einer Präsentationsbeschreibung festgelegt. Ein Beispiel eines solchen Formats wäre das Session Description Format (SDP, RFC 2327) [6].
Eine Präsentationsbeschreibung kann dabei mehrere Präsentationen beinhalten, die wiederum aus verschiedenen Medienströmen bestehen. Die Mediendaten können sich auf unterschiedlichen Servern, z.B. zur Lastverteilung die Videodaten auf einem Server und auf einem anderen die Audiodaten, befinden.

Neben diesen Parametern müssen auch die Zieladresse und der Port festgelegt werden. Dazu gibt es verschiedene Möglichkeiten. Bei einer Unicast-Übertragung werden die Daten an die Quelle der RTSP-Anfrage, mit einem vom Client gewählten Port, gesendet. Bei einer Mulicast-Übertragung besteht die Möglichkeit, dass der Server oder aber auch der Client die Multicast-Adresse und Port wählen. Die Adresse serverseitig zu wählen, käme typischerweise bei einer Liveübertragung zur Anwendung. Der Server bestimmt eine geeignete Multicastadresse und sendet an diese die Daten. Die Clients haben dabei eine passive Rolle; sie erhalten nur jene Daten, die zur Laufzeit gesendet werden. Bereits übertragene Daten können nicht erneut betrachtet werden. Bei einer nicht öffentlichen Livekonferenz hingegen, kann die Multicast-Adresse und der Port aus einer Präsentationsbeschreibung an den Server gesendet werden. Die Adresse wird also clientseitig ausgewählt.

Von seiner Struktur her ähnelt das Real-time Streaming Protocol dem Protokoll HTTP/1.1 (RFC 2068) [7]. Es ist ein textbasiertes Protokoll und verfügt über eine große Anzahl an Parametern und Methoden. Die Nachrichten (Messages) können mit einem einfachen MIME-Parser ausgewertet werden. Nicht implementierte Methoden bzw. Parameter werden von Server bzw. Client verworfen. Damit ist RTSP einfach erweiterbar und kann an aktuelle Entwicklungen angepasst werden.
In einigen wichtigen Aspekten unterscheidet sich RTSP von HTTP:

- HTTP unterstützt nur eine einseitige Kommunikation. Der Client fordert Daten vom Server an, dieser liefert sie aus und wartet auf weitere Instruktionen. Bei RTSP ist jedoch eine beidseitige Kommunikation möglich. Während der Übertragung kann so vom Server die Auslieferungsadresse geändert werden oder der Server kann Informationen über die vom Client unterstützten Methoden anfordern.
- RTSP ist im Gegensatz zum zustandslosen HTTP ein zustandsorientiertes Protokoll. Client und Server bedienen sich hierzu einer Zustandsmaschine (siehe auch Kapitel 2.4 auf Seite 14).
- Die Nutzdaten werden mittels eines anderen Protokolls übertragen (Out-of-band Übertragung). Dadurch ergibt sich eine höhere Flexibilität.
- RTSP verwendet im Gegensatz zu HTTP absolute URIs.

2.2 RTSP-Nachrichten

Während einer Sitzung werden zwischen dem Client und Server Nachrichten ausgetauscht. Nachrichten können entweder Anfragen (Requests) oder Antworten (Responses) sein. Mittels eines Requests kann der Server beispielsweise angewiesen werden eine bestimmte Datei abzuspielen. Als Response würde der Client Statusmeldungen vom Server erhalten.

2.2.1 Format

RTSP ist ein textbasiertes Protokoll und benutzt den ISO 10646 Zeichensatz mit UFT-8 Kodierung (RFC 2279) [8]. Zeilenumbrüche werden mit CRLF eingeleitet, der Empfänger sollte aber auch CR und LF einzeln als Zeilenumbruch interpretieren. Zum Parsen können also Standard MIME- oder HTTP-Parser eingesetzt werden. Die RTSP-Nachrichten können durch alle anderen Protokolle unterer Schichten, welche 8-bit persistent sind, getragen werden.

2.2.2 Aufbau

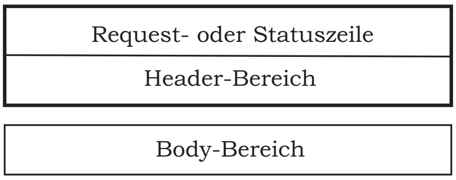

Abbildung 2.1: Aufbau einer RTSP-Nachicht.

Alle Nachrichten, sowohl RTSP-Requests als auch RTSP-Responses, sind nach demselben Schema aufgebaut. Die Nachrichten bestehen aus einer Request- bzw. einer Statuszeile, gefolgt von einem Headerblock und dem Body-Bereich.

Abbildung 2.2: Aufbau der Requestzeile.

Die Requestzeile ist in drei Teile, die voneinander per Leerzeichen getrennt sind, gegliedert (Abb. 2.2). Der erste Teil beinhaltet die Methode, anschließend folgt die Request-URI und die RTSP-Version. Die Methode bezieht sich auf die angegebene URI. Sie beschreibt Aktionen, die der Partner auf die Quelle anwenden kann. Nicht alle existierenden Methoden müssen tatsächlich bei einer Implementierung umgesetzt werden. Unbekannte Methoden können ignoriert und mittels eines Errorcodes quittiert werden. Durch die Angabe der Versionsnummer wird sichergestellt, dass die Gegenseite die Methoden bzw. Parameter richtig interpretiert. Auf die Methoden im Einzelnen wird in Abschnitt 2.3 auf Seite 8 eingegangen.

Abbildung 2.3: Aufbau der Statuszeile.

Die Statuszeile enthält ebenfalls drei von einem Leerzeichen getrennte Felder (Abb. 2.3). Am Anfang wird die RTSP-Version angegeben, anschließend folgen der Statuscode und die Statusbeschreibung. Der Statuscode ist als eine dreistellige Zahl codiert. Die Statusbeschreibung enthält diese Information noch einmal im Klartext in einer für den Anwender verständlichen Form. Für den Client selbst ist nur der Statuscode wichtig. Anhand der ersten Ziffer kann der Client die Nachricht klassifizieren:

- 1xx: Informational - Nur informativen Charakter, der Request ist angekommen und wird verarbeitet.
- 2xx: Success - Die Aktion wurde verstanden und erfolgreich ausgeführt.
- 3xx: Redirection - Um den Request erfolgreich abzuschließen, sind weitere Aktionen erforderlich.

- 4xx: Client Error - Beim Client ist ein Fehler aufgetreten.
- 5xx: Server Error - Beim Server ist ein Fehler aufgetreten.

Parameter	Typ	Art	Methoden
Accept	Request	optional	Body
Accept-Encoding	Request	optional	Body
Accept-Language	Request	optional	alle
Allow	Response	optional	alle
Authorization	Request	optional	alle
Bandwidth	Request	optional	alle
Blocksize	Request	optional	nicht OPTIONS, TEARDOWN
Cache-Control	Request, Response	optional	SETUP
Conference	Request	optional	SETUP
Connection	Request, Response	benötigt	alle
Content-Type	Response	benötigt	entity
CSeq	Request, Response	benötigt	alle
Date	Request, Response	optional	alle
From	Request	optional	alle
If-Modified-Since	Request	optional	DESCRIBE, SETUP
Proxy-Require	Request	benötigt	alle
Public	Response	optional	alle
Range	Request, Response	optional	PLAY, PAUSE, RECORD
Referer	Request	optional	alle
Require	Request	benötigt	alle
Retry-After	Response	optional	alle
RTP-Info	Response	benötigt	PLAY
Scale	Request, Response	optional	PLAY, RECORD
Session	Request	benötigt	alle, außer SETUP, OPTIONS
Server	Response	optional	alle
Speed	Request, Response	optional	PLAY
Transport	Request, Response	benötigt	SETUP
Unsupported	Response	benötigt	alle
User-Agent	Request	optional	alle
Via	Request, Response	optional	alle
WWW-Authent.	Request	optional	alle

Tabelle 2.1: Die Parameter der RTSP-Requests und -Responses.

Der folgende Header-Bereich enthält Parameter (vgl. Tabelle 2.1), die im Bezug auf die angegebene Methode sinnvoll sind. Sie werden mittels Zeilenumbruch voneinander getrennt. Falls ein Parameter mehrere Informationen trägt, trennt ein Semikolon-Zeichen (;) die einzelnen Abschnitte. Besonders wichtig sind die Parameter:

- **Transport**
  Dieser Parameter gibt an, welches Transport-Protokoll benutzt werden

soll. Ebenfalls beschreibt er dessen Parameter, wie Zieladresse, Kompression, Multicast time-to-live und Zielport.

- **Cseq**
  Das Cseq-Feld enthält die Sequenznummer des Requests. Die korrespondierende Response enthält dieselbe Nummer. So kann eine Response einem Request eindeutig zugeordnet werden. Bei Übertragungswiederholungen wird stets die gleiche Sequenznummer wie in der Ausgangsnachricht verwendet.

- **Session**
  Die Session-ID wird zufällig vom Server generiert und kennzeichnet eindeutig eine RTSP-Session. Anhand dieser ID ist es möglich eine Nachricht einer Session eindeutig zuzuordnen, unerlässlich wenn verschiedene Anfragen von einem Client für die gleiche URL den Server erreichen.

- **RTP-Info**
  Mit Hilfe dieses Parameters ist es möglich, dem Client (als Response bei einem Play-Request) RTP-spezifische Parameter zu übermitteln.

Der Body-Bereich ist optional und muss, falls vorhanden, im Header mit angegeben und mit einer Leerzeile eingeleitet werden. Er kann Daten anderer Protokolle, in Ausnahmefällen auch die Nutzdaten, aufnehmen.

2.3 Methoden

Die RTSP-Methode gibt an, welche Operation auf die angegebene Request-URI angewendet werden soll. Eine minimale Implementierung muss die Methoden ANNOUNCE, SETUP, PLAY oder RECORD und TEARDOWN zwingend verarbeiten können. Im Folgenden werden die Methoden der aktuellen Version RTSP/1.0 kurz beschrieben und deren Funktion jeweils durch ein kurzes Beispiel aufgezeigt.

2.3.1 OPTIONS

Mit Hilfe der OPTIONS-Methode kann herausgefunden werden, welche Methoden die Gegenseite unterstützt. Da nicht jede Implementierung, wie bereits erwähnt, alle Methoden unterstützen muss, ist es durch diese Methode möglich eine Liste der implementierten Methoden zu bekommen. OPTIONS bezieht sich als einzige Methode nicht auf die angegebene RTSP-URI, stattdessen wird der "*" als Platzhalter verwendet.

Beispiel:

```
C→S:  OPTIONS * RTSP/1.0
      CSeq: 1
      Require: implicit-play
      Proxy-Require: gzipped-messages

S→C:  RTSP/1.0 200 OK
      CSeq: 1
      Public: DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE
```

2.3.2 DESCRIBE

DESCRIBE veranlasst den Server die Präsentationsbeschreibung des durch die URL beschriebene Objekt zu senden. Dabei kann mittels des *Accept*-Parameters angegeben werden, welche Arten von Präsentationsbeschreibungen der Client verarbeiten kann. Wird keine angegeben oder ein vom Server nicht unterstütztes Format, ist es dem Server möglich, die Präsentationsbeschreibung frei zu wählen.
Die DESCRIBE-Methode kann nur vom Client benutzt werden.

Beispiel:

```
C→S:  DESCRIBE rtsp://server.example.com/fiz/foo RTSP/1.0
      CSeq: 312
      Accept: application/sdp, application/rtsl

S→C:  RTSP/1.0 200 OK
      CSeq:312
      Content-Type: application/sdp
      Content-Length: 44

      v=0
      o=- 2890844526 2890842807 IN IP4 192.16.24.202
      s=RTSP Session
      m=audio 0 RTP/AVP 0
      a=control:rtsp://audio.example.com/fiz/audio.en
      m=video 0 RTP/AVP 31
      a=control:rtsp://video.example.com/fiz/video
```

2.3.3 ANNOUNCE

Die ANNOUNCE-Methode hat unterschiedliche Funktionen. Falls sie als Nachricht vom Client zum Server geschickt wird, kann die Präsentationsbeschreibung des angegebenen Objekts verändert werden. Wird sie hingegen vom Server zum Client gesendet, wird die Präsentationsbeschreibung aktualisiert.

Beispiel:

```
C→S:  ANNOUNCE rtsp://server.example.com/fiz/foo RTSP/1.0
      CSeq: 312
      Session 471123444
      Content-Type: application/sdp
      Content-Length: 332

      v=0
      o=mhandley 2890844526 2890845468 IN IP4 126.16.64.4
      s=SDP Seminar
      i=A Seminar on the session description protocol
      u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
      e=mjh@isi.edu (Mark Handley)
      c=IN IP4 224.2.17.12/127
      t=2873397496 2873404696
      a=recvonly
      m=audio 3456 RTP/AVP 0
      m=video 2232 RTP/AVP 31

S→C:  RTSP/1.0 200 OK
      CSeq:312
```

2.3.4 SETUP

Die SETUP-Methode kann nur vom Client verwendet werden. Mittels dieser Methode werden Transporteinzelheiten spezifiziert, die bzgl. der Quelle gelten sollen. (vgl. auch Parameter *Transport* im Abschnitt 2.2.2). Im Laufe ei-

ner Übertragung ist es so möglich, die Transportparameter mit einem weiteren Aufruf dieser Methode zu ändern. Diese Funktion muss jedoch nicht in jeder Implementierung umgesetzt sein.
Als Antwort wird unter anderem die vom Server generierte Session-ID übermittelt.

Beispiel:

```
C→S:  SETUP rtsp://example.com/foo/bar/baz.rm RTSP/1.0
      CSeq: 302
      Transport: RTP/AVP;unicast;client_port=4588-4589

S→C:  RTSP/1.0 200 OK
      CSeq: 302
      Session: 47112344
      Transport: RTP/AVP;unicast;client_port=4588-4589;
      server_port=6256-6257
```

2.3.5 PLAY

PLAY startet die Übertragung der Daten mit dem in der SETUP-Methode spezifizierten Transportparameter. PLAY wird solange ausgeführt, bis der Server eine PAUSE-Nachricht bekommt oder die Übertragung der Daten beendet ist. Normalerweise wird die Quelle in voller Länge wiedergegeben. Es ist jedoch möglich mit dem Parameter *Range* einen Zeitraum anzugeben, der abgespielt werden soll. Sendet der Client mehrere PLAY-Requests hintereinander, werden die Requests in einer PLAY-Queue vom Server zwischengespeichert und nacheinander abgearbeitet. Im Falle eines auf eines PAUSE-Requests folgenden PLAY-Requests wird die Übertragung an der unterbrochenen Stelle wieder aufgenommen.

Beispiel:

```
C→S:  PLAY rtsp://audio.example.com/twister.en RTSP/1.0
      CSeq: 833
      Session: 12345678
      Range: smtpe=0:10:20-;time=19970123T153600Z

S→C:  RTSP/1.0 200 OK
      CSeq: 833
      Session: 12345678
      Range: smtpe=0:10:22-;time=19970123T153600Z
      Date: 23 Jan 1997 15:35:06 GMT
```

2.3.6 PAUSE

Die Methode PAUSE unterbricht die laufende Übertragung. Mit dem *Range*-Parameter kann wiederum ein fester Unterbrechungszeitpunkt angegeben werden. Die Ressourcen werden nicht sofort freigegeben, so dass nach einer gewissen Zeit die Wiedergabe oder Aufnahme erneut gestartet werden kann, ohne dass eine erneute Initialisierung mit SETUP nötig wäre.

Beispiel:

```
C→S:  PAUSE rtsp://example.com/fizzle/foo RTSP/1.0
      CSeq: 834
      Session: 12345678

S→C:  RTSP/1.0 200 OK
      CSeq: 834
      Session: 12345678
```

2.3.7 TEARDOWN

TEARDOWN stoppt die durch die URI spezifizierte Übertragung und gibt die beanspruchten Ressourcen wieder frei. Die vereinbarte Session-ID verliert ihre Gültigkeit und kann in folgenden Requests nicht mehr zugeordnet werden. Nur mit einem erneuten SETUP können die Transportparameter wieder gesetzt und die Quelle erneut abgespielt werden.

Beispiel:

```
C→S:  TEARDOWN rtsp://example.com/fiz/foo RTSP/1.0
      CSeq: 892
      Session: 12345678

S→C:  RTSP/1.0 200 OK
      CSeq: 892
```

2.3.8 GET_PARAMETER

Die GET_PARAMETER-Methode ermöglicht es, Parameter einer Präsentation auszulesen. Die Art bzw. Inhalt der Response wird dabei der jeweiligen Implementierung überlassen.

Beispiel:

```
S→C:  GET_PARAMETER rtsp://example.com/fiz/foo RTSP/1.0
      CSeq: 431
      Content-Type: text/parameters
      Session: 12345678
      Content-Length: 15

      packets_received
      jitter

C→S:  RTSP/1.0 200 OK
      CSeq: 431
      Session: 12345678
      Content-Length: 46
      Content-Type: text/parameters

      packets_received: 10
      jitter: 0.3838
```

2.3.9 SET_PARAMETER

Mit SET_PARAMETER können einzelne Parameter der Sitzung geändert werden. Dabei ist es nur möglich, einen Parameter zurzeit zu ändern, damit die Gegenseite den Parameter oder Wert in einem Response ablehnen kann. Anzumerken ist hierbei, dass mit dieser Methode keinesfalls Parameter, wie *Transport*, geändert werden dürfen. Dieser ist der SETUP-Methode vorbehalten.

Beispiel:

```
C→S:   SET_PARAMETER rtsp://example.com/fiz/foo RTSP/1.0
       CSeq: 421
       Content-Type: text/parameters
       Session: 12345678
       Content-Length: 20

       barparam: barstuff

S→C:   RTSP/1.0 451 Invalid Parameter
       CSeq: 431
       Session: 12345678
       Content-Length: 10
       Content-Type: text/parameters

       barparam
```

2.3.10 REDIRECT

Die REDIRECT-Methode informiert den Client über die Notwendigkeit einen anderen, durch den Parameter *Location* angegebenen, Server zu kontaktieren. Im Anschluss auf REDIRECT ist der Client angewiesen sich mit TEARDOWN bei dem aktuellen Server abzumelden und einen SETUP-Request an den neuen Server zu senden. Mit dem Parameter *Range* kann der Zeitpunkt angegeben werden, ab welchen Zeitpunkt die Umleitung durchgeführt werden soll. REDIRECT ist die einzige Methode die nur vom Server an den Client gerichtet sein darf.

Beispiel:

```
S→C:   REDIRECT rtsp://example.com/fiz/foo RTSP/1.0
       CSeq: 732
       Location: rtsp://bigserver.com:8001
       Range: clock=19960213T143205Z-
```

2.3.11 RECORD

RECORD bietet Optionen ähnlich der PLAY-Methode. Im Gegensatz zur PLAY-Methode wird der Server jedoch nicht angewiesen die Quelle abzuspielen, sondern aufzunehmen. Ansonsten sind dieselben Parameter wie in einem PLAY-Request erlaubt.

Beispiel:

```
C→S:  RECORD rtsp://example.com/meeting/audio.en RTSP/1.0
      CSeq: 954
      Session: 12345678
      Conference: 128.16.64.19/32492374
```

2.4 Zustandsmodelle

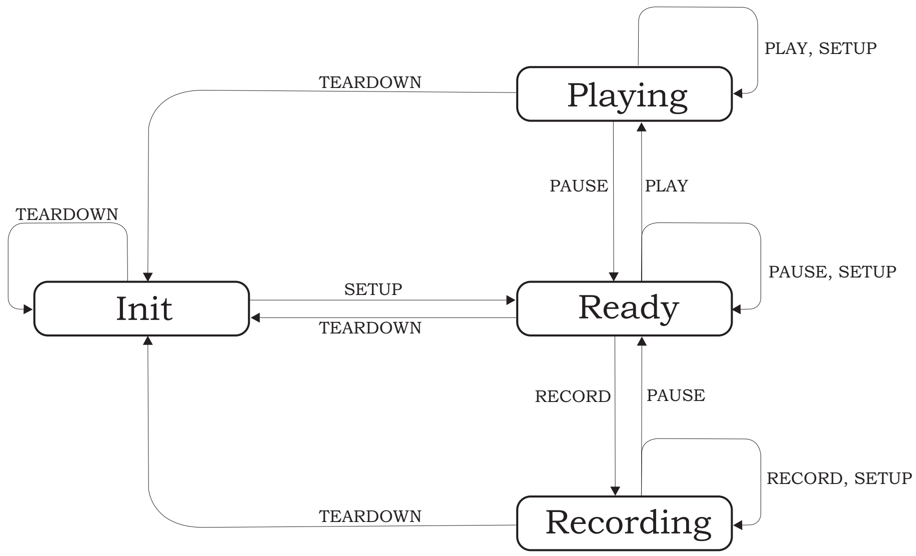

Abbildung 2.4: Das Zustandsmodell von Client und Server.

Das RTSP-Zustandsmodell beschreibt das Verhalten des Protokolls von der Initialisierung bis zur Terminierung einer RTSP-Sitzung. Durch die Einführung von Protokollzuständen können Client und Server relativ unabhängig voneinander arbeiten und sind nicht auf eine dauerhafte Steuerungsverbindung angewiesen. Wenn in einer Sitzung mehrere Medienquellen genutzt werden, wird für jedes Medium ein Objekt mit den RTSP-Zuständen angelegt. Dies geschieht bei Client und Server. Anhand der URL und der Session-ID können die RTSP-Nachrichten den Objekten zugeordnet werden. Eingehende Nachrichten bewirken dann evtl. Zustandsänderungen in diesem Objekt. Anhand des aktuellen Zustands entscheiden dann Client bzw. Server wie auf die eingegangene Nachrichten reagieren werden soll. Die Methoden OPTIONS, ANNOUNCE, DESCRIBE GET_PARAMETER und SET_PARAMETER bewirken keine Zustandsänderungen und sind in den Zustandstabellen daher auch nicht aufgelistet.

2.4.1 Zustandsmodell des Clients

Das Zustandsmodell des Clients beinhaltet vier Zustände:

- **Init:**
  Ein SETUP-Request wurde gesendet, warten auf Antwort des Servers.

- **Ready:**
  SETUP-Response (Zustand Init) oder PAUSE-Reponse im Zustand Playing empfangen.

- **Playing:**
  PLAY-Response empfangen.

- **Recording:**
  RECORD-Response empfangen.

Eine Zustandsänderung tritt beim Client nur ein, falls der Client einen positiven Response, also eine Nachricht mit einem Zustandscode *2xx*, erhält. In der folgenden Tabelle 2.2 ist der Folgezustand in der Spalte *neuer Zustand* aufgelistet. Falls der Client nach einem Request eine Nachricht bekommt, die als Statuscode einen Code *3xx* ausweist, wird der aktuelle Zustand auf INIT zurückgesetzt. Bei einem Code *4xx* tritt keine Zustandsänderung ein. Wenn kein SETUP-Request benötigt wird (bei einer Multicast-Übertragung) ist READY der Initialzustand.

Zustand	Request-Methode	neuer Zustand
Init	SETUP	Ready
	TEARDOWN	Init
Ready	PLAY	Playing
	RECORD	Recording
	TEARDOWN	Init
	SETUP	Ready
Playing	PAUSE	Ready
	TEARDOWN	Init
	PLAY	Playing
	SETUP	Playing
Recording	PAUSE	Ready
	TEARDOWN	Init
	RECORD	Recording
	SETUP	Recording

Tabelle 2.2: Die Zustände des Clients.

2.4.2 Zustandsmodell des Servers

Das Zustandsmodell des Servers enthält, ähnlich dem Zustandsmodell des Clients, ebenfalls vier Zustände:

- **Init:**
  Der Initialzustand, es wurde kein gültiger SETUP-Request empfangen.

- **Ready:**
  Ein SETUP-Request oder PAUSE-Request wurde empfangen und eine positive Response gesendet.

- **Playing:**
  Der letzte PLAY-Request war erfolgreich. Die Nutzdaten werden gesendet.

- **Recording:**
  Der Server nimmt Audio- oder Videodaten auf.

Eine Zustandsänderung beim Server tritt nur ein, falls der Server einen Request empfangen und ihn positiv beantwortet kann. Da der Server keine Empfangsbestätigungen vom Client erhält, ist ein Timeout vorgegeben nach dem der Server die Sitzung beendet bzw. in den Zustand INIT zurückkehrt. Der Timeout ist normalerweise auf 60 Sekunden festgelegt, kann aber im Response-Header durch dem Parameter *Timeout* neu definiert werden. Das Zustandsübergangsdiagramm entspricht dem Modell des Clients [4] (vgl. Abb. 2.4).

Zustand	Request-Methode	neuer Zustand
Init	SETUP	Ready
	TEARDOWN	Init
Ready	PLAY	Playing
	RECORD	Recording
	TEARDOWN	Init
	SETUP	Ready
Playing	PAUSE	Ready
	TEARDOWN	Init
	PLAY	Playing
	SETUP	Playing
Recording	PAUSE	Ready
	TEARDOWN	Init
	RECORD	Recording
	SETUP	Recording

Tabelle 2.3: Die Zustände des Servers.

3 Die Simulationsumgebung OMNeT++

OMNeT++ [9] ist eine objekt-orientierte, diskrete Ereignis-Simulationsumgebung. Die Abkürzung OMNeT++ steht dabei für *Objective Modular Network Testbed in C++* und hat seine Wurzeln in der Simulationsumgebung OMNeT von Dr. György Pongor. Die Umgebung ist geeignet eine Vielzahl von Systemen zu modellieren, u.a. Kommunikationsprotokolle, Computernetzwerke, Multiprozessor- und verteile Systeme.

Ein OMNeT++ Modell besteht dabei aus hierarchisch voneinander abgeleiteten Modulen, die mittels Nachrichten miteinander kommunizieren. Die Module in der untersten Hierarchiestufe enthalten die Algorithmen des Modells. Diese einfachen Module (simple modules) werden in der Programmiersprache C++ implementiert. Zusammengesetzte Module (compound modules) enthalten hingegen keinen eigenen Quelltext. Sie bestehen aus einfachen Modulen oder können wiederum zusammengesetzte Module enthalten. Die Module können unbegrenzt geschachtelt werden und dienen der Strukturierung des Modells.

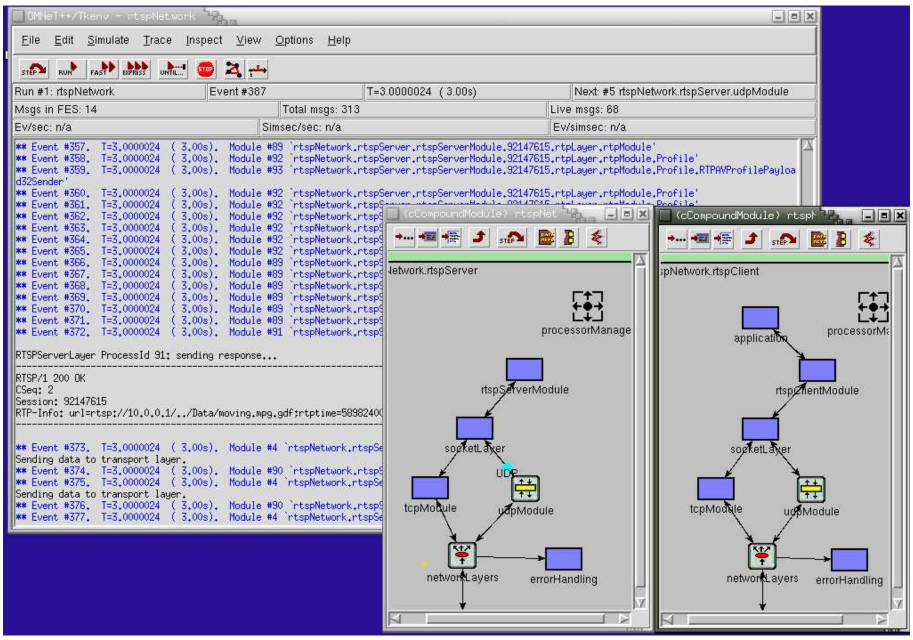

Abbildung 3.1: Die tkenv-Umgebung.

Für die Simulation mit OMNeT++ stehen zwei Laufzeitumgebungen zur Verfügung: die *cmdenv*-Umgebung und die *tkenv*-Umgebung. Die *cmdenv*-Laufzeitumgebung ist textbasiert und ist mit der Kommandozeile zu bedienen. *Cmdenv* bietet eine schnelle Ausführung der Simulation, ist zudem plattformunabhängig und wurde für die Stapelverarbeitung konzipiert. Eine graphische Ansicht des Simulationsablaufs bietet die *tkenv*-Umgebung. Die fensterbasierte Oberfläche erlaubt es, die Abläufe der Simulation graphisch zu verfolgen, sinnvoll für eine interaktive Ausführung, Debugging oder für Simulationspräsentationen.

3.1 Module

Wie bereits erwähnt unterscheidet OMNeT++ einfache und zusammengesetzte Module. Die einfachen Module enthalten dabei die Anwendungslogik und können nicht in weitere Module unterteilt werden. Um ein einfaches Modul zu erstellen, wird eine neue Klasse von *cSimple* abgeleitet. Die Funktionalität dieser neuen Klasse konzentriert sich dabei auf die Methode `activity()`. Diese Methode wird bei der Ausführung des OMNeT++-Modells aufgerufen. Nachrichten (vgl. Abschnitt 3.2) können nun (beispielsweise) mittels der Methoden `send()` und `receive()` zwischen verschiedenen Modulen ausgetauscht werden.

Eine andere Möglichkeit besteht in der Nutzung der `handleMessage()`-Methode. Diese wird jedes Mal durchlaufen, wenn ein Modul eine Nachricht erhält. Die Nachricht wird dabei der Methode als Parameter übergeben. Der Hauptunterschied dieser beiden Methoden, `activity()` und `handleMessage()`, besteht darin, dass es bei der `handleMessage()`-Methode nur unter Umständen möglich ist Zustandsinformationen zu speichern. Dafür ist es jedoch einfacher, eine grosse Anzahl unterschiedlicher Nachrichten effektiv zu bearbeiten. Zu beachten ist, dass jeweils nur eine der beiden Methoden genutzt werden darf.
Während eines Simulationslaufs wird der Code aller Module scheinbar gleichzeitig ausgeführt. Dies wird durch die Ausführung der Module als leichtgewichtige Prozesse (*coroutines*) erreicht.
Die zusammengesetzten Module können einfache oder weitere zusammengesetzte Module beinhalten. Sie dienen vor allem dazu, das Modell weiter zu strukturieren.

Die Topologie eines Modells wird durch die *NED*[1]-Sprache beschrieben. Sie erlaubt die Erstellung von Gattern und Verbindungen (vgl. Abschnitt 3.2) und die Parametrisierung der Module. Die Parametrisierung erlaubt es, den Modulen untereinander zu kommunizieren, durch Parameter die einfachen Module anzupassen oder auch die Zusammensetzung des Modells erst zur Laufzeit festzulegen. Die Parameter können dabei von unterschiedlichen Datentypen sein. Mittels des Editors *GNED* können diese Strukturen auch graphisch erstellt werden. Die NED-Beschreibung wird hierbei durch den Editor automatisch generiert.

[1]NED - NEtwork Description

3.2 Gatter, Verbindungen und Nachrichten

Gatter (*gates*) sind die Sende- und Empfangseinheiten der Module. Ein Ausgangsgatter kann durch eine Verbindung (*connection*) mit einem Eingangsgatter verbunden werden. Die Daten werden dabei zwischen den Modulen unidirektional übertragen. Für eine bidirektionale Übertragung sind also bei beiden kommunizierenden Modulen mindestens ein Ausgangsgatter und ein Empfangsgatter nötig. Eine Verbindung kann nur von Gattern derselben Hierachieebene, z.B. denen eines zusammengesetzten Moduls, hergestellt werden. Über Parameter ist es möglich einer Verbindung eine bestimmte Übertragungszeit, eine Datenrate und eine Bitfehlerrate zuzuweisen.

Entlang der erstellten Verbindungen können Nachrichten (*messages*) übertragen werden. Eine Nachricht ist dabei eine Instanz der Klasse *cMessage* aus der OMNeT++ Klassenbibliothek. Durch die Verwendung von Parametern können eigene Daten einer Nachricht hinzugefügt werden. Ebenfalls ist es möglich Nachrichten in andere Nachrichten einzukapseln. Diese Vorgehensweise ist bei der Simulation von Kommunikationsprotokollen nützlich. Außerdem ist es möglich eigene Klassen von cMessage abzuleiten und sich so den Umweg über Parameter zu ersparen, was sich günstig auf die Laufzeit des Simulation auswirkt.

In OMNeT++ 2.2 ist die Möglichkeit des *message subclassing* hinzugekommen. Das message subclassing ermöglicht die automatische Generierung von neuen Nachrichtenklassen. Parameter der neuen Klasse werden dabei aus einer Datei eingelesen und eine neue C++-Nachrichtenklasse erzeugt. Diese Methode gewährleistet die Erstellung von effizientem Code und eine einfache Erweiterung bestehender Nachrichtenklassen.

3.3 Die Klassenbibliothek

OMNeT++ umfasst auch eine umfangreiche Klassenbibliothek für die Implementierung der einfachen Module mittels C++. Für die Simulation von Kommunikationsprotokollen sind besonders die Klassen für den Nachrichtenaustausch, die Simulation von Warteschlangen und Zustandsautomaten hilfreich. Nähere Informationen können dem Benutzerhandbuch [9] entnommen werden.

4 Die Struktur der Module

Das RTSP-Simulationsmodell mit OMNeT++ baut auf eine Reihe bereits bestehender Module auf. Die Module *NetworkLayers*, *TcpModule* und *UdpModule* simulieren die Internet-Protokolle der unteren Schichten bis einschließlich der Transportschicht. Für die Simulation des Real-time Streaming Protocols werden jedoch nur die Module *SocketLayer* und *RtpLayer* der Anwendungsschicht direkt genutzt. Das *SocketLayer*-Modul greift dabei auf die Module *UdpModule* und *TcpModule* zurück, welche wiederum das Modul *NetworkLayers* benötigen.

4.1 Die Module des Clients

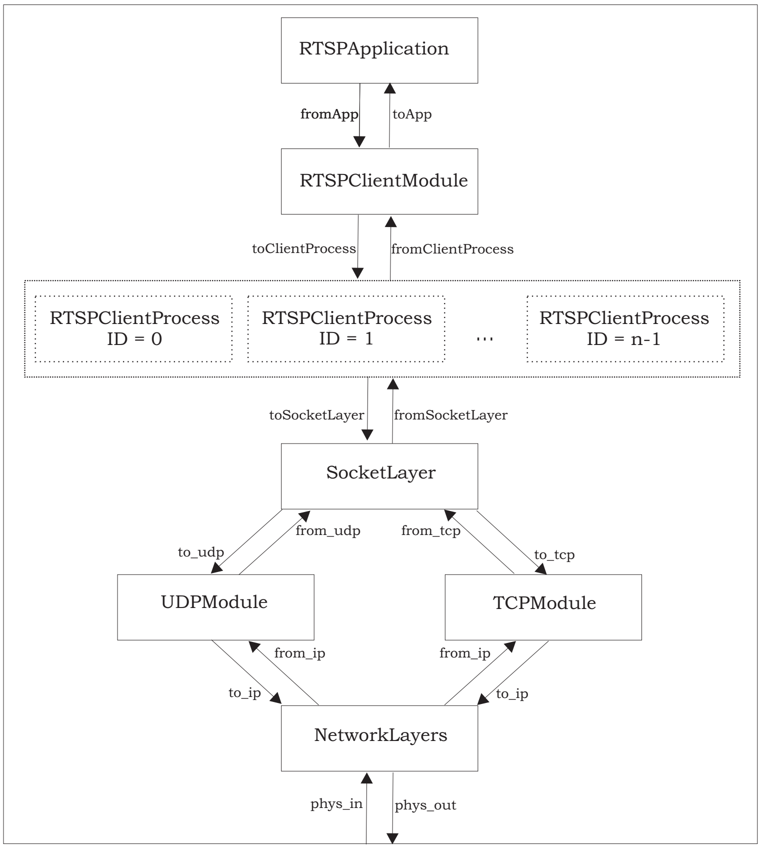

Abbildung 4.1: Die Modulstruktur des Clients.

4.1.1 Das Anwendungsmodul RTSPApplication

RTSPApplication
parameters: cmdFileName: string gates: toRTSP fromRTSP

Abbildung 4.2: Das RTSPApplication-Modul.

Das Modul RTSPApplication stellt die RTSP-Anwendung dar. Zugunsten der Wiederverwendung ist es getrennt von dem Modul der Protokollverarbeitung. In der Realität ist dieses Modul mit einem Userinterface zu vergleichen, dass es dem Benutzer ermöglicht den Medienstrom zu steuern, ähnlich einem Videorecorder mit Play-, Pause-, Recordtasten, Vorlauf, Zeitanzeige, usw. Dieses Simulationsmodell beschränkt sich jedoch darauf, die Benutzereingaben aus einer Datei einzulesen. Mittels dieser Datei können einzelne Kommandos, welche syntaktisch ähnlich der RTSP-Requests aufgebaut sind, einzelnen Zeitpunkten eines Simulationsablaufs zugeordnet werden. Das RTSPApplication-Modul liest und parst diese Datei und gibt dann an den vorgegebenen Zeitpunkten die Daten an das Modul RTSPClientModule weiter, welches die eigentliche Protokollverarbeitung beinhaltet.

Beschreibung der Parameter:

cmdFileName: Dieser String gibt den Namen der Datei an, in welcher die Benutzerkommandos lokalisiert sind.

4.1.2 Das Modul RTSPClientModule

Das RTSPClientModul ist das zentrale Modul des Clients. Es nimmt die Benutzerkommandos von der RTSPApplication entgegen, wertet diese aus und wandelt sie in gültige RTSP-Requests um. Falls das Kommando zu einer bereits bestehen Sitzung zugeordnet werden kann (Unterscheidung anhand der RTSP-URL), wird die Nachricht an die betreffende Sitzung weitergeleitet. Wenn die URL jedoch nicht zugeordnet werden kann, wird eine neue Sitzung erstellt. Dazu wird dynamisch ein Modul vom Typ RTSPClientProcess generiert. Um das Modul später korrekt zuordnen zu können, wird es nach der betreffenden URl benannt. Anschließend wird der Request an das neu erstellte Modul weitergeleitet und dort weiterverarbeitet.
Ein erstelltes Modul wird auch vom RTSPClientModul wieder gelöscht. Falls das Modul eine Nachricht *closeSession()* von einem RTSPClientSession-Modul empfängt wird dieses daraufhin entfernt. Diese Situation tritt ein, falls das Modul RTSPClientProcess einen Request TEARDOWN mit Bestätigung vom Server, empfangen hat.

RTSPClientModule
parameters: commonName: string profileName: string bandwidth: numeric portNumberRTP: numeric portNumberRTSP: string rtspTransportProtocol: string rtspUDPTimeout: numeric gates: toApp toClientProcess fromApp fromClientProcess

Abbildung 4.3: Das RTSPClientModul.

Beschreibung der Parameter:

commonName: Mit commonName wird der CNAME für den Teilnehmer festgelegt. Bei diesem und den folgenden drei Parametern handelt es sich um RTP-spezifische Parameter.

profileName: Das Profil, unter welchem die Übertragung stattfindet.

bandwidth: Die Bandbreite (in Byte pro Sekunde), welche für die Datenübertragung mit RTP zur Verfügung steht.

portNumberRTP: Mit diesem Parameter kann der RTP- und RTCP-Port festgelegt werden, falls nicht der Standardport verwendet werden soll. Für RTP wird normalerweise eine gerade Portnummer verwendet, für RTCP die nächst höhere.

portNumberRTSP: Der zu verwendende RTSP-Port.

rtspTransportProtocol: Dieser String gibt das Transportprotokoll an, welches RTSP zur Übertragung der Nachrichten verwenden soll. Im Moment sind die Protokolle TCP und UDP möglich.

rtspUDPTimeout: Spezifiziert den Timeout für Übertragungswiederholungen bei Verwendung des Protokolls UDP. Die Angabe erfolgt hierbei in Sekunden.

4.1.3 Das Modul RTSPClientProcess

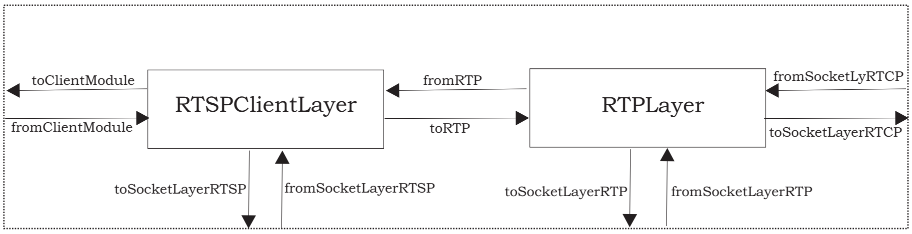

Abbildung 4.4: Das Modul RTSPClientProcess.

Bei dem Modul RTSPClientProcess handelt es sich um ein zusammengesetztes Modul, bestehend aus den einfachen Modulen RTSPClientLayer und RTPLayer. Es wird dynamisch von dem Modul RTSPClientModul erzeugt und auch wieder gelöscht. Das Modul RTPLayer wurde bereits in einer vorangegangenen Studienarbeit [2] entwickelt, das Modul RTSPClientLayer wird im nächsten Abschnitt genauer beschrieben.

RTSPClientProcess
parameters: commonName: string profileName: string bandwidth: numeric portNumberRTP: numeric portNumberRTSP: string rtspTransportProtocol: string rtspUDPTimeout: numeric gates: toApp toSocketLayerRTP toSocketLayerRTCP toSocketLayerRTSP fromApp fromSocketLayerRTP fromSocketLayerRTCP fromSocketLayerRTSP

Abbildung 4.5: Das RTSPClientProcess-Modul.

Beschreibung der Parameter:

commonName: Alle Parameter entsprechen den Parameter vom RTSPClient-Modul.

profileName: vgl. RTSPClientModul.

bandwidth: vgl. RTSPClientModul.

portNumberRTP: vgl. RTSPClientModul.

portNumberRTSP: vgl. RTSPClientModul.

rtspTransportProtocol: vgl. RTSPClientModul.

rtspUDPTimeout: vgl. RTSPClientModul.

4.1.4 Das Modul RTSPClientLayer

```
RTSPClientLayer
parameters:
    commonName: string
    profileName: string
    bandwidth: numeric
    portNumberRTP: numeric
    portNumberRTSP: string
    rtspTransportProtocol: string
    rtspUDPTimeout: numeric

gates:
    toApp
    toRTP
    toSocketLayer
    fromApp
    fromRTP
    fromSocketLayer
```

Abbildung 4.6: Die RTSPClientLayer.

Das Modul RTSPClientLayer spiegelt eine RTSP-Sitzung wieder. Es empfängt die weitergeleiteten Requests vom RTSPClientModul und wertet diese weiter aus. Anschließend wird über das Modul *SocketLayer*, der simulierten Socketschnittstelle, eine Verbindung zum RTSP-Server aufgebaut. Außerdem wird die RTPLayer initialisiert und auf die evtl. zu empfangenen Nutzdaten vorbereitet. Alle wichtigen Funktionen, wie Auswertung der Responses und Übertragungswiederholungen (bei der Verwendung von UDP), sind in diesem Modul implementiert. Die zentrale Rolle wird hierbei von einem Zustandsautomaten aus der OMNeT++ Klassenbibliothek übernommen, der genau das RTSP Zustandsmodell aus Abschnitt 2.4 widerspiegelt. Anhand des Automaten können leicht gültige von ungültigen Methoden und Parametern unterschieden werden.

Beschreibung der Parameter:

commonName: Alle Parameter entsprechen den Parameter von dem RTSP-ClientModul.

profileName: vgl. RTSPClientModul.

bandwidth: vgl. RTSPClientModul.

portNumberRTP: vgl. RTSPClientModul.

portNumberRTSP: vgl. RTSPClientModul.

rtspTransportProtocol: vgl. RTSPClientModul.

rtspUDPTimeout: vgl. RTSPClientModul.

4.2 Die Module des Servers

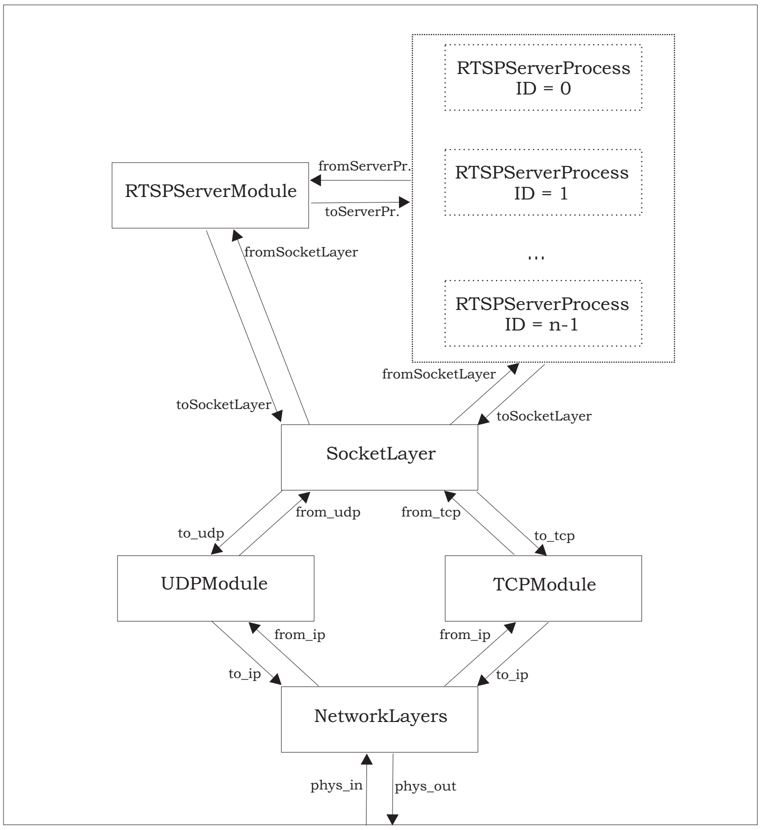

Abbildung 4.7: Die Modulstruktur des Servers.

4.2.1 Das Modul RTSPServerModule

Das RTSPServerModul ist das serverseitige Pendant zum RTSPClientModul auf dem Client. Das Server-Modul wartet auf Request von einem Client. Wenn der Server einen SETUP-Request erhält, wird ein neues RTSPServerProcess-Modul zur Laufzeit generiert. Als Bezeichnung wird die im Vorfeld generierte Session-ID gewählt. Anschließend wird der Request an das neue Modul weitergeleitet und dort weiterverarbeitet. Requests welche bereits eine gültige Session-ID enthalten, werden an das bereits existierende Modul weitergeleitet.

Beschreibung der Parameter:

portNumberRTP: Mit diesem Parameter kann der RTP- und RTCP-Port festgelegt werden, falls nicht der Standardport verwendet werden soll. Für RTP wird normalerweise eine gerade Portnummer verwendet, für RTCP die nächst höhere.

portNumberRTSP: Der zu verwendende RTSP-Port.

RTSPServerModule
parameters: portNumberRTP: numeric portNumberRTSP: string rtspTransportProtocol: string gates: toSocketLayer toServerProcess fromRTP fromSocketLayer fromServerProcess

Abbildung 4.8: Das RTSPServerModul.

rtspTransportProtocol: Dieser String gibt das Transportprotokoll an, welches RTSP zur Übertragung der Nachrichten verwenden soll. Im Moment sind die Protokolle TCP und UDP möglich.

4.2.2 Das Modul RTSPServerProcess

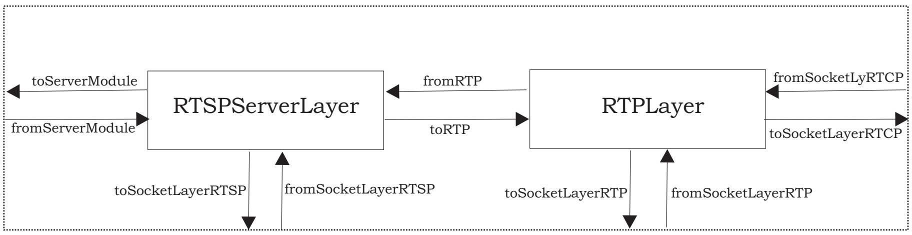

Abbildung 4.9: Das Modul RTSPServerProcess.

Der RTSPServerProcess ist ähnlich dem RTSPClientProcess aufgebaut. Das Modul ist ebenfalls ein zusammengesetztes Modul aus den einfachen Modulen RTSPServerLayer und wiederum dem Modul RTPLayer. Es erlaubt dem RTSPServerModul dynamisch eine neue RTSP-Sitzung mit einer RTP-Layer zu erstellen. Auf die RTSPServerLayer wird im nächsten Abschnitt eingegangen.

Beschreibung der Parameter:

portNumberRTP: vgl. RTSPServerModule.

rtspTransportProtocol: vgl. RTSPServerModule.

RTSPServerProcess
parameters: portNumberRTP: numeric rtspTransportProtocol: string gates: toApp toSocketLayerRTP toSocketLayerRTCP toSocketLayerRTSP fromApp fromSocketLayerRTP fromSocketLayerRTCP fromSocketLayerRTSP

Abbildung 4.10: Das RTSPServerProcess-Modul.

4.2.3 Das Modul RTSPServerLayer

Die RTSPServerLayer ist wiederum ähnlich der RTSPClientLayer aufgebaut. Anstatt die Responses zu bearbeiten, werden allerdings die Request behandelt. Der Zustandsautomat ist ähnlich dem des Clients aufgebaut. Das Modul ist ebenfalls für die Kommunikation mit dem RTPLayer zuständig. Nach entsprechenden Requests wird die RTPLayer initialisiert, mit der Übertragung der Nutzdaten begonnen, die Übertragung unterbrochen oder beenden.

RTSPServerLayer
parameters: portNumberRTP: numeric rtspTransportProtocol: string remote_addr: string remote_port: numeric gates: toRTP toServerModul toSocketLayer fromRTP fromServerModul fromSocketLayer

Abbildung 4.11: Das RTSPServerLayer-Modul.

Beschreibung der Parameter:

portNumberRTP: vgl. RTSPServerModule.

rtspTransportProtocol: vgl. RTSPServerModule.

remote_addr: Spezifiziert die Adresse des Clients, von dem der Request entgegengenommen wurde.

remote_port: Spezifiziert den Port des Clients, von dem der Request entgegengenommen wurde.

5 Implementation

In diesem Kapitel wird die Klassenstruktur des RTSP Simulationsmodells beschrieben. Im ersten Abschnitt werden die Klasse vorgestellt, die die Funktionalität der im letzten Kapitel genannten Module beinhalten. Anschließend werden im zweiten Abschnitt die Nachrichtenklassen rtspRequest und rtspRequest vorgestellt, die eine Request- bzw. eine Response-Nachricht darstellen.

5.1 Die Modul-Klassen

5.1.1 Die Klasse RTSPApplication

Die Klasse RTSPApplication beinhaltet die Implementierung des RTSPApplication-Moduls. Sie dient dazu Kommandos aus einer Datei einzulesen und diese an das RTSPClientModule weiterzuleiten. Die Klasse implementiert die `activity()`-Methode. Diese ruft die Methode `readCmdFile()` auf, und verweilt anschließend in einer Endlosschleife.
Die `readCmdFile()`-Methode liest die Kommandos aus einer, über den Parameter *cmdFile* angegebenen, Datei ein. Jede einzelne Zeile der Datei wird eingelesen und anschließend geparst. Eine Zeile ist dabei nach dem Schema

Zeit SP RTSP-Methode SP Quelladresse SP Quelldatei SP opt. Parameter

aufgebaut. Dabei gibt der Parameter *Zeit* an, nach welchem Zeitpunkt (bezogen auf die Simulationszeit) das Kommando gesendet werden soll. Die weiteren Parameter sind weitgehend selbsterklärend; eine Beispieldatei liegt der Implementierung bei. Im Folgenden werden die Kommandos in Nachrichten der Klasse rtspRequest umgewandelt und an das RTSPClientModul geschickt.

5.1.2 Die Klasse RTSPClientModule

Zur Nachrichtenverarbeitung benutzt diese Klasse die `handleMessage()`-Methode. Diese unterscheidet die Nachrichten nach ihrer Herkunft und leitet sie an die entsprechenden Methoden weiter. Die Nachrichten können dabei von der RTSPApplication oder einem RTSPClientProcess-Modul (eine RTSP-Sitzung) stammen.
Bei einer Nachricht von der Anwendung wird in der Methode `handleMessage-fromApp()` geprüft, ob die Nachricht einer bereits existierenden Session zugeordnet werden kann. Falls ja, wird die Nachricht an den betreffenden RTSPClientProcess weitergeleitet. Im anderen Fall wird eine neue Session mittels der Methode `createSession()` erzeugt. Alle erforderlichen Links werden gesetzt

und ein Modul vom Typ RTSPClientProcess erstellt und initialisiert. Die Parameter des RTSPClientModuls werden an das neu erstellte Modul weitergegeben. Anschließend wird der von der Anwendung empfangene Request vervollständig und um einige variable RTSP-Parameter ergänzt. Dieser Request wird dann an das erstellte Modul (an das dort eingebettete RTSPClientLayer-Modul) weitergeleitet.

Bei einer Nachricht von einem RTSPServerProcess kann es sich um eine Statusnachricht handeln, die Statusinformationen über den letzten Request/Response enthält, oder um eine `closeSession()`-Nachricht. Die Statusinformationen werden, ohne geprüft zu werden, an die Anwendung weitergeleitet. Wenn das Modul jedoch eine `closeSession()`-Nachricht erhält, wird das RTSPClientProcess-Modul von dem die Nachricht einging, gelöscht. Die Sitzung ist damit beendet.

In dieser und in den folgenden drei Klassen RTSPClientProcess, RTSPServer-Modul und RTSPServerProcess sind jeweils noch die Methoden `writeRequest-Contents()` und `writeResponseContents()` implementiert, die dazu dienen, die Requests und Responses in der OMNeT++-Umgebung auszugeben.

5.1.3 Die Klasse RTSPClientLayer

Nach dem Eingang einer Nachricht vom RTSPClientModul wird ein neuer Socket über das SocketLayer-Modul geöffnet. Die Adresse und der Port werden dazu der empfangenen Nachricht entnommen. Anschließend wird jede weitere eingehende Nachricht wiederum nach ihrer Herkunft unterschieden und in den Methoden `handleMessageFromSocketLayer()`, `handleMessageFromRTP()` oder `handleMessageFromApp()` weiterverarbeitet. Außerdem ist auch der Empfang einer *SelfMessage* möglich, die nach dem Ablauf eines Timers (Timeout) generiert wird.

Empfangene Nachrichten von der Anwendung, d.h. über das RTSPClientModul weitergeleitete Nachrichten, werden um sitzungsspezifische Parameter ergänzt und über den geöffneten Socket an den Server gesendet. Diese Funktionalität wird in der Methode `handleMessageFromApp()` realisiert. Wenn als Transportprotokoll UDP gewählt wurde, wird ein Timeout gesetzt und eine Kopie jeder zu sendenden Nachricht angelegt. Beim Ablauf des Timers wird eine *Selfmessage* generiert, welche in der Methode `handleSelfMessage()` bearbeitet wird. In dieser Methode wird die Kopie der Nachricht erneut übertragen und anschließend der Timer wieder zurückgesetzt. Ebenfalls wird je nach Methode des Requests die RTPLayer gesteuert. In der `handleMessageFromRTP()`-Methode werden die Nachrichten der dieser Sitzung, über das zusammengesetzte Modul RTSPClientProcess, zugeordnete RTPLayer bearbeitet. Hierbei handelt es sich um Statusmeldungen die nach dem initialisieren, pausieren, stoppen oder starten der RTPLayer gesendet werden.

Bei den über die SocketLayer eingehenden Nachrichten handelt es sich (in den überwiegenden Fällen) um Responses des Servers. Die Klasse RTSPClientProcess beinhaltet zur Bearbeitung dieser Responses eine Finite State Machine *cFSM* aus der OMNeT++ Klassenbibliothek. Anhand des gerade aktiven Zustands *INIT*, *READY* oder *PLAYING*, werden die eintreffenden Nachrichten in der Methode `handleResponse()` bearbeitet. Diese leitet die Nachrichten dann nach dem gerade aktiven Zustands an die Methoden `handleResponseStateInit()`, `handleResponseStateReady()` oder `handleResponseStatePlaying()` weiter. In diesen Methoden wird die Response nach dem enthaltenden Statuscode verifiziert. Folgend wird eine Statusmeldung an die Anwendung gesendet. Falls der gesendete Request ein TEARDOWN-Request war, wird nach positiver Quittung eine `closeSession()`-Nachricht an das RTSPClientModul gesendet und von dort die Sitzung (das Modul) gelöscht.

5.1.4 Die Klasse RTSPServerModule

Je nach dem über die Parameter gewählten Transportprotokoll wird die Methode `createServerSocketUDP()` oder `createServerSocketTCP()` aufgerufen und ein neuer Socket erstellt und dem gewählten Port zugewiesen. Anschließend wird in der `handleMessage()`-Methode auf ankommende Nachrichten gewartet. Unterschieden werden Nachrichten von der SocketLayer und vom RTSPServerProcess. Bei einer Nachricht vom RTSPServerProcess kann es sich nur um eine evtl. gesendete `closeSession()`-Nachricht handeln (vgl. Abschnitt 5.1.2).

Nachrichten vom SocketLayer werden in der Methode `handleMessageFromSocketLayer()` behandelt. In dieser Methode wird anfangs geprüft ob es sich um einen RTSP-Request handelt. Dieser wird dann an die Methode `handleRequest()` übergeben. Nun wird kontrolliert, ob der Request eine Session-ID enthält und anhand dieser einer bestehenden Sitzung zugeordnet werden kann. Falls dies nicht so ist, wird über die Methode `createSession()` eine neue Sitzung erzeugt. Andernfalls wird die Session-ID aus dem Request ausgelesen und an das korrekte weitergeleitet. Die `createSession()`-Methode benutzt die Methode `generateSessionId()` um eine neue Session-ID zu erzeugen. Diese besteht aus einer zufällig generierten Zahlenfolge. Anschließend wird das neue RTSPServerProcess-Modul erzeugt und um es später eindeutig identifizieren zu können nach der Session-ID benannt. Alles benötigten Links werden gesetzt, die Parameter an das neue Modul übergeben und die Nachricht weitergeleitet.

5.1.5 Die Klasse RTSPServerLayer

Die Klasse RTSPServerProcess bildet das Zustandsmodell des Servers ab. Sie bedient sich dazu ebenfalls der Klasse *cFSM* der Klassenbibliothek zur Simulation des Zustandsautomaten. Die Methodenstruktur ist mit der der RTSPClientLayer vergleichbar. Der Unterschied besteht darin, dass keine Übertragungswiederholungen durchgeführt werden und anstatt Responses Requests behandelt werden. Anhand des aktuellen Zustands werden in der `handleRequest()`-Methode die Methoden `handleRequestStateInit()`, `handleRequestStateReady()` oder `handleRequestStatePlaying()` aufgerufen.
Diese überprüfen wiederum den Request und senden die erforderlichen Nachrichten an die RTPLayer. Anschließend wird auf eingehende Nachrichten von

der RTPLayer gewartet und in der Methode `handleMessageFromRTP()` behandelt. Falls die Ausführung der Kommandos positiv von der RTPLayer quittiert wurde, wird ein Response generiert, mit den erforderlichen Parametern, Statuscode und Beschreibung, und zurück an den Client gesendet. Ansonsten ist die Struktur der Methoden äquivalent der der RTSPClientLayer.

5.2 Die Nachrichten-Klassen

5.2.1 Die Klassen rtspRequest und rtspResponse

Die Nachrichtenklassen rtspRequest und rtspResponse sind mittels des bereits in Abschnitt 3.2 auf Seite 19 beschriebenen message subclassing implementiert worden. Zur Änderung bzw. Erweiterung der Klassen muss die NED-Datei RTSPMessage.ned bearbeitet werden. Mittels einer Skriptsprache können die Parameter und Variablen der Nachrichten geändert werden. Nach der Übersetzung in eine C++-Klasse kann auf diese Parameter mit *set*Variable und *get*Variable zugegriffen werden.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Studienarbeit wurde eine Simulation des Real-Time Streaming Protocols mit OMNeT++ entwickelt und implementiert. Bei der Implementierung wurde besonderen Wert auf die Umsetzung des im RFC spezifizierten Zustandsautomaten gelegt, um eine einfache Erweiterung der Simulation ähnlichem dem Protokoll zu gewährleisten. Neue bzw. noch nicht verfügbare RTSP-Methoden können relativ einfach eingefügt werden. Dank des in der neuen OMNeT++-Version verfügbaren message subclassing ist es auch möglich geworden die C++-Nachrichtenklassen einfach nachträglich anzupassen und so die RTSP-Nachrichten um neue Parameter zu ergänzen.

Außerdem wurde die Implementierung auf das bereits erstellte RTP-Simulationsmodell abgestimmt, um eine möglichst problemlose Interaktion der beiden Protokolle zu ermöglichen. Um die Kommunikation dieser beiden Protokolle noch weiter zu verbessern, wäre eine weitere Anpassung beider Simulationsmodelle nötig, um sie noch besser zu verzahnen. In einigen Fällen ist es beispielsweise nötig, dass das Protokoll RTSP Parameter von RTP übernimmt um sie weiter zu verarbeiten (vgl. mit Parameter RTP-Info, Abschnitt 2.2.2 auf Seite 6).

Als weitere Ergänzungen des Modells käme die Implementierung weiterer Parameter und Methoden in Betracht. Im Moment enthält das Modell nur solche Methoden und Parameter die zur Umsetzung eines minimalen Servers und Clients nötig sind. Weiterhin verwendet das Modell im Moment nur UDP als Transportprotokoll, ist jedoch für die Verwendung von TCP vorbereitet. Dazu ist die Erweiterung der SocketLayer-Simulation um die Verwendung von TCP nötig.
Unabhängig von den vorgeschlagenen Verbesserungen ist es mit der bestehenden Simulation möglich, den Ablauf von RTSP-Sitzungen zu simulieren und die Ergebnisse zur evtl. Verbesserungen bzw. Ergänzungen des Protokolls zu verwenden.

Literaturverzeichnis

[1] H. Schulzrinne. Internet Media-on-Demand: The Real-time Streaming Protocol. Overview slides, Dept. of Computer Science, Columbia University, December 2001.

[2] M. Oppitz. Entwurf und Implementierung einer Simulation des Real-time Streaming Protocol unter OMNeT++, June 2002.

[3] M. Zhu. Implementation and evaluation of a socket interface layer under OMNeT++, 2002.

[4] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326, Columbia University, Netscape, RealNetworks, April 1998.

[5] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, Audio-Video Transport Working Group, January 1996.

[6] M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327, ISI/LBNL, April 1998.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, UC Irvine and DEC and MIT/LCS, January 1997.

[8] F. Yergeau. UTF-8, a transformation format of ISO 10646. RFC 2279, Alis Technologies, January 1998.

[9] A. Varga. OMNeT++ - Discrete Event Simulation System. User manual, Technical University of Budapest, May 2001.