Simulation des Session Initiation Protocol (SIP) mit OMNeT++

15. September 2003

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Sommersemester 2002

Jan Burghard
Hans-Geitel-Straße 18
38126 Braunschweig
Wirtschaftsinformatik, 10. Semester
j.burghard@tu-bs.de

Susann Geisler
Bienroder Weg 54
38108 Braunschweig
Wirtschaftsinformatik, 10. Semester
s.geisler@tu-bs.de

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Studienarbeit

Simulation des Session Initiation Protocol (SIP) mit OMNeT++

von
cand. Wi-Inf. Susann Geisler

cand. Wi-Inf. Jan Burghard

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

Braunschweig, den 15. September 2003

Erklärung an Eides statt

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

(Ort, Datum) (Unterschrift - Susann Geisler)

(Ort, Datum) (Unterschrift - Jan Burghard)

Kurzfassung

In dieser Arbeit wird das **Session Initiation Protocol** (SIP) mit Hilfe der Simulationsumgebung **OMNeT++** simuliert. **SIP** ist ein Protokoll der Sitzungsschicht, das dazu dient Multimediasitzungen aufzubauen. **OMNeT++** ist eine ereignisorientierte Simulationsumgebung, durch die Netzwerke simuliert werden können.
Diese Arbeit umfasst den Entwurf und die Implementierung des *SIP*. Zunächst werden die Bestandteile des *SIP* modelliert und danach implementiert.

Aufgabenstellung Seite 1

Aufgabenstellung Seite 2

Aufgabenstellung Seite 3

Aufgabenstellung Seite 4

Inhaltsverzeichnis

1. Einleitung

Das **Session Initiation Protocol** (SIP) ist ein Protokoll zur Einrichtung und Verwaltung von Multimedia-Sessions. Es hat sich z.B. in den letzten Jahren als eine Alternative zum H.323-Protokoll für die IP-Telefonie entwickelt. Um das Verhalten eines solchen Protokolls auch für größere Netze effizient untersuchen zu können, bietet sich die Möglichkeit der Simulation. Das bedeutet, dass in einer Simulationsumgebung ein Modell des zu testenden Netzes erzeugt wird. Dort kann das Modell dann unterschiedlichen Einflüssen und Probeläufen ausgesetzt werden, ohne dass das Netz wirklich schon existieren müsste. Die Simulation stellt also eine ressourcensparende Möglichkeit dar, ein zukünftiges Netzwerk zu testen.

In dieser Studienarbeit soll das SIP mit Hilfe der Simulationsumgebung **OMNeT++** simuliert werden. **OMNeT++** ist eine Simulationsumgebung, welche die Generierung komplexer Modelle aus grundlegenden Einheiten gut unterstützt.[1] Beispielsweise können so Server- und Client-Komponenten zu Proxys zusammengebaut werden. In **OMNeT++** lassen sich auch Zustandsautomaten direkt modellieren, was für Signalisierungsprotokolle besonders interessant ist.

Da zur Zeit noch keine Implementierung des SIP für **OMNeT++** existiert, besteht die Hauptaufgabe darin, das SIP für diese Simulationsumgebung zu implementieren. Dabei soll auf schon vorhandene Komponenten wie der Netzwerkschicht (*IP-Suite*) und insbesondere dem *Socket Layer* aus [Zhu01] aufgebaut werden. Als Transportprotokoll soll das *Realtime Transport Protocol* (RTP)[2] zum Einsatz kommen.

Der Aufbau und die Funktionen des SIP sind als RFC[3] veröffentlicht und können daraus entnommen werden. Um an dem modularen Prinzip der Simulationsumgebung fest zu halten, müssen das Protokoll bzw. seine Funktionen sinnvoll untergliedert werden. Wir haben uns für eine strukturelle Gliederung nach Client, Server, Proxy-Server, usw. entschieden, da diese späteren Elementen eines Netzwerkes am nächsten kommen.

Das Ergebnis jeder Simulation kann immer nur so gut wie das zugrunde gelegte Modell sein. Alle Einflüsse auf das reale Netzwerk müssen in das Modell mit aufgenommen und dort verarbeitet werden. Vor der Implementierung ist also zu untersuchen, welche Einflüsse auf den Untersuchungsgegenstand wirken und wie sie ihn beeinflussen. Da das SIP auf den schon vorhandenen UDP-Socket der IP-Suite aufsetzen soll, und wir davon ausgehen, dass dieser gewissenhaft überprüft wurde, beschäftigt uns die Umgebung des Protokolls in dieser Studienarbeit allerdings nur am Rande.

[1] vgl. [Vara]
[2] siehe [Opp02] und [S+96]
[3] RFC 3261 von [R+02]

2. Spezifikation des Session Initiation Protocol (sg)

Der Aufbau des Session Initiation Protocol (SIP) wurde ursprünglich in [H+99] dokumentiert. Mittlerweile wurde es aber durch die neuere Version [R+02] abgelöst. Diese Arbeit orientiert sich ausschließlich an letzterem Dokument. Das RFC beschreibt Endpunkte, Nachrichten und Konditionen, zu denen Endpunkte Nachrichten austauschen. Diese einzelnen Elemente werden im Folgenden beschrieben.

2.1. Dialog vs. Sitzung

SIP unterscheidet zwischen Dialogen und Sitzungen. Ein Dialog wird generiert, wenn ein SIP-Endsystem(*User Agent (UA)*) eine Anfrage an einen anderen *UA* sendet und eine endgültige Antwort der Statusklasse 2 erhält. In diesem Fall wird immer ein Dialog generiert. Eine Sitzung wird nur generiert, wenn ein Dialog aufgebaut wird, und beide Kommunikationspartner die Parameter der Sitzung akzeptieren. Ein *UA* sendet eine Sitzungsbeschreibung, die seinen Wünschen entspricht. Nur wenn der andere *UA* dieser Beschreibung zustimmt, wird eine Sitzung generiert.
Detaillierter wird darauf noch einmal im Kapitel 5 eingegangen. Im Folgenden werden die Begriffe *Sitzung* und *Dialog* gleichbedeutend verwendet. Erst im Kapitel 5 erfolgt eine begriffliche Trennung.

2.2. Der Aufbau von SIP-Nachrichten

Eine SIP-Nachricht ist entweder eine Anfrage (*Request*) von einem Client an einen Server oder eine Antwort (*Response*) von einem Server an einen Client.

2.2.1. Aufbau

SIP-Nachrichten bestehen aus einer Start-Zeile, gefolgt von einer oder typischerweise mehreren Header-Zeilen. Der Körper der SIP-Nachricht wird durch eine Leerzeile (CRLF) nach der letzten Header-Zeile angeschlossen.

Start-Zeile = Request-Line / Status-Line
Header-Zeilen = *message-header
CRLF
Nachrichtenkörper = message-body

Tabelle 1: Aufbau von SIP-Nachrichten

Wie in Tabelle 1 zu sehen ist, unterscheiden sich Anfragen und Antworten hauptsächlich in der Start-Zeile. Bei Anfragen ist die Start-Zeile die „Request-Line“ mit der Syntax:

Request-Line = *Method* SP *Request-URI* SP *SIP-Version* CRLF

wobei „Method“ eine der Anfrage-Methoden ist, „SP“ für ein Leerzeichen (Space) steht und „CRLF“ die Zeile beendet. Antworten beginnen mit der „Status-Line“ und haben ansonsten den gleichen Aufbau. Die Syntax der Statuszeile ist:

Status-Line = *SIP-Version* SP *Status-Code* SP *Reason-Phrase* CRLF

2.2.2. Header-Felder

Das SIP kennt eine Vielzahl von Header-Feldern. In Tabelle 2 sind diejenigen, die in dieser Arbeit verwendet werden, kurz vorgestellt. Der Name ist der im Protokoll normalerweise ausgeschriebene Feldbezeichner. Der Typ gibt an, ob das Header-Feld in Anfragen (Requests), Antworten (Response) oder beiden vorkommt. Die Art eines Feldes kann obligatorisch oder optional sein. Unter Methoden ist aufgelistet, bei welchen Anfrage- und Antwort-Methoden ein Header-Feld vorkommen kann bzw. muss.
Die Tabelle listet nur die Verwendung der Headerfelder auf, die implementiert ist. Weitere Verwendungsmöglichkeiten sind daher nicht ausgeschlossen.

Name	Typ	Art	Methoden
Call-ID	beide	obligatorisch	alle
Contact	beide	obligatorisch	INV
Content-Length	beide	obligatorisch	alle
Content-Type	beide	obligatorisch	alle
CSeq	beide	obligatorisch	alle
Max-Forwards	beide	obligatorisch	alle
From	beide	obligatorisch	alle
To	beide	obligatorisch	alle
Via	beide	obligatorisch	alle
Require	Anfrage	optional	INV
Allow	Anfrage	optional	INV BYE OPT REG
Supported	Anfrage	obligatorisch	INV
Supported	Anfrage	optional	CANC BYE OPT REG
Accept	Anfrage	obligatorisch	OPT
Accept	Anfrage	optional	INV BYE REG
Content-Disposition	Anfrage	optional	INV ACK BYE OPT REG
Content-Language	Anfrage	optional	INV ACK BYE OPT REG
Content-Encoding	Anfrage	optional	INV ACK BYE OPT REG
Expires	Anfrage	optional	INV

Tabelle 2: Header-Felder in SIP-Nachrichten

Call-ID gibt die ID der Einladung an, zu der die Nachricht gehört. Durch sie ist eine eindeutige Zuordnung einer Nachricht zu einer Sitzung möglich.

Contact gibt die Adresse des Senders an, unter der dieser bei späteren Nachrichten kontaktiert werden möchte. Das *To*-Feld späterer Nachrichten nimmt den Wert von *Contact* an.

Content-Length gibt die Länge des Nachrichteninhalts (message body) an.

Content-Type gibt den Typ des Nachrichteninhaltes an, falls dieser vorhanden ist. Dies kann z.B. *SDP* (*Session Description Protocol*) zum Übertragen von Sitzungsbeschreibungen sein.

CSeq gibt die Paketnummer der gesendeten Nachricht an. Dadurch ist das Sortieren eingehender Nachrichten und das Neu-Anfordern von verlorengegangenen Paketen möglich, wenn diese Aufgabe nicht von der Netzwerkschicht übernommen wird. Dies ist z.B. der Fall, wenn UDP als Netzwerkprotokoll verwendet wird.

Max-Forwards gibt die Anzahl der Hosts an, die passiert werden dürfen, bevor eine Nachricht beim Empfänger ankommt. Dadurch sollen Schleifen entdeckt werden. Der Standardwert dieses Headerfeldes ist 70.

From gibt den Sender der Nachricht an.

To gibt den Empfänger der Nachricht an. Dies können ein (Unicast) oder mehrere Hosts (Multicast) sein.

Via gibt die Hosts an, die beim Übertragen der Nachricht zum Empfänger kontaktiert wurden. Damit repräsentiert *Via* die Route des Paketes, die dieses bei der Übertragung durchläuft. Erhält ein Host eine Nachricht und leitet diese weiter (Proxy), so trägt er seine Adresse in das Feld ein. Dies garantiert, dass eine Antwortnachricht den gleichen "Weg" durchläuft wie die Anfrage. Beim Senden einer Antwort wird die eigene Adresse gelöscht und die Nachricht an die zuletzt eingefügte Adresse im *Via*-Feld gesendet. Beim Empfänger der Antwort besteht das *Via*-Feld aus einem Eintrag, der Adresse des Hosts selbst. Das ist für *stateful proxies* wichtig.
Ein *Via*-Feld enthält eine *branch-ID*, die eine lokale Transaktions-ID für den beteiligten Rechner darstellt. Ein *received*-Feld ist vorhanden, wenn die Nachricht von einer anderen IP-Adresse empfangen wurde, als angegeben wurde. Das Feld enthält dann die tatsächliche IP-Adresse.

Alle weiteren Headerfelder beziehen sich auf einen enthaltenen Nachrichteninhalt:

Require enthält Tags, die SIP-Erweiterungen repräsentieren. Der adressierte *UAS* muss diese Erweiterungen verstehen, um die Anfrage bearbeiten zu können. Die Tags werden auch *option tags* genannt.

Allow enthält alle Methoden, die der *UAC*, der die Anfrage stellt, versteht.

Supported enthält die Erweiterungen, die der *UAC* versteht.

Accept enthält die Content-Typen, die der *UAC* versteht.

Content-Disposition gibt an, wie der Nachrichteninhalt interpretiert werden soll. Ein möglicher Wert dieses Feldes ist "session".

Content-Language gibt an, welche Sprachen der *UAC* versteht.

Content-Encoding gibt an, welche Kodierungen der *UAC* versteht.

Expires gibt die Gültigkeitsdauer der Einladung an. Dieses Feld wird nur bei *INVITE*-Anfragen verwendet.

2.2.3. Die SIP-Methoden

SIP basiert auf einem HTTP-ähnlichen Anfrage- / Antwort-Transaktionsmodell (request / response transaction model). Zu jeder Transaktion gehört eine Anfrage (Request) und mindestens eine Antwort (Response). Durch die Anfrage wird die angegebene Methode auf dem adressierten Server aufgerufen. Durch den Aufruf von Methoden können Sitzungen aufgebaut und beendet werden. Außerdem können Parameter bereits bestehender Sitzungen geändert werden.
SIP stellt folgende sechs Methoden zur Verfügung:

- INVITE: dient dem Aufbau einer Sitzung
- ACK: dient dem Aufbau einer Sitzung
- CANCEL: beendet eine schwebende Sitzung
- BYE: beendet eine etablierte Sitzung
- OPTIONS: Beschaffung von Informationen über den Kommunikationspartner
- REGISTER: dient dem Registrieren eines *UA*s unter einer Adresse

Um einen Dialog aufzubauen, müssen zwei Anfragen gesendet werden. Die erste Anfrage enthält die Methode *INVITE*, die zweite *ACK*. Erst mit dem Senden bzw. Empfangen der *ACK*-Anfrage wird der Dialog etabliert. Durch eine *INVITE*-Anfrage wird ein Benutzer zu einer Sitzung eingeladen. Um Parameter

einer bereits bestehenden Sitzung zu ändern, muss ein *re-INVITE* gesendet werden.
In dieser Studienarbeit werden die Methoden INVITE, ACK, CANCEL, BYE und elementar REGISTER implementiert.

2.2.4. Die Status-Code Klassen

SIP verwendet folgende Status-Code Klassen:

- 1xx: Provisional - die Anfrage wurde erhalten und wird bearbeitet,
- 2xx: Success - die Anfrage wurde erfolgreich erhalten, verstanden und akzeptiert,
- 3xx: Redirection - um die Anfrage zu vervollständigen, sind weitere Informationen notwendig,
- 4xx: Client Error - die Anfrage ist syntaktisch falsch oder kann von dem Server nicht bearbeitet werden,
- 5xx: Server Error - eine offensichtlich richtige Anfrage konnte nicht bearbeitet werden,
- 6xx: Global Failure - die Anfrage kann von keinem Server bearbeitet werden.

Wichtige Meldungen für den Aufbau einer Sitzung sind:

- 100 Trying: signalisiert, dass versucht wird, das Paket weiterzuleiten,
- 180 Ringing: signalisiert, dass beim Empfänger „geklingelt“ wird,
- 183 Session Progress: signalisiert, dass die Anfrage bearbeitet wird und
- 200 OK: signalisiert, daß die Nachricht erfolgreich empfangen und bearbeitet wurde.

2.3. SIP-Interaktionsmodell

RFC 3261 definiert die Art und Reihenfolge der Nachrichten, die UAs austauschen müssen, um die Funktionalität von SIP zu verwenden. Dies umfasst eine Spezifikation für jede Methode. Im Folgenden werden

- der Aufbau einer Sitzung mit den Methoden *INVITE* und *ACK*,
- das Beenden einer schwebenden Sitzung mit der Methode *CANCEL* und
- das Beenden einer etablierten Sitzung mit der Methode *BYE*

beschrieben.
Zu beachten ist, daß die Rollen des UAS bzw. des UAC im Laufe der Sitzung getauscht werden können. Beim Aufbau der Sitzung ist allerdings ein Rollentausch nicht möglich, sondern erst nach dem Etablieren eines Dialoges.

2.3.1. Der Aufbau einer Sitzung

Genereller Ablauf

Für den Aufbau einer Sitzung wird der sogenannte Drei-Wege-Handshake verwendet. Zunächst sendet ein als Client agierender UA (*UAC*) eine Anfrage mit der Methode *INVITE*, auf die der adressierte als Server agierende UA (*UAS*) antwortet. Nach Erhalt der Antwort sendet der *UAC* erneut eine Anfrage. Diese Anfrage enthält die Methode *ACK*. Damit wird die Sitzung etabliert. Der Drei-Wege-Handshake wurde gewählt, da für den Aufbau einer Sitzung eine Benutzereingabe beim *UAS* notwendig ist. Der adressierte Benutzer muss der Sitzung zustimmen. Der Zeitraum vom Eingang der Anfrage bis zur Benutzereingabe ist normalerweise länger als die übliche Antwortzeit des *UAS*. Der *UAC* signalisiert durch das Senden der *ACK*-Anfrage, dass er weiterhin am Aufbau der Sitzung interessiert ist.
Für den erfolgreichen Aufbau einer Sitzung muss die Antwort des *UAS* einen Statuscode der Klasse 2 besitzen. Andernfalls ist bei der Bearbeitung der Anfrage ein Fehler aufgetreten, und die Sitzung kann nicht aufgebaut werden.

Zu sendende Nachrichten

Abbildung 1 zeigt den Nachrichtenfluss beim Aufbau einer Sitzung.

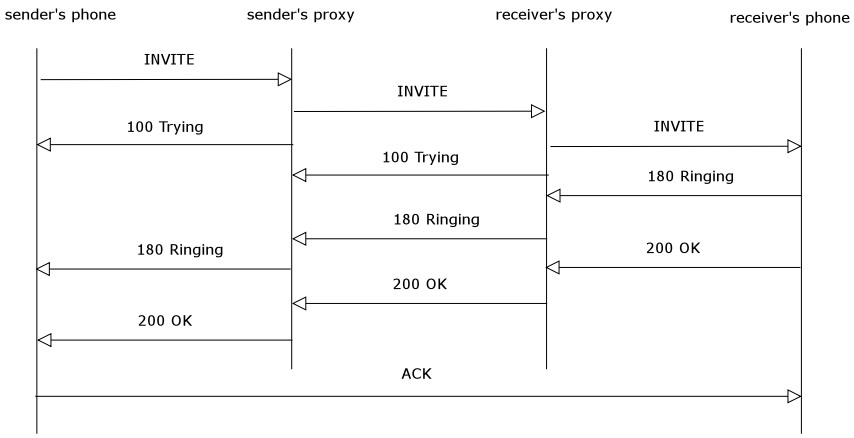

Abbildung 1: Aufbau der SIP-Sitzung

Zunächst sendet der *UAC* eine Anfrage mit der Methode *INVITE*. Bei dieser Anfrage müssen folgende Headerfelder gesetzt werden:

To gibt den Benutzer an, der zu einer Sitzung eingeladen werden soll (Adressat)

From gibt den Benutzer an, der einen anderen Benutzer zu einer Sitzung einladen möchte (Sender der INVITE-Anfrage)

Call-ID gibt die globale ID der Einladung an

CSeq gibt die Paketnummer der Anfrage an

Contact gibt die Adresse an, unter der der *UAC* später kontaktiert werden möchte

Content-Type gibt das Format des Nachrichteninhaltes an

Content-Length gibt die Länge des Nachrichteninhaltes an

Via gibt die Route des Paketes an

Max-Forwards gibt die Anzahl der Hosts an, die bei der Übertragung beteiligt sein dürfen

Method gibt die Methode an, hier *INVITE*

SIP Version gibt die verwendete SIP Version an

Request-URI gibt die Adresse des Empfängers an

Method, *SIP Version* und *Request-URI* sind Parameter der Request-Line (siehe Kapitel 2.2).

Jeder Proxy, der an der Übertragung der *INVITE*-Anfrage zum Empfänger beteiligt ist, sendet die Anfrage an einen anderen Proxy weiter, der die Nachricht weiter an den Benutzer leitet. Ist der Proxy für den Empfänger zuständig, leitet er die Anfrage an den Benutzer direkt weiter. In der Abbildung leitet der "sender's proxy" die Anfrage an den "receiver's proxy" weiter. Der "receiver's proxy" ist der zuständige Proxy für den Empfänger (receiver) und leitet die Anfrage an den adressierten Benutzer. Jeder Proxy, der die Anfrage erhält, sendet eine *100 Trying* Antwort an den direkten Vorgänger. In diesem Fall ist das für den "sender's proxy" der Benutzerhost selbst, der die Sitzung aufbauen möchte. Der "receiver's proxy" sendet die *100 Trying* Antwort an den "sender's proxy". Da dieser bereits eine solche Antwort an den Benutzer gesendet hat, registriert er diese Antwort, sendet sie aber nicht an den Benutzer weiter.
Erhält der Empfängerhost die Anfrage, so prüft er die Anfrage auf generelle Fehler. Enthält die Anfrage Fehler, so sendet er eine Antwort mit dem entsprechenden Fehler-Status-Code an den Sender. Ist die Anfrage korrekt, klingelt er beim

Benutzer. Dieser bestimmt, ob er der Sitzung beitreten möchte. Da dies länger dauern kann, sendet der Host zunächst eine provisorische Antwort. In diesem Fall ist dies eine *180 Ringing* Antwort. Der Host muss bis zum Senden der endgültigen Antwort in regelmäßigen Abständen provisorische Antworten senden, damit die beteiligten Proxies wissen, dass die Anfrage noch läuft und nicht den dazugehörigen Prozess löschen.[4] Diese provisorischen Antworten werden von jedem Proxy bis zum Sender durchgeleitet. Normalerweise werden in diesem Zeitraum *183 Session Progress* Antworten gesendet. Eine endgültige Antwort ist eine Antwort der Statusklasse 2, die signalisiert, dass der adressierte Benutzer der Sitzung zustimmt.
Um die Sitzung zu etablieren, muss der *UAC* nun eine erneute Anfrage stellen. Diese Anfrage enthält die Methode *ACK*. Alle Headerfelder dieser Nachricht stimmen bis auf eine Ausnahme mit den Headerfeldern der *INVITE*-Anfrage überein. Diese Ausnahme ist das Methodenfeld im *CSeq*-Headerfeld. Die Methode der Anfrage ist *ACK*. Die *ACK*-Anfrage wird direkt an den Empfänger gesendet und durchläuft keine SIP-Proxies.

2.3.2. Das Beenden einer schwebenden Sitzung

Genereller Ablauf

Eine Sitzung ist im schwebenden Zustand, wenn eine *INVITE*-Anfrage gesendet, aber noch keine endgültige Antwort empfangen wurde. Ein *UAC* kann in diesem Zeitraum an einem Aufbau der Sitzung nicht mehr interessiert sein. Um den Aufbau der Sitzung zu stoppen, sendet er eine *CANCEL*-Anfrage an den zuvor adressierten *UAS*. Der *UAS* stoppt bei Erhalt dieser Anfrage das "Ringing" beim Benutzer.
Zu beachten ist, dass eine *CANCEL*-Anfrage erst nach Erhalt einer provisorischen Antwort gesendet werden darf. Dies garantiert, dass die *INVITE*-Anfrage vor der *CANCEL*-Anfrage vom *UAS* empfangen wird. Eine *CANCEL* Anfrage sollte nicht mehr nach dem Erhalt einer endgültigen Antwort gesendet werden, da sie keine Änderungen bei der nun bereits etablierten Sitzung verursacht.
Eine *CANCEL*-Anfrage wird direkt an den *UAS* gesendet und durchläuft keine SIP-Proxies.

Zu sendende Nachrichten

Abbildung 2 zeigt den Nachrichtenfluss beim Beenden einer schwebenden Sitzung.

Zunächst wird eine *INVITE*-Anfrage gesendet. Nach Erhalt mindestens einer provisorischen Antwort sendet der *UAC* die *CANCEL*-Anfrage. Alle Headerfelder dieser Anfrage stimmen bis auf eine Ausnahme mit den Headerfeldern der *INVITE*-Anfrage überein. Das Methodenfeld des *CSeq*-Headerfeldes nimmt den

[4] siehe hierzu auch Kapitel 6.3 auf Seite 45 unter „Timer C“

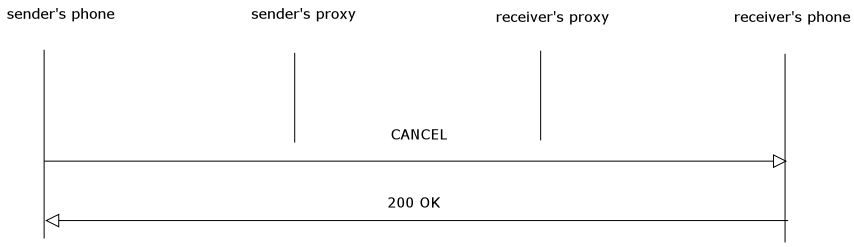

Abbildung 2: Beenden einer schwebenden SIP-Sitzung

Wert *CANCEL* an. Erhält der *UAS* die *CANCEL*-Anfrage, sendet er eine *487 Request Terminated* Antwort auf die *INVITE*-Anfrage. Auf die *CANCEL*-Anfrage antwortet er mit einer *200 OK* Antwort. Der *UAS* beendet nun die schwebende Sitzung und alle damit verbundenen Transaktionen und Sitzungen. Erhält der *UAC* beide Antworten, so löst er ebenfalls alle mit der Sitzung verbundenen Objekte auf.

2.3.3. Das Beenden einer etablierten Sitzung

Genereller Ablauf

Eine Sitzung ist im etablierten Zustand, wenn der Drei-Wege-Handshake erfolgreich durchgeführt wurde. Eine etablierte Sitzung wird mit einer *BYE*-Anfrage beendet. Besonderheit ist, dass diese Anfrage vom *UAS* gesendet wird. Dies besagt, daß der UA, der die *BYE*-Anfrage sendet, als *UAS* agiert.

Zu sendende Nachrichten

Abbildung 3 zeigt den Nachrichtenfluss beim Beenden einer etablierten Sitzung.

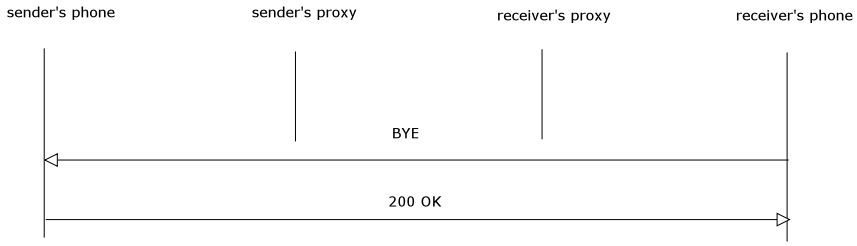

Abbildung 3: Beenden einer etablierten SIP-Sitzung

Zunächst sendet der *UAS* die *BYE*-Anfrage an den *UAC*. Dieser beendet bei Erhalt der Anfrage die Sitzung und sendet auf die *BYE*-Anfrage eine *200 OK* Antwort. Erhält der *UAS* die *200 OK* Antwort, beendet er ebenfalls die Sitzung.

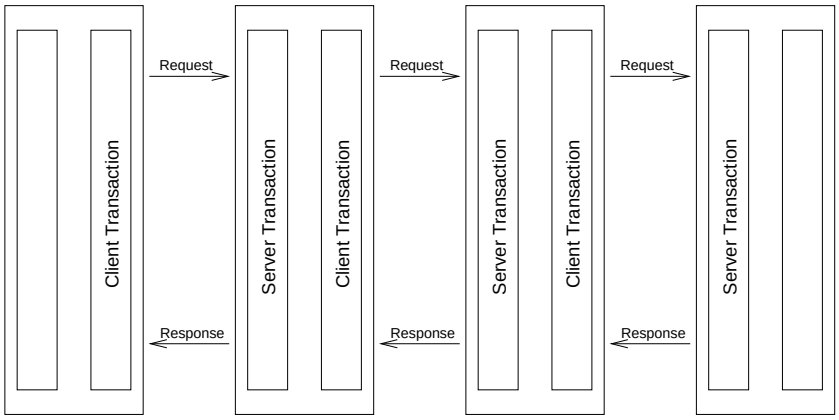

Abbildung 4: Transaktionsbeziehungen - vgl. [R+02] Abbildung 4

2.4. Transaktionsmodell

SIP ist ein transaktionsorientiertes Protokoll, da SIP-Elemente über den seriellen Austausch unabhängiger Nachrichten interagieren. Eine SIP-Transaktion besteht somit aus einer Anfrage (Request), null oder mehrerer provisorischer und einer abschließenden Antwort (Response). Somit hat eine Transaktion eine Client- und eine Server-Seite. Transaktionen sind logische Funktionen, die in allen Anwendungen, die das SIP implementieren, enthalten sind (Abbildung 4). Übergeordnete Module, die so genannten „Transaction User“ (*TU*) bedienen sich ihrer um den Nachrichtenaustausch vorzunehmen. Client-Transaktionen nehmen somit vom *TU* SIP-Anfragen entgegen und übertragen sie zuverlässig an Server-Transaktionen, die diese dann an „ihren“ *TU* übergeben. Umgekehrt ist es die Aufgabe der Server-Transaktion Antworten auf die Anfragen zu generieren bzw. vom *TU* erzeugte Antworten an die anfragende Client-Transaktion zuverlässig zurück zu senden.

Zusätzlich zur Unterteilung in Client- und Server-Transaktionen werden beide nochmal in *INVITE* und *Non-INVITE* Client- bzw. Server-Transaktion unterschieden. Das SIP kennt somit folgende Transaktionen:

1. die „INVITE Client-Transaktion“,
2. die „Non-INVITE Client-Transaktion“,
3. die „INVITE Server-Transaktion“ und
4. die „Non-INVITE Server-Transaktion“.

Die Unterscheidung von *INVITE*- und *Non-INVITE*-Transaktionen liegt daran, dass für die Anfragemethoden unterschiedliche Voraussetzungen vorliegen.

Um auf eine *INVITE*-Anfrage zu antworten wird typischerweise eine menschliche Interaktion vorausgesetzt. Das wiederum bedeutet im Vergleich zu automatisierten Interaktionen erheblich größere Antwortzeiten. Diesem Sachverhalt wird dadurch Rechnung getragen, dass zum Verbindungsaufbau statt eines Zwei-Wege-Handshake ein Drei-Wege-Handshake verwendet wird. Im Folgenden wird auf die einzelnen Transaktionsmodelle eingegangen.

2.4.1. INVITE Client-Transaktion

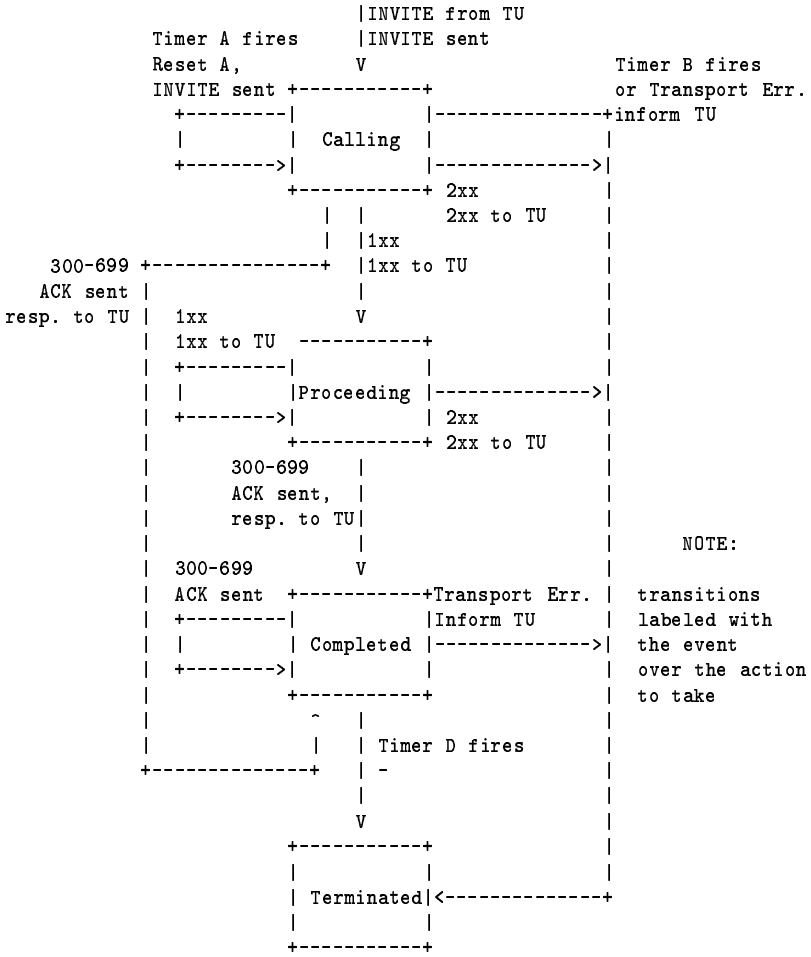

Abbildung 5: INVITE Client-Transaktion - vgl. [R+02] Abbildung 5

Die *INVITE*-Transaktion besteht aus einem Drei-Wege-Handshake. Die Client-Transaktion sendet eine Einladung (*INVITE*), die Server-Transaktion antwortet und die Client-Transaktion senden daraufhin eine Bestätigung (*ACK*). Dabei ist zu beachten, dass die Bestätigung nur bei Antworten mit dem Status größer oder gleich 300 zur Transaktion gehört. Die „ACK-Anfrage“ auf eine 2xx-Antwort

eines Servers muss direkt vom UAC generiert und verschickt werden. Das liegt daran, dass ein UAC anders auf eine 2xx-Antwort als ein Proxy reagieren muss. Während der Proxy ein 2xx verarbeitet und evtl. weiterleitet, muss der UAC eine Bestätigungsanfrage (*ACK*-Request) senden. Die genauen Verfahren der ACK-Behandlung werden später in den jeweiligen Kapiteln (5 und 6) aufgeführt.

Der Zustandsautomat der „INVITE Client-Transaktion“ ist in Abbildung 5 auf Seite 12 dargestellt und seine Zustände sind im Folgenden kurz erläutert:

Calling ist der initiale Zustand des Automaten. Von ihm aus muss der Automat gestartet werden, wenn der *TU* eine neue Client-Transaktion mit der *INVITE*-Anfrage startet. Die Transaktion muss die Anfrage zur Transportebene zur Versendung weitergeben. Wird Zeitgeber A (Timer A) ausgelöst, muss die Anfrage erneut gesendet und der Zeitgeber zurückgesetzt werden. Löst Zeitgeber B (Timer B) aus, wird der *TU* über die Zeitüberschreitung informiert und der Automat geht in den Zustand **Terminated** über.

Proceeding wird erreicht, wenn sich der Automat im Zustand **Calling** befindet und eine provisorische Antwort empfängt. Z.B. „180 Ringing“.

Complete Beim Erreichen dieses Zustandes muss Zeitgeber D gestartet werden. Über diesen Zeitgeber wird der Zustand auch wieder verlassen. Das kurze Verweilen in diesem Zustand dient nur dazu verzögert eintreffende oder erneut übertragene Antwortnachrichten zuordnen und evtl. beantworten zu können.

Terminated Sobald dieser Zustand erreicht wird, muss die Transaktion aufgelöst werden. Nachrichten, die danach für diese Transaktion eingehen, müssen vom *TU* behandelt werden.

2.4.2. Non-INVITE Client-Transaktion

Der große Unterschied zur *INVITE* Client-Transaktion besteht hier darin, dass es sich um einen Zwei-Wege-Handshake handelt. Das Senden der Bestätigung (*ACK*) fällt somit weg.

Das Zustandsmodell für diese Transaktion wird durch Tabelle 4 dargestellt.

2.4.3. Zuordnung von Antworten zu Client-Transaktionen

Eingehende Antworten müssen den richtigen Transaktionen zugeordnet werden, sodass sie weiter verarbeitet werden können. Das geschieht durch die Auswertung des *Branch*-Parameters des ersten *Via*-Feldes. Eine Antwort ist einer Transaktion zuzuordnen, wenn

Zustand	Eingabe	Aktion	neuer Zustand
Calling	1xx	1xx an TU	Proceeding
	2xx	2xx an TU	Terminated
	3xx-6xx	an TU - ACK senden	Complete
	Zeitg. A	INVITE erneut senden	
	Zeitg. B	Zeitüberschreitung melden	Terminated
Proceeding	1xx	1xx an TU	
	2xx	2xx an TU	Terminated
	3xx-6xx	an TU - ACK senden	Complete
Complete	3xx-6xx	ACK senden	
	Zeitg. D		Terminated
Terminated		Transaktion zerstören	

Tabelle 3: Zustandsmodell einer INVITE Client-Transaktion

Zustand	Eingabe	Aktion	neuer Zustand
Calling	1xx	1xx an TU	Proceeding
	2xx-6xx	an TU	Complete
	Zeitg. E	Anfrage erneut senden	
	Zeitg. F	Zeitüberschreitung melden	Terminated
Proceeding	1xx	1xx an TU	
	2xx-6xx	an TU	Complete
	Zeitg. E	Anfrage erneut senden	
	Zeitg. F	Zeitüberschreitung melden	Terminated
Complete	Zeitg. K		Terminated
Terminated		Transaktion zerstören	

Tabelle 4: Zustandsmodell einer Non-INVITE Client-Transaktion

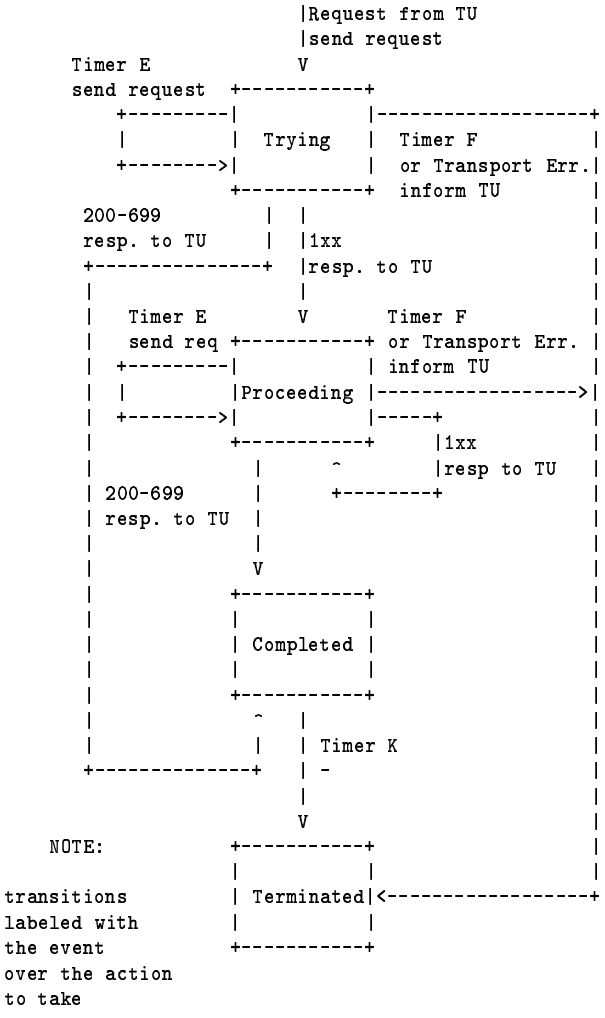

Abbildung 6: Non-INVITE Client-Transaktion - vgl. [R+02] Abbildung 6

1. der *Branch*-Parameter im ersten *Via*-Feld der Anfrage gleich dem *Branch*-Parameter im ersten *Via*-Feld der Antwort und

2. der Methoden-Parameter im *CSeq*-Feld der Anfrage gleich dem Methoden-Parameter im *CSeq*-Feld der Antwort ist.

2.4.4. INVITE Server-Transaktion

Die *INVITE* Server-Transaktion ist das Gegenstück zur *INVITE* Client-Transaktion. Sie nimmt Anfragen von der Transportschicht entgegen und reicht sie an den *TU* weiter. Diese generiert Antworten, die sie entgegen nimmt und zuverlässig zur Client-Transaktion sendet. Ihr Zustandsdiagramm zeigt Abbildung 7.

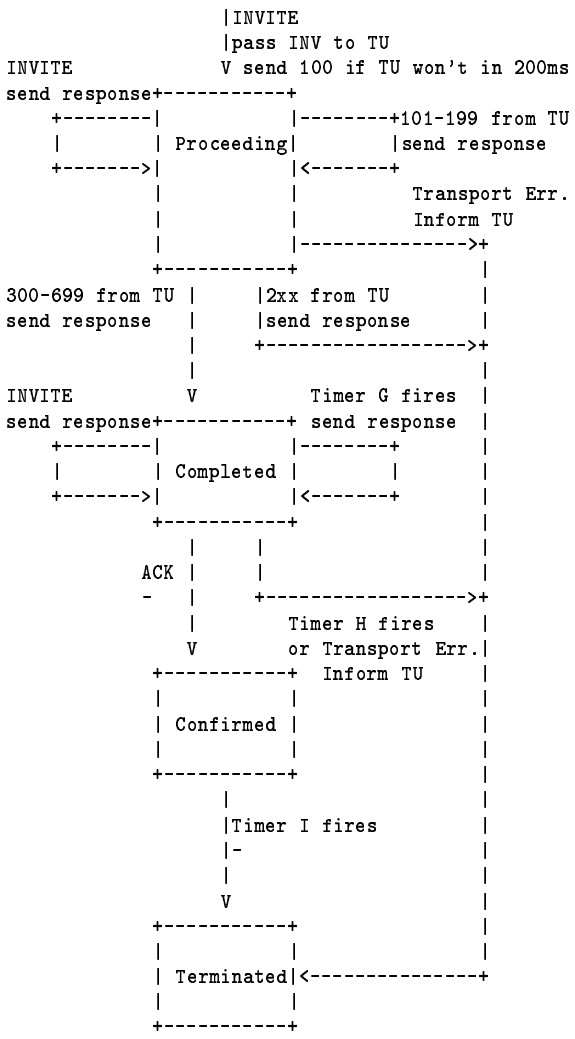

Abbildung 7: INVITE Server-Transaktion - vgl. [R+02] Abbildung 7

Zustand	Eingabe	Aktion	neuer Zustand
Beim Erzeugen	INVITE	INV zum TU und 100 Antwort falls TU nicht in 200ms antwortet	Proceeding
Proceeding	INVITE 101-199 vom TU 2xx vom TU 3xx-6xx vom TU	Antwort senden Antwort senden Antwort senden Antwort senden Zeitg. G starten	 Terminated Completed
Completed	INVITE ACK Zeitg. G Zeitg. H	Antwort senden Zeitg. I starten Antwort senden TU über Zeitüber- schr. informieren	 Confirmed Terminated
Confirmed	Zeitg. I		Terminated
Terminated		Transaktion zerstören	

Tabelle 5: Zustandsmodell einer INVITE Server-Transaktion

2.4.5. Non-INVITE Server-Transaktion

Die *Non-INVITE* Server-Transaktion unterscheidet sich kaum von der *INVITE* Server-Transaktion. Das genaue Verhalten lässt sich aus dem Zustandsdiagramm aus Abbildung 8 und aus Tabelle 6 erschließen.

2.4.6. Zuordnung von Anfragen zu Server-Transaktionen

Die Zuordnung einer Anfrage zu einer existierenden Server-Transaktion erfolgt über den *Branch*-Parameter des obersten *Via*-Header-Feldes. Um mit der Spezifikation von [R+02] RFC 3261 kompatibel zu sein, muss dieser Parameter mit einer bestimmten Zeichenkette - dem „Magic Cookie“ - starten. Ist das nicht der Fall, so muss die Zuordnung über ein Verfahren, das im RFC 2543 von [H+99] beschrieben ist, erfolgen. Darauf wird in dieser Arbeit nicht eingegangen.
Eine Anfrage gehört somit zu einer Transaktion, wenn

1. der *Branch*-Parameter mit dem „Magic Cookie“ beginnt,
2. der *Branch*-Parameter der Anfrage gleich dem der Anfrage ist, für die die Transaktion erzeugt wurde,
3. der *sent-by*-Wert im obersten *Via*-Feld der gleiche wie der der erzeugenden Anfrage ist und

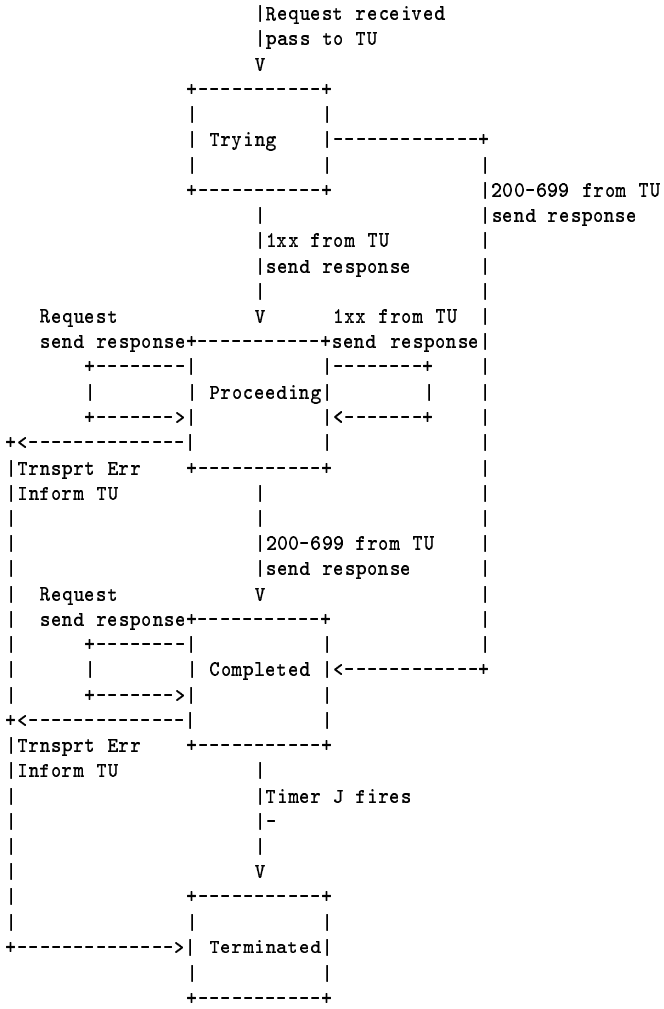

Abbildung 8: Non-INVITE Server-Transaktion - vgl. [R+02] Abbildung 8

Zustand	Eingabe	Aktion	neuer Zustand
Beim Erzeugen	Anfrage	Anfrage zum TU	Trying
Trying	1xx vom TU 2xx-6xx vom TU	Antwort senden Antwort senden	Proceeding Completed
Proceeding	Anfrage 1xx vom TU 2xx-6xx vom TU	Antwort senden Antwort senden Antwort senden Zeitg. J starten	Completed
Completed	Anfrage Zeitg. J	Antwort senden	Terminated
Terminated		Transaktion zerstören	

Tabelle 6: Zustandsmodell einer Non-INVITE Server-Transaktion

4. die Methode der Anfrage die gleiche ist, wie die derer, die die Transaktion erzeugte, mit Ausnahme der *ACK*-Methode, die durch eine *INVITE*-Methode erzeugt wurde.

3. Entwurf einer SIP-Anwendung (sg)

Da alle SIP-Anwendungen, seien es User Agents oder SIP-Server, viele Arbeitsschritte gemein haben, werden diese Arbeitsschritte unterteilt und durch separate Module abgebildet. Dadurch kann ein Großteil des Codes von User Agents auch in SIP-Servern zur Anwendung kommen.

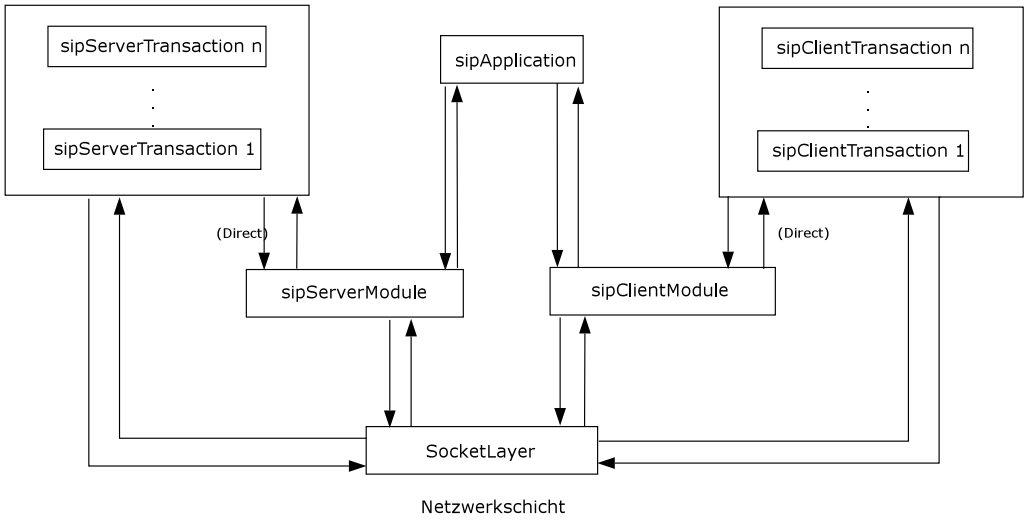

Abbildung 9: Aufbau eines SIP-Hosts

Abbildung 9 zeigt eine Unterteilung der einzelnen Module, bei der sich SIP-Anwendungen nur noch durch genau ein Modul unterscheiden - dem Modul *sipApplication*. Trotzdem befindet sich ein Großteil der Anwendungsfunktionalität in den gemeinsamen Modulen und muss bei Anpassungen der Anwendungsfunktionalität nicht mehr betrachtet werden.

3.1. Module

Hier werden die einzelnen Module beschrieben, aus denen ein SIP-Host oder ein SIP-Proxy „zusammengesetzt wird“.

3.1.1. sipClientTransaction

Die *sipClientTransaction* repräsentiert eine SIP-Transaktion. Für jede einzelne Client-Transaktion, die die Anwendungsebene anfordert, wird vom *sipClientModule* eine *sipClientTransaction* erzeugt, die einen *Socket* zum Server verbindet, die Transaktion bis zum *Terminated*-Zustand abarbeitet, dann den Socket wieder schließt und sich zum Schluss wieder auflöst.

Tor	Nachricht-Typ	Inhalt	Sender/Empfänger
fromApp	sipInterfacePacket	sipRequest	sipClientModule
fromSocketLayer	SocketInterfacePacket	Socket-management	SocketLayer
fromSocketLayer	SocketInterfacePacket	sipResponse	SocketLayer
selfMessage()	cMessage	Zeitgeber	
toSocketLayer	SocketInterfacePacket	Socket-management	SocketLayer
toSocketLayer	SocketInterfacePacket	sipRequest	SocketLayer

Tabelle 7: Tore (Gates) der sipClientTransaction

3.1.2. sipClientModule

Das *sipClientModule* ist für die Verwaltung der einzelnen Transaktionen und somit der *sipClientTransaction*-Module zuständig. Wird eine Anfrage von der übergeordneten Ebene empfangen, wird ein neues Untermodul vom Typ *sipClientTransaction* erzeugt und die Anfrage an dieses weiter geleitet. Ebenso kann die übergeordnete Ebene den Abbruch der Transaktion verlangen, was eine sofortige Zerstörung des betroffenen Moduls bedeutet.

Da in **OMNeT++** die Verbindungen von „normalen“ Modulen zu Untermodulen, wie es die *sipClientTransaction*en sind nicht dargestellt werden und die Datenflüsse somit auch nicht angezeigt werden können, kann dieses Modul auch

als eine Art Proxymodul für den Verkehr von *sipClientTransaction*en zum *SocketLayer* und umgekehrt dienen.

Tor	Nachricht-Typ	Inhalt	Sender/Empfänger
fromApp	sipInterfacePacket	sipRequest	höhere Ebene
fromApp	sipInterfacePacket	Transaktions-management	höhere Ebene
fromClientProc	sipInterfacePacket	Fehlermeldungen	sipClientTransact.
toApp	sipInterfacePacket	sipResponse	höhere Ebene
toApp	sipInterfacePacket	Fehlermeldungen	höhere Ebene

Tabelle 8: Tore (Gates) vom sipClientModule

3.1.3. sipServerModule

Das *sipServerModule* ist das Hauptmodul der Serverseite des *UA*, d.h. des *UAS* auf Transaktionsebene. Die Aufgabe dieses Moduls ist die Vermittlung von eingehenden Nachrichten. Es können Nachrichten von der *SocketLayer*, der Applikation und einer der Transaktionen empfangen werden. Nachrichten von der *SocketLayer* werden an die dazugehörige Transaktion bzw. die Applikation weitergeleitet. Wird von einer Transaktion eine Anfrage empfangen, wird diese an die Applikation weitergesendet, eine Antwort an die *SocketLayer*.
Existiert zu einer eingehenden Anfrage noch keine dazugehörige Transaktion, so erzeugt das *sipServerModule* eine entsprechende Transaktion. Das bedeutet, es wird ein Modul vom Typ *sipServerTransaction* mit der entsprechenden ID generiert.
Weiterhin ist das Modul dafür zuständig, nicht mehr benötigte Transaktionen zu löschen.

3.1.4. sipServerTransaction

Die *sipServerTransaction* repräsentiert eine Server-Transaktion. Sie nimmt Anfragen vom *sipServerModule* entgegen und bearbeitet diese. Zur Bearbeitung bezüglich eines Dialoges wird die Anfrage über das *sipServerModule* an die Applikation weitergeleitet. Antworten von der Applikation erhält das Modul vom *sipServerModul*, das als Vermittler agiert. Diese werden an die *SocketLayer* weitergeleitet.

3.1.5. Anwendung auf höherer Ebene - sipApplication

Die Erzeugung und Verarbeitung der Anfragen und Antworten geschieht im SIP auf höherer Ebene. Hier gibt es eine *sipApplication*, die z.B. die Rolle eines *User*

Tor	Nachricht-Typ	Inhalt	Sender/Empfänger
fromApp	sipInterfacePacket	Socket-management	höhere Ebene
fromApp	sipInterfacePacket	sipResponse	höhere Ebene
fromServerTransact.	sipInterfacePacket	sipRequest	sipServerTransact.
fromServerTransact.	sipInterfacePacket	Transaktions-management	sipServerTransact.
fromSocketLayer	SocketInterfacePacket	Socket-management	SocketLayer
fromSocketLayer	SocketInterfacePacket	sipRequest	SocketLayer
toApp	sipInterfacePacket	sipRequest	höhere Ebene
toServerTransact.	sipInterfacePacket	sipRequest	sipServerTransact.
toSocketLayer	SocketInterfacePacket	Socket-management	SocketLayer

Tabelle 9: Tore (Gates) vom sipServerModule

Tor	Nachricht-Typ	Inhalt	Sender/Empfänger
fromSocketLayer	SocketInterfacePacket	Socket-management	SocketLayer
fromSocketLayer	SocketInterfacePacket	sipRequest	SocketLayer
fromServerModule	sipInterfacePacket	sipRequest	sipServerModule
fromServerModule	sipInterfacePacket	sipResponse	sipServerModule
fromServerModule	sipInterfacePacket	Transaktions-management	sipServerModule
toServerModule	sipInterfacePacket	sipRequest	sipServerModule
toServerModule	sipInterfacePacket	Transaktion-management	sipServerModule
toSocketLayer	SocketInterfacePacket	Socket-management	SocketLayer
toSocketLayer	SocketInterfacePacket	sipResponse	SocketLayer

Tabelle 10: Tore (Gates) der sipServerTransaction

Agent Clients (UAC), *User Agent Servers* (UAS) oder auch eines *SIP Proxys* übernehmen kann und somit die anderen Module kontrolliert. Der Nachrichtenaustausch findet auch hier über Tore (Gates) statt, die für die *sipApplication* in Tabelle 11 beschrieben werden.

Tor	Nachricht-Typ	Inhalt	Sender/Empfänger
fromClientModule	sipInterfacePacket	sipResponse	sipClientModule
fromClientModule	sipInterfacePacket	Fehler-meldungen	sipClientModule
toClientModule	sipInterfacePacket	sipRequest	sipClientModule
toClientModule	sipInterfacePacket	Management	sipClientModule

Tabelle 11: Tore (Gates) der sipApplication

3.2. Schnittstellen

Zwischen den einzelnen Modulen des SIP-Modells ergeben sich Schnittstellen, über die Informationen (Nachrichten) ausgetauscht werden müssen. Im Folgenden wird beschrieben, welche Informationen ausgetauscht werden müssen und welche Anforderungen sich somit an die Schnittstellen ergeben. In Abbildung 9 auf Seite 19 sind die Zusammenhänge der Module als Pfeile dargestellt. Module, die mit Pfeilen verbunden sind, benötigen für jeden Pfeil eine Schnittstellendefinition, die das Format und den möglichen Inhalt der auszutauschenden Nachrichten definiert.

3.2.1. Das sipInterfacePacket

Angelehnt an die Socket-Layer-Architektur kommunizieren die SIP-Module untereinander mit Hilfe einer Interface-Klasse: dem *sipInterfacePacket*. Dieses Nachrichtenpaket bietet, wie das *SocketInterfacePacket*, Methoden zur Definition der Nachrichtenart und die *action()*-Methode um diese auszulesen an. Die implementierten Methoden sind:

Anfrage die Nachricht enthält eine Anfrage (Request)

Antwort die Nachricht enthält eine Antwort (Response)

Transaktion löschen wird von Transaktionen verwendet, um dem *sipServerModule* bzw. *sipClientModule* zu signalisieren, dass sie gelöscht werden sollen

Abbruch wird vom *sipClientModule* verwendet, um einer Transaktion ihren Abbruch zu befehlen

Port öffnen wird von der Applikation verwendet, um dem *UAS* auf Transaktionsebene zu signalisieren, einen Socket zu öffnen

Client-Fehlermeldung wird von einer Transaktion verwendet, um der Applikation zu signalisieren, dass beim Versenden einer Anfrage ein Timeout aufgetreten ist

Server-Fehlermeldung wird von einer Transaktion verwendet, um der Applikation zu signalisieren, dass beim Versenden einer Antwort ein Timeout aufgetreten ist

3.2.2. Schnittstellen zum Socketlayer

Die Schnittstellen zum Socket-Layer sind bereits von der SocketLayer-Implementation festgelegt. Der Socket-Layer akzeptiert nur Nachrichtenpakete der Klasse *SocketInterfacePacket*. Auf Nachrichten dieser Klasse können verschiedene Methoden ausgeführt werden, die dann die Art der Nachricht definieren, welche durch die *action()*-Methode auch ausgelesen werden kann. Um z.B. ein Datenpaket über den Socket-Layer zu verschicken wird auf dem Interface-Paket die Methode *write([socketFD], [Datenpaket])* ausgeführt, wobei [socketFD] der Zeiger auf den Socket, über den das Paket verschickt werden soll, ist.

Zur Kommunikation über den Socket-Layer stehen folgende Nachrichtenarten zur Verfügung:

socket fordert einen Socket von SocketLayer an.

connect baut eine Verbindung mit einem anderen Socket auf.

bind lässt den Socket auf eine Verbindungsanfrage eines anderen Sockets warten.

write verschickt ein Datenpaket an den entfernten Socket.

read liest ein empfangenes Datenpaket vom Socket.

3.2.3. sipApplication - sipClientTransaction

Alle Nachrichten, die nach den Abbildungen 5 und 6 zwischen der Applikation (TU) und der Transaktion ausgetauscht werden, müssen über das *sipClientModule* umgeleitet werden. Im Folgenden sind sie aufgeführt:

Anfrage von der Applikation an die Transaktion.

Antwort von der Transaktion an die Applikation.

Fehlermeldung von der Transaktion an die Applikation. Z.B. über Fehler bei der Nachrichtenübermittlung oder Zeitüberschreitungen.

3.2.4. sipApplication - sipClientModule

Das *sipClientModule* verwaltet die Transaktionen. Im Falle von *INVITE*-Nachrichten muß es eine Transaktion erzeugen und die Nachricht an sie weiter leiten. Für andere Anfrage-Methoden muss die Nachricht an die richtige Transaktion geleitet werden. Für diese Aufgaben muss das Modul die von der Applikation eingehenden Nachrichten untersuchen um entsprechende Maßnahmen zu treffen. Somit „versteht“ es alle Nachrichten der Applikation, die auch die *sipClientTransaction* von der Applikation versteht.

3.2.5. sipClientModule - sipClientTransaction

Zusätzlich zu den Nachrichtenarten, die zwischen Transaktion und Applikation existieren, benötigen diese beiden Module Nachrichtenarten, über die das *sipClientModule* seine Transaktionen verwalten kann. Das beinhaltet, dass das *sipClientModule* der *sipClientTransaction* mitteilen können muss, dass sie sich beenden soll und umgekehrt, dass sich die Transaktion beendet hat und gelöscht werden kann. Daraus folgen die Nachrichtenarten:

Abbruch um der Transaktion deren Abbruch zu befehlen und

Schließen um dem Client-Modul mitzuteilen, dass die Transaktion gelöscht werden kann.

3.2.6. sipApplication - sipServerModule

Die Applikation sendet Antworten über das *sipServerModule* an die Transaktion. Damit agiert das *sipServerModule* als Vermittler zwischen Applikation und Transaktionsen. Es werden ausschließlich *sipInterfacePacket*s ausgetauscht.

Port öffnen Das *sipServerModule* öffnet einen Port zur *SocketLayer*, von der später Nachrichten von einem anderen *UA* empfangen werden können

Antwort Das *sipServerModule* leitet erhaltene Antworten von der Applikation an die Transaktion weiter

3.2.7. sipApplication - sipServerTransaction

Die Applikation erhält von der Transaktion Anfragen und generiert für diese Antworten. Diese werden über das *sipServerModule* an die Transaktion gesendet. Dabei werden nur *sipInterfacePacket*s verwendet.

Antwort Die Transaktion erhält eine von der Applikation generierte Antwort

3.2.8. sipServerModule - sipServerTransaction

Das *sipServerModule* erhält von der *SocketLayer* Anfragen, die an die dazugehörige Transaktion weitergeleitet werden müssen. Existiert noch keine dazugehörige Transaktion, so wird eine neue generiert.

Anfrage Weiterleitung der von der *SocketLayer* erhaltenen Anfrage

3.2.9. sipServerTransaction - sipServerModule

Die *sipServerTransaction* sendet erhaltene Anfragen über das *sipServerModule* an die Applikation.

Anfrage zur Weiterleitung an die Applikation

Transaktion löschen terminiert eine Transaktion, so signalisiert sie dem *sipServerModule*, dass sie gelöscht werden kann

3.3. Interaktion der Module

In diesem Kapitel werden die beteiligten Transaktionen und ihre Parameter beschrieben. Die Netzwerkschicht wird vereinfachenderweise vernachlässigt.

3.3.1. Ablauf INVITE

In Abbildung 10 ist dargestellt, welche Transaktionen bei einer *INVITE*-Anfrage beteiligt sind.

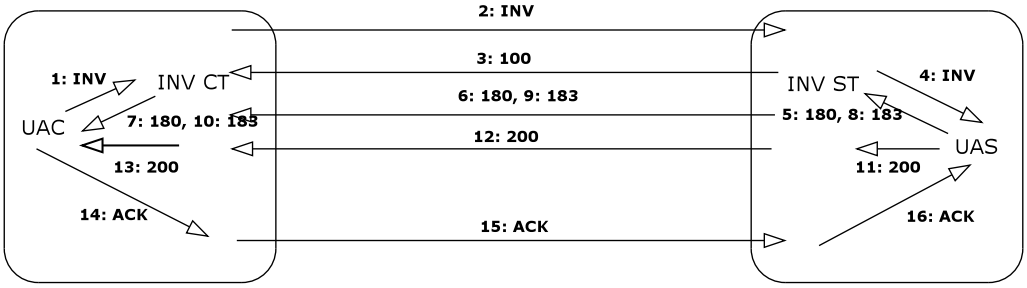

Abbildung 10: Ablauf einer INVITE Anfrage

Zunächst generiert die SIP-Applikation des *UAC* eine *INVITE Client-Transaction* (INV CT), die die *INVITE*-Anfrage an einen *UAS* sendet. Erhält der *UAS* diese Anfrage, so generiert er eine *INVITE Server-Transaction* (INV ST). Diese leitet die Anfrage an die SIP-Applikation weiter.
Erhält die INV ST eine endgültige Antwort der Statusklasse 2 von der SIP-Applikation, leitet sie diese an die Netzwerkschicht zur Übertragung an den *UAC*

weiter und beendet sich. Bei Erhalt dieser Antwort beendet sich die INV CT sofort. Die *ACK*-Anfrage wird direkt von der SIP-Applikation gesendet und betrifft keine Transaktionen. Nach dem Senden bzw. Erhalten der *ACK*-Anfrage ist die Sitzung auf beiden Seiten etabliert.
Erhält die INV ST eine endgültige Antwort der Statusklassen 3 - 6 von der SIP-Applikation, so sendet sie diese an den *UAC* weiter. Der *UAC* leitet die Antwort an die SIP-Applikation weiter und sendet eine *ACK*-Anfrage an den *UAS*. Beide Transaktionen beenden sich nach Ablauf eines Timers. Die Sitzung wird nicht etabliert.

3.3.2. Ablauf CANCEL

Abbildung 11 zeigt den Ablauf einer *CANCEL*-Anfrage.

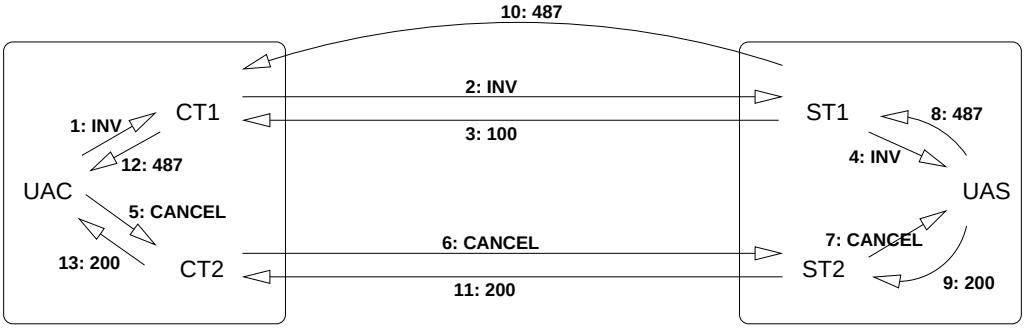

Abbildung 11: Ablauf einer CANCEL-Anfrage

Die Schritte 1 bis 4 verlaufen wie in jeder *INVITE*-Anfrage. Nun soll die Anfrage abgebrochen werden. Dafür startet der Client eine komplett neue Anfrage, die *CANCEL*-Anfrage, für die durch Schritt 5 eine neue Transaktion (CT2) eröffnet wird. Nach dem 6. Schritt wurde im *User Agent Server* (UAS) die entsprechende serverseitige Transaktion gestartet, die die *CANCEL*-Anfrage verwaltet und sie an die Applikationsebene des UAS weiterleitet. Erst auf dieser Ebene kann die Abbruchanfrage der Einladungsanfrage zugeordnet und darauf reagiert werden. Die Applikationsebene des UAS antwortet ihrer ST1 mit einer „487“-Antwort, was für „Request Terminated“ steht, und an ST2 wird eine „200“-Antwort gesendet. Beide serverseitigen Transaktionen übertragen die Antworten an den UAC und beenden sich regulär. Selbiges machen die Client-Transaktionen nachdem sie die empfangenen Antworten an ihre Applikationsebene weitergegeben haben.

3.3.3. Ablauf BYE

Abbildung 12 zeigt die mit einer *BYE*-Anfrage verbundenen Transaktionen.

Voraussetzung ist eine etablierte Sitzung, so dass eine *INVITE-Transaktion* nicht mehr existiert.

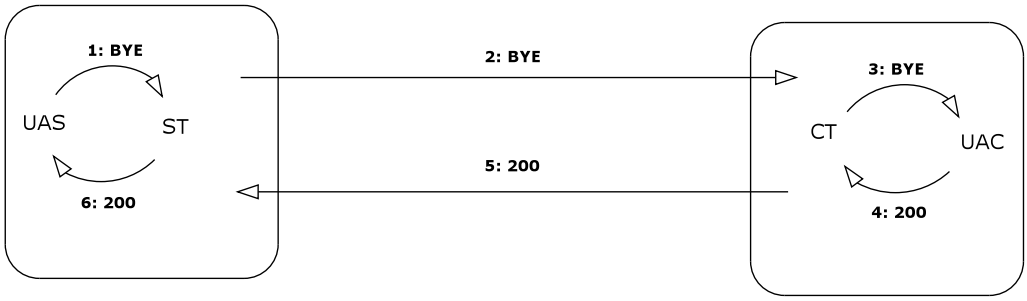

Abbildung 12: Ablauf einer BYE Anfrage

Die SIP-Applikation des *UAS* stößt das Senden einer *BYE*-Anfrage an. Es wird eine *Non-INVITE Server-Transaction* (ST) generiert, die die *BYE*-Anfrage an den *UAC* sendet. Der *UAC* generiert bei Erhalt der *BYE*-Anfrage eine *Non-INVITE Client Transaction* (CT). Diese leitet die Anfrage an die SIP-Applikation weiter. Die SIP-Applikation generiert eine *200 OK*-Antwort, die an die CT "gesendet" wird und beendet die Sitzung. Die CT leitet die Antwort an den *UAS* weiter und beendet sich. Bei Erhalt der *200 OK*-Antwort leitet die ST die Antwort an die SIP-Applikation und beendet sich sofort. Die SIP-Applikation beendet dann die Sitzung auf *UAS*-Seite.

4. Implementierung der Grundmodule (jb)

Im Folgenden wird zusammenfassend die Implementierung der eingeführten Nachrichten und Module vorgestellt und auf deren Besonderheiten eingegangen. Die Implementierung ganzer Anwendungen, wie *User Agents* oder *Proxys* folgt dann in späteren Kapiteln.

4.1. Implementierung der SIP-Nachrichten

sipRequest und *sipResponse* werden von der **OMNeT++** -Klasse *cMessage* abgeleitet. Diese stellt die grundlegenden Funktionalitäten, die zum Versand von Nachrichten in **OMNeT++** benötigt werden zur Verfügung. Das **OMNeT++** -interne „Message-Subclassing"-Konzept wird nicht verwendet, da die Daten, die eine SIP-Nachricht benötigt, dafür zu heterogen und umfangreich sind. Abbildung 13 zeigt das Klassendiagramm der SIP-Nachrichten *sipRequest* und *sipResponse*.

4.1.1. sipInterfacePacket

Im folgenden sind die API-Funktionen des InterfacePackets kurz aufgelistet:

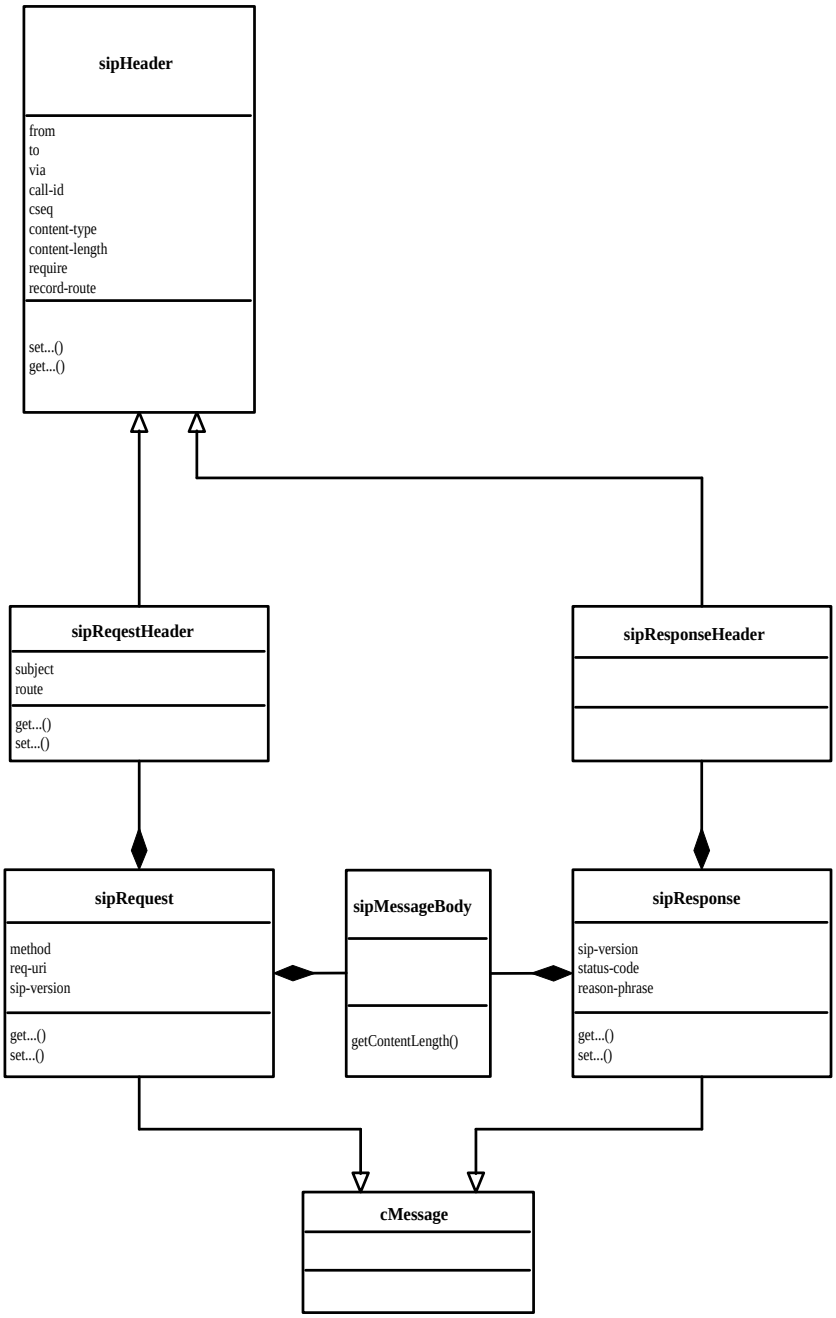

Abbildung 13: Aufbau der SIP-Nachrichten

action() gibt die Nachrichtenart des Interface-Pakets zurück. Die Nachrichtenart ist als integer-Wert kodiert, für den in der *sipInterfacePacket.h*-Datei Konstanten definiert sind.

request(sipRequest *request) wird verwendet um *sipRequest*-Nachrichten innerhalb der SIP-Anwendung weiter zu leiten.

getRequest() gibt ein vorher eingekapseltes *sipRequest*-Objekt zurück.

respond(sipResponse *response) wird verwendet um *sipResponse*-Nachrichten innerhalb der SIP-Anwendung auszutauschen.

getResponse() gibt ein vorher eingekapseltes *sipResponse*-Objekt zurück.

timeoutRequest(sipRequest *request) wird zur Benachrichtigung einer Zeitüberschreitung eines Anfragevorgangs benutzt. Die originale Anfrage muss mit übergeben werden und kann mit *getRequest()* wieder abgerufen werden.

timeoutResponse(sipResponse *response) das Gleiche wie oben nur für Anfragen.

terminate() wird benötigt, um einem anderen SIP-Modul der Anwendung zu signalisieren, dass diese sich beenden soll.

closeSession() signalisiert dem Sender einer *terminate()*-Nachricht, dass dieses Modul zerstört werden kann.

fromSocket(sipRequest *request) zur Weiterleitung einer durch den Socketlayer empfangenen SIP-Anfrage.

Das sipInterfacePacket besitzt folgende Aktionen:

SA_REQUEST die eingekapselte Nachricht ist eine Anfrage

SA_RESPONSE die eingekapselte Nachricht ist eine Antwort

SA_CLOSE_SESSION wird von Transaktionen verwendet, um dem *sipServerModule* bzw. *sipClientModule* zu signalisieren, dass sie gelöscht werden soll.

SA_OPEN_PORT wird von der Applikation verwendet, um dem *sipServerModule* zu signalisieren, dass dieses einen Port zur *SocketLayer* öffnen soll.

SA_TIMEOUT_REQUEST wird von der Client-Transaktion verwendet, um der Applikation zu signalisieren, dass bei der Übertragung der Anfrage ein Fehler aufgetreten ist.

SA_TIMEOUT_RESPONSE wird von der Server-Transaktion verwendet, um der Applikation zu signalisieren, dass bei der Übertragung der Antwort ein Fehler aufgetreten ist.

4.1.2. sipHeader

Im *sipHeader* sind die meisten Daten, die eine SIP-Nachricht ausmachen, gekapselt. Da für die Kopf-Felder mehrere Randbedingungen gelten, können sie nicht als **OMNeT++** -Parameter implementiert werden. So gilt für bestimmte Kopf-Felder zum Beispiel, dass sie mehrfach auftreten können und ihre Reihenfolge von Bedeutung ist. Ein Beispiel dafür ist das *Via*-Feld. Die Reihenfolge der *Via*-Felder gibt die Route an, die eine SIP-Nachricht genommen hat und darf somit nicht verändert werden. Der *sipHeader* stellt Methoden zur Verfügung, die für die Manipulierung der einzelenen Felder benötigt werden.

Spezifische Felder werden nur von Anfrage- oder Antwort-Nachrichten benötigt. Daraus resultiert die Vererbung zu *sipRequestHeader* und *sipResponseHeader*.

4.1.3. sipRequest

Der *sipRequest* stellt diverse Methoden zur Einrichtung benötigter Daten bereit. Sie sind nach den Anfrage-Methoden aus Abschnitt 2.2.3 benannt und spiegeln dadurch ihre Funktion wider:

invite(from, to, call_id, cseq, via) richtet die Anfrage als *INVITE*-Anfrage ein. Die Argumente haben die folgenden Typen:

1. sipAddress *from, *to
2. char *call_id
3. int cseq
4. sipTagVia *via

ack(sipRequest *request) richtet die Anfrage als *ACK*-Anfrage ein. Die benötigten Daten werden der übergebenen Anfrage entnommen.

cancel(sipRequest *request) richtet die Anfrage als *CANCEL*-Anfrage ein. Die benötigten Daten werden aus der übergebenen Anfrage entnommen.

bye(sipRequest *request) richtet die Anfrage als *BYE*-Anfrage ein. Die benötigten Daten werden der übergebenen Anfrage entnommen.

4.1.4. sipResponse

Die Klasse *sipResponse* repräsentiert eine SIP-Antwort. Sie wird von einem *UAS* (*INVITE*-Anfrage) bzw. *UAC* (sonst) generiert und ist die Antwort auf eine Anfrage. Eine Antwort besitzt einen Statuscode (siehe Kapitel 2.2.4), der angibt, ob die Bearbeitung der Anfrage erfolgreich war. Zusätzlich besitzt eine Antwort einen Begründungstext. Diese gibt in textueller Form das Ergebnis der Anfragenbearbeitung an, so dass auch ein Benutzer in der Lage ist, die Antwort zu

deuten. Einen Antwort enthält, wie eine Anfrage, einen Methodenparameter, der ausgelesen werden kann. Dieser Parameter ist nicht wie in der Anfrage als Variable realisiert, sondern wird aus dem *CSeq*-Headerfeld der Antwort ausgelesen. Eine Antwort benötigt keine zusätzlichen Headerfelder zur Klasse *sipHeader*. Daher ergeben sich für *sipResponse* folgende Methoden:

setStatusCode(int status) setzt den Statuscode der Antwort

setReasonPhrase(char *reason) setzt den Begründungstext der Antwort

getStatuscode() gibt den Statuscode der Antwort zurück

getReasonPhrase() gibt den Begründungstext der Antwort zurück

getMethod() gibt die Methode der Antwort zurück

4.2. Implementierung der Module

Im Folgenden wird auf die Implementierung der gemeinsamen Module eingegangen. Die speziellen Module werden in gesonderten Kapiteln einzeln erläutert.

4.2.1. sipClientTransaction

Das Modul *sipClientTransaction* implementiert den vollen Umfang der gleichnamigen Transaktion aus Abschnitt 3.1.1. Dieses Modul wird dynamisch vom *sipClientModule* erzeugt und nach Beendigung seiner Aufgabe wieder zerstört.

Oberklasse	cSimpleModule
Eingangs-Tore	fromApp fromSocketLayer
Ausgangs-Tore	toSocketLayer
Parameter	debug : boolean

Tabelle 12: Eigenschaften der sipClientTransaction

Nachdem es erzeugt wurde, arbeitet das Modul ähnlich einem endlichen Automaten eine der Zustandsfolgen aus den *(Non-)INVITE* Client-Transaktionen aus Abschnitt 2.4.1 und 2.4.2 ab. Wenn es schließlich den Zustand *TERMINATED* erreicht, werden alle untergeordneten Objekte gelöscht und eine Nachricht an das *sipClientModule* gesendet, dass es dieses Modul entfernen kann.

Das genaue Zustandsüberführungsdiagramm des Moduls ist in Tabelle 13 dargestellt. Zustand „0“ ist der Start- und Zustand „4“ der Finalzustand. „A/E“,

	INV	Non-INV	1xx	2xx	3xx-6xx	A/E	B/F	D/K
0 = INITIALIZE	1	5	Err	Err	Err	Err	Err	Err
1 = INV_CALLING	Err	Err	2	4	3	1	4	Err
2 = INV_PROCEEDING	Err	Err	2	4	3	Err	Err	Err
3 = INV_COMPLETE	Err	Err	Err	Err	Err	Err	Err	4
4 = TERMINATED	Err	Err	Err	Err	Err	Err	Err	Err
5 = NON_INV_TRYING	Err	Err	6	7	7	5	4	Err
6 = NON_INV_PROCEEDING	Err	Err	6	7	7	6	4	Err
7 = NON_INV_COMPLETE	Err	Err	Err	Err	Err	Err	Err	4

Tabelle 13: Zustandsüberführungsdiagramm: sipClientTransaction

„B/F“ und „D/K“ sind jeweils zwei Zeitgeber, von denen der erstgenannte nur in *INVITE*-Transaktionen und letzterer nur in *Non-INVITE*-Transaktionen auftritt. „Err“ stellt einen Zustandsübergang zu Zustand „4“ mit gleichzeitiger Fehlerausgabe an das *sipClientModule* dar. Die genauen Aktionen, die bei Eintritt in einen der Zustände erfolgen, können den Abbildungen 5 und 6 ab Seite 12 entnommen werden.

Die Schnittstellen-Eigenschaften der sipClientTransaktion, wie Parameter oder Tore (Gates) zu anderen **OMNeT++** -Modulen können der Tabelle 12 entnommen werden.

4.2.2. sipClientModule

Das dynamische Erzeugen der Transaktionsmodule, deren weitere Verwaltung sowie deren letzliche Zerstörung, ist eine für alle SIP-Anwendungen gleiche Aufgabe. Somit wird diese in das *sipClientModule* ausgelagert. Dieses Modul nimmt vom Anwendungsmodul die Anfragen entgegen und gibt sie an das zuständige Transaktionsmodul weiter. Existiert noch kein Transaktionsmodul, so wird es dynamisch erzeugt und mit den notwendigen Initialwerten gestartet. Außerdem werden die notwendigen Verbindungen zwischen dem Transaktions- und dem Socketlayer-Modul hergestellt. Hat ein Transaktionsmodul Nachrichten an das Anwendungsmodul zu senden, so geschieht das über dieses Modul, da die Transaktionsmodule nicht die „Adresse“ des Anwendungsmoduls kennen. Auf Aufforderung einer Transaktion selbst oder der des Anwendungsmoduls beendet und löscht dieses Modul untergeordnete Transaktionsmodule. Tabelle 14 zeigt die Schnittstellen-Eigenschaften des sipClientModule.

4.2.3. sipServerTransaction

Die *sipServerTransaction* wird dynamisch vom *sipServerModule* generiert, wenn eine neue Anfrage empfangen wird, die keiner existierenden Transaktion zugeordnet werden kann. Auch hier wird ein Zustandsautomat verwendet. Es werden, im Gegensatz zum *sipClientModule*, keine unterschiedlichen Zustände für die *INVITE*- und *Non-INVITE*-Transaktion definiert, da sich beide in ihrem Verhal-

Oberklasse	cSimpleModule
Eingangs-Tore	fromApp fromClientTrans fromSocketLayer
Ausgangs-Tore	toApp toClientTrans toSocketLayer
Parameter	debug : boolean commonName : string

Tabelle 14: Eigenschaften des sipClientModule

ten sehr ähneln. Ein wesentlicher Unterschied ist, dass der Zustand

TRYING nur für die Non-INVITE-Transaktion und

CONFIRMED nur für die INVITE-Transaktion existiert.

Alle weiteren Zustände existieren sowohl für die INVITE- als auch für die Non-INVITE-Transaktion. Geht eine Transaktion in den Zustand *TERMINATED* über, so wird sie vom *sipServerModule* sofort gelöscht. Dafür sendet die Transaktion ein *sipInterfacePacket* mit der Aktion *SA_CLOSE_SESSION* an das *sipServerModule*.

	INV	Non-INV	1xx	2xx	3xx-6xx	G	H	I	J
0 = INIT	2	1	Err	Err	Err	Err	Err	Err	Err
1 = TRYING	-	Err	2	INV: -, Non-INV: 3	INV: -, Non-INV: 3	Err	Err	Err	Err
2 = PROCEEDING	2	2	2	INV: 5, Non-INV: 3	3	Err	Err	Err	Err
3 = COMPLETED	3	3	Err	Err	Err	3	5	Err	5
4 = CONFIRMED	Err	-	Err	Err	Err	Err	Err	5	Err
5 = TERMINATED	Err	Err	Err	Err	Err	Err	Err	Err	Err

Tabelle 15: Zustandsüberführungsdiagramm: sipServerTransaction

Die verwendeten Symbole in Tabelle 15 sind bereits im Kapitel über die *sipClientTransaction* erklärt wurden. Die Spalten *INV* und *Non-INV* repräsentieren eingehende Nachrichten. *INV* steht für eine *INVITE*-Anfrage, *Non-INV* für alle restlichen Anfragen. *INV* betrifft somit das INVITE Server-Transaktionsmodell, *Non-INV* das Non-INVITE Server-Transaktionsmodell. Die Spalten *1xx*, *2xx* und *3xx-6xx* repräsentieren die Statusklassen der empfangenen Antworten. Die Spalten *G*, *H*, *I* und *J* repräsentieren Zeitgeber, die feuern.
Es wird ein *Err* angenommen, wenn das Verhalten für einen Fall im Transaktionsmodell nicht angegeben ist. Dies ist unabhängig von der Implementierung. In diesem Fall wird die eingehende Nachricht ignoriert.

4.2.4. sipServerModule

Das *sipServerModule* wird entsprechend dem Entwurf aus Kapitel 3.1.3 implementiert. Es enthält die volle Funktionalität des Entwurfs.
Empfangene *ACK*-Anfragen von der *SocketLayer* werden entsprechend der davor gesendeten Antwort auf die *INVITE*-Anfrage behandelt:

2xx Antwort Es wird eine Sitzung aufgebaut. Die *INVITE Server-Transaction* wurde bei Erhalt der *200 OK*-Antwort gelöscht. Die *ACK*-Anfrage wird direkt an die SIP-Applikation weitergeleitet und betrifft keine Transaktion.

3xx-699 Antwort Es wird keine Sitzung aufgebaut, da bei der Bearbeitung der *INVITE*-Anfrage ein Fehler aufgetreten ist. Die *INVITE Server-Transaction* existiert noch. Daher leitet das *sipServerModule* die *ACK*-Anfrage an die Transaktion weiter. Die SIP-Applikation ist von dieser Anfrage unberührt.

5. SIP User-Agents (sg)

SIP User Agents (UA) repräsentieren die Endsysteme des SIP-Systems. UAs bestehen aus den Elementen eines Clients und eines Servers. Die Elemente des Clients werden durch den *UAC* repräsentiert und übernehmen das Senden von Anfragen und Empfangen von Antworten. Die Elemente des Servers werden durch den *UAS* repräsentiert und übernehmen das Bearbeiten von Anfragen sowie das Generieren und Senden von Antworten. Zusätzlich enthält ein SIP-UA eine übergeordnete Instanz, die den *UAC* und den *UAS* koordiniert. Diese stellt somit die Intelligenz des Endsystems dar.
Zunächst soll noch einmal auf allgemeine Eigenschaften einer Sitzung eingegangen werden. Danach werden die speziellen Aufgaben und die Implementierung beschrieben.

5.1. Die Sitzung

Eine Sitzung unterscheidet sich zum Dialog darin, dass der Aufbau eines Dialoges notwendig für den Aufbau einer Sitzung ist, aber nicht aus dem Aufbau eines Dialoges zwangläufig auch der Aufbau einer Sitzung resultiert.
Eine Sitzung wird durch den sogenannten "offer/answer exchange" aufgebaut. Zunächst wird ein "offer" gesendet, das alle festzulegenden Parameter für die Sitzung enthält. Daraufhin sendet der Kommunikationspartner eine "answer". Signalisiert die "answer", daß das "offer" akzeptiert wird, wird eine Sitzung aufgebaut.
Eine Sitzung wird z.B. aufgebaut, wenn auf eine *INVITE* Anfrage eine *200 OK* Antwort empfangen wurde, und das "offer" akzeptiert wird. Wird eine *200 OK*

Antwort empfangen, das "offer" aber nicht akzeptiert, so wird nur ein Dialog generiert, aber keine Sitzung.
Das "offer" kann in der *INVITE* Anfrage des als UAC oder der *200 OK* Antwort des als UAS agierenden UA enthalten sein. Die "answer" wird in die nächste gültige Nachricht eingefügt. Laut RFC 3261 ist dies die *200 OK* Antwort des als UAS agierenden bzw. die *ACK* Anfrage des als UAC agierenden UA.

Eine Sitzung besitzt Parameter, die sie eindeutig identifizieren. Diese sind die Parameter eines Dialoges[5]. Eine Sitzung bzw. ein Dialog besitzt für die eindeutige Identifizierbarkeit eine ID. Diese besteht aus der *Call-ID* und den Tags der Adressfelder (*localTag, remoteTag*). Weiterhin werden die lokale (*localCSeq*) und entfernte (*remoteCSeq*) Sequenznummer, die lokale (*localUri*) und entfernte (*remoteUri*) Adresse (URI), die entfernte Zieladresse *remoteTarget*, der secure-Parameter (*secure*) und die Route (*route*) gespeichert.
Die lokale Sequenznummer speichert die zuletzt verwendete Paketnummer selbst versendeter Anfragen. Die entfernte Sequenznummer speichert die zuletzt verwendete Paketnummer des Kommunikationspartners.
Die Adressen geben die eigene bzw. die Adresse des Kommunikationspartners an. Die entfernte Zieladresse gibt die Adresse an, unter der der Kommunikationspartner kontaktiert werden möchte.
Der secure-Parameter gibt an, ob die Kommunikation verschlüsselt stattfinden soll.
Die Route gibt an, welche Hosts bei der Übertragung von Paketen durchlaufen werden sollen.

5.2. Aufbau eines User Agents

Auf oberster Ebene existiert ein Host-Prozess, der Dialoge generiert, verwaltet und beendet. Die generierten Dialoge können als Client und als Server agieren. Für die Zeit des Dialogaufbaus sind die Rollen festgelegt. Der UA, der eine Anfrage stellt, agiert als UAC. Der UA, der eine Anfrage erhält, agiert als UAS. Innerhalb des Dialoges, auch für das Beenden des Dialoges, dürfen die Rollen getauscht werden.
Jeder UA besitzt je eine Instanz des Client- und Servermoduls der Transaktionsebene, und kann somit die bereits vorgestellten Funktionalitäten eines Clients und eines Servers nutzen.

In Abbildung 14 sind beispielhaft vier Dialoge dargestellt.

[5] vgl. auch [R+02] Kap. 12

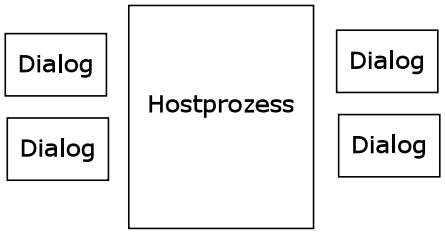

Abbildung 14: Aufbau eines User Agents

5.3. Generieren von Anfragen

Eine gültige Anfrage eines UAs besitzt die Headerfelder[6]

- To,
- From,
- Sequenznummer,
- Call-ID,
- Max-Forwards,
- Via.

Die Headerfelder wurden bereits im Kapitel 2.3 beschrieben.
Eine Anfrage kann die Methoden *INVITE*, *ACK*, *CANCEL*, *BYE* oder *REGISTER* enthalten. In RFC 3261 ist außerdem die Methode *OPTIONS* definiert, die aber nicht im Rahmen dieser Arbeit implementiert wurde.
Besonderheiten einer Anfrage bezüglich eines Dialoges werden in Kapitel 5.5 diskutiert.

5.4. Bearbeiten von Anfragen

Erhält ein UA eine Anfrage, so führt er zunächst allgemeine Überprüfungen durch, um festzustellen, ob die empfangene Anfrage gültig ist. Danach erfolgt eine methoden- und dialogabhängige Verarbeitung, die in Kapitel 5.5 betrachtet werden soll.
Zu den allgemeinen Überprüfungen[7] gehören:

Überprüfung der Methode Der UA überprüft, ob er die Methode der Anfrage versteht. Ist dies nicht der Fall, generiert er eine *405 Method Not Allowed*

[6] vgl. auch [R+02] Kap. 8.1.1
[7] vgl. auch [R+02] Kap. 8.2

Antwort, die ein *Allow* Headerfeld enthält. Dieses gibt die Methoden an, die der UA versteht. Versteht der UA die Methode, wird die Anfrage weiter bearbeitet.

Überprüfung der Headerfelder Der UA überprüft die Headerfelder *To*, *Request-URI* und *Route*. Ist die Anfrage an einen UA gestellt, den der UA nicht bedient bzw. nicht er selbst ist, so generiert er eine *404 Not Found* Antwort.

Überprüfung nachrichteninhalt-spezifischer Headerfelder Der UA überprüft Headerfelder, die mit dem Nachrichteninhalt in Zusammenhang stehen. Diese sind *Content-Type*, *Content-Language*, *Content-Encoding* und *Supported*. Sie dienen dem Aufbau einer Sitzung. Zunächst wird *Supported* überprüft. Werden alle Erweiterungen in *Supported* unterstützt, werden die restlichen Headerfelder überpüft. Enthält mindestens eins dieser Headerfelder einen Wert, der nicht unterstützt wird, wird eine *415 Unsupported Media Type* Antwort generiert. Entsprechend der nicht unterstützten Anforderungen, enthält die Antwort weitere Headerfelder, die die unterstützten Typen, Sprachen bzw. Kodierungen enthält. SIP stellt dafür die Headerfelder *Accept*, *Accept-Language* und *Accept-Encoding* zur Verfügung.

Überprüfung der Erweiterungen Der *UA* überprüft, welche Erweiterungen beim Generieren der Antwort verwendet werden können. Es dürfen nur Erweiterungen verwendet werden, die im *Suppported*-Headerfeld der Anfrage vorhanden sind, da dieses die Erweiterungen enthält, die der andere *UA* unterstützt. Wird eine zusätzliche Erweiterung benötigt, so wird im Normalfall auf Standard-Erweiterungen zurückgegriffen. Ist eine Bearbeitung ohne eine spezielle Erweiterung nicht möglich, so kann der *UA* eine *421 Extension Required*-Antwort generieren. Diese enthält ein zusätzliches Headerfeld *Require*, das die benötigte Erweiterung enthält. Dieses Verhalten wird allerdings nicht empfohlen, da die Kommunikation unterbrochen wird.

Sind die Headerfelder fehlerfrei, generiert der UA eine Antwort auf die Anfrage. Zu beachten ist, dass provisorische Antworten nur auf eine *INVITE* Anfrage generiert werden. Provisorische Antworten dienen dazu, zu signalisieren, dass die Anfrage bearbeitet wird. Bei einer *INVITE* Anfrage ist eine Benutzereingabe notwendig. Dies wird im Normalfall mehr Zeit in Anspruch nehmen, als üblich. Auf Anfragen anderer Methoden wird keine provisorische Antwort gesendet.

Wichtige Statuscodes sind im Anhang A aufgeführt.

5.5. Erweiterte Dialog-Methodensyntax

Es gibt einige Besonderheiten bezüglich der Methoden. Diese werden im Folgenden beschrieben.

5.5.1. INVITE-Anfrage

Eine *INVITE* Anfrage kann zusätzliche Headerfelder[8] enthalten, die die Eigenschaften der Sitzung festlegen.
Folgende Headerfelder sollten vorhanden sein:

Allow gibt die Methoden an, die der UA unterstützt,

Supported gibt die Erweiterungen an, die der UA unterstützt.

Folgende Headerfelder können ausserdem enthalten sein:

Accept gibt an, welche Content-Typen der UA versteht,

Expires gibt den Zeitraum an, in dem die Einladung gültig ist.

Je nachdem, ob der adressierte UA der Sitzung beitreten möchte, generiert er eine 2xx bzw. 3xx-699 Antwort. Eine Antwort der Statusklasse 2 signalisiert, dass der UA der Sitzung beitreten möchte. Eine Antwort der Statusklasse 3-6 signalisiert, dass bei der Bearbeitung der Anfrage ein Fehler aufgetreten ist, und keine Sitzung aufgebaut wird.
Erhält der UA, der die Anfrage gestellt hat, eine 2xx Antwort, sendet er eine *ACK*-Anfrage auf diese Antwort. Durch diese Antwort wird der Dialog generiert. Wurde die Sitzungsbeschreibung angenommen, wird gleichzeitig eine Sitzung etabliert.
Die Headerfelder der *ACK*-Anfrage stimmen mit den Headerfeldern der *INVITE*-Anfrage überein. Ausnahme ist das Methodenfeld der Sequenznummer. Dies nimmt den Wert *ACK* an. Damit ist auch die Methode der Anfrage *ACK*.

5.5.2. CANCEL-Anfrage

Eine *CANCEL*-Anfrage wird gesendet, um eine schwebende Sitzung zu beenden. Die Headerfelder dieser Anfrage stimmen bis auf das Methoden-Headerfeld mit den Headerfeldern der *INVITE*-Anfrage überein. Die Methode ist *CANCEL*.

5.5.3. BYE-Anfrage

Eine *BYE*-Anfrage wird gesendet, um eine etablierte Sitzung zu beenden. Es gelten die gleichen Regeln wie für eine *CANCEL*-Anfrage, nur dass die Methode *BYE* ist.

[8] vgl. auch [R+02] Kap. 13.2.1

5.6. Implementierung

Alle im vorigen Kapitel beschriebenen Verhaltensweisen und Antwortklassen werden nicht implementiert. Nicht implementiert werden:

- Überprüfung der Richtigkeit verschiedener Headerfelder wie z.B. der *Request-URI* und *Route* und nachrichteninhalt-spezifischer Headerfelder
- es existiert keine Anbindung an SDP; es werden auch nicht die SDP-spezifischen bzw. nachrichteninhalt-spezifischen Headerfelder überprüft
- Es werden nicht alle Statuscodes implementiert. Es gibt nur Antworten mit den Statuscodes
  - 100 Trying,
  - 180 Ringing,
  - 183 Session Progress,
  - 200 OK,
  - 300 Multiple Choices,
  - 301 Moved Permanently,
  - 302 Moved Temporarily,
  - 403 Forbidden,
  - 404 Not Found,
  - 486 Busy Here.

Der UA wird durch drei Klassen implementiert. *sipApplication* stellt den Hostprozess dar. *sipClientDialog* ist ein Dialog auf Clientseite. Er enthält nicht die vollständige Funktionalität eines Dialoges, sondern dient dem Zuordnen von eingehenden Nachrichten auf Clientseite. Erhält ein UA mehrere Antworten auf eine *INVITE*-Anfrage, muss eine Zuordnung der Antwort zu einem Dialog garantiert werden. Bei mehreren Antworten werden mehrere Dialoge generiert, die unterschiedliche IDs besitzen. Die generierten Dialoge sind Objekte der Klasse *sipDialog* und besitzen die vollständige Funktionalität. Die *sipApplication* generiert auf Serverseite sofort Dialoge der Klasse *sipDialog*.

5.6.1. Die Klasse sipApplication

Die Klasse *sipApplication* empfängt Nachrichten vom *serverModule* und *clientModule* und leitet diese an den entsprechenden Dialog weiter.
Nachrichten vom *serverModule* sind Anfragen und betreffen die Serverseite des *UA*. Sie werden direkt an das entsprechende Objekt der Klasse *sipDialog* weitergeleitet. Existiert kein dazugehöriger Dialog und findet die Kommunikation

außerhalb eines Dialoges statt, so wird ein neuer Serverdialog generiert. Enthält das *To*-Feld der Anfrage ein Tag, wurde die Anfrage innerhalb eines Dialoges gesendet. Wird kein dazugehöriger Dialog gefunden, so ist der Dialog abgestürzt. In dieser Implementierung ist es nicht möglich, den Dialog erneut zu generieren. Es wird eine Fehlermeldung an den sendenden *UA* geschickt.
Nachrichten vom *clientModule* sind Antworten und betreffen die Clientseite des *UA*. Es existiert ein "Oberdialog" der Klasse *sipClientDialog*. An diesen wird die Antwort weitergeleitet. Dieser übernimmt das Weiterleiten an den entsprechenden "Unterdialog". Der "Unterdialog" ist ein Dialog der Klasse *sipDialog*.
Die Dialoge werden über den Namen eindeutig identifiziert. Für Objekte der Klasse *sipDialog* gilt
Name = Call-ID + localTag + remoteTag.
Dies gilt auch für Dialoge, die von einem Objekt der Klasse *sipClientDialog* generiert wurden. Für Objekte der Klasse *sipClientDialog* ist der Name
Name = Call-ID + localTag + NULL
festgelegt. Damit wird sichergestellt, dass Antworten von mehreren *UA*s dem Dialog der ursprünglichen *INVITE*-Anfrage zugeordnet werden können. Für jede erhaltene Antwort wird dann ein neues Objekt der Klasse *sipDialog* generiert.
Auf Clientseite übernimmt die *sipApplication* das Generieren der Call-ID und des Tags des *From* Headerfeldes (*localTag*). Auf Serverseite wird ebenfalls das Tag generiert. Auf Serverseite ist dies ebenfalls das *localTag*, es ist aber das Tag des *To* Headerfeldes. Dieses Tag wird später vom antwortenden UAS in die Antwort eingefügt. In der *INVITE*-Anfrage ist dieses Tag null.

5.6.2. Die Klasse sipClientDialog

Die Klasse *sipClientDialog* übernimmt das Empfangen von Antworten auf der Clientseite und das Generieren und Verwalten von "richtigen" Dialogen auf Clientseite.
Ein Objekt der Klasse *sipClientDialog* besitzt alle Parameter eines Dialoges und die Funktionalität des UAC. Die ID des Dialoges besteht aus der *Call-ID* und dem Tag aus der eigenen Adresse. Durch diese ID wird garantiert, dass mehrere Antworten auf ein *INVITE* dem Originaldialog zugeordnet werden können.
Erhält der *sipClientDialog* eine Antwort, generiert er ein Objekt der Klasse *sipDialog* und übergibt alle Parameter inklusive des Tags der entfernten Zieladresse. *sipClientDialog* dient nun dem Vermitteln zwischen der *sipApplication* und dem *sipDialog*. Nachrichten von der *sipApplication* werden an den *sipDialog* weitergeleitet und umgekehrt.
Der *sipClientDialog* generiert für jede 1xx Antwort, außer *100 Trying*, eine Instanz der Klasse *sipDialog*. Somit wird in jedem Fall ein *early dialog* generiert. Jede 1xx Antwort außer der *100 Trying* enthält ein Tag im *To* Headerfeld.
Die Werte der Parameter eines Dialoges werden den Headerfeldern der Anfrage entnommen (siehe Tabelle 16).

Parameter	Headerfeld
remoteUri	To
localUri	From
Call-ID	Call-ID
localTag	Tag des From Feldes
remoteTag	Tag des To Feldes
localCSeq	CSeq
remoteCSeq	0
secure	secure
route	route

Tabelle 16: Parameterwerte eines Dialoges auf Clientseite

5.6.3. Die Klasse sipDialog

Die Klasse *sipDialog* repräsentiert einen SIP-Dialog. Sie enthält alle Parameter, die von einem Dialog gespeichert werden müssen[9], und die komplette Funktionalität des Dialoges. Die Werte der Parameter sind in Tabelle 17 dargestellt.

Parameter	Headerfeld
remoteUri	From
localUri	To
Call-ID	Call-ID
localTag	Tag des To Feldes
remoteTag	Tag des From Feldes
localCSeq	0
remoteCSeq	CSeq
secure	secure
route	route

Tabelle 17: Parameterwerte eines Dialoges auf Serverseite

Ein Objekt der Klasse *sipDialog* kann *CANCEL*- und *BYE*-Anfragen stellen. Es übernimmt auch das Senden der *ACK*-Anfrage für eine erfolgreiche Antwort auf eine *INVITE*-Anfrage.
Wurde eine Sitzung erfolgreich aufgebaut, so wird eine *RTP*-Verbindung aufgebaut und Daten gesendet.

[9] vgl. auch [R+02] Kap. 12

6. Der Proxy (jb)

SIP-Proxys sind die Elemente der SIP-Nachrichtenkette, die SIP-Requests (Anfragen) zu User Agent Servern (UAS) und SIP-Responses (Antworten) zu User Agent Clients (UAC) weiterleiten. Dabei kann eine Anfrage auf dem Weg zu ihrem Ziel mehrere SIP-Proxys passieren. Die darauf folgende Antwort passiert auf ihrem Weg zum UAC die gleichen Proxys, wie die Anfrage, nur in umgekehrter Reihenfolge. Ein SIP-Proxy trifft zur Weiterleitung einer Nachricht Routing-Entscheidungen und hat die Nachricht entsprechend Regeln, die im Folgenden erläutert werden, zu verändern.

Es gibt für SIP-Proxys zwei verschiedene Arbeitsweisen: „stateful“ und „stateless“ (zustandsorientiert / Nicht zustandsorientiert). Arbeitet ein Proxy stateless, dann agiert er als reines „Weiterleitungselement“ in der Nachrichtenkette, das Anfragen stromabwärts in Richtung des UAS und Antworten stromaufwärts in Richtung UAC weiterleitet. Jegliche Information über die Nachricht wird sofort nach der Weiterleitung verworfen. Ein zustandsorientierter Proxy hingegen behält Informationen über alle eingehenden Anfragen, über aus diesen resultierende, ausgehende Anfragen und deren Ergebnisse. Die gespeicherten Informationen werden bei der Bearbeitung späterer Anfragen herangezogen. Ein zustandsorientierter Proxy kann Anfragen verzweigen („forking“). Eine Anfrage wird dabei gleichzeitig an verschiedene Ziele geleitet.

Die grundlegende Arbeitsweise eines SIP-Proxy ist in [R+02] Kap. 16 beschrieben. Wir beschränken uns in dieser Arbeit auf die Schritte, die in den Abschnitten 6.2 und 6.3 behandelt werden.

6.1. Aufbau eines SIP-Proxy

Ein SIP-Proxy kann prinzipiell wie folgt aufgebaut werden[10]:

Auf oberster Ebene existiert ein Proxy-Prozess, der für jede eingehende Anfrage einen Server-Prozess und ein oder mehrere Client-Prozesse öffnet.

Die Client-Prozesse senden Kopien der Anfrage an unterschiedliche Ziele und deren Ergebnisse zurück an den „Proxy-Prozess“, der die Antworten zusammenfasst und an den Absender der Anfrage sendet.

6.2. Behandlung von Anfragen

Für jede Anfrage, die der Proxy verarbeiten soll, müssen folgende Arbeitsschritte unternommen werden:

1. Validierung der Anfrage:

[10] vgl. auch [R+02] Kap. 16.2

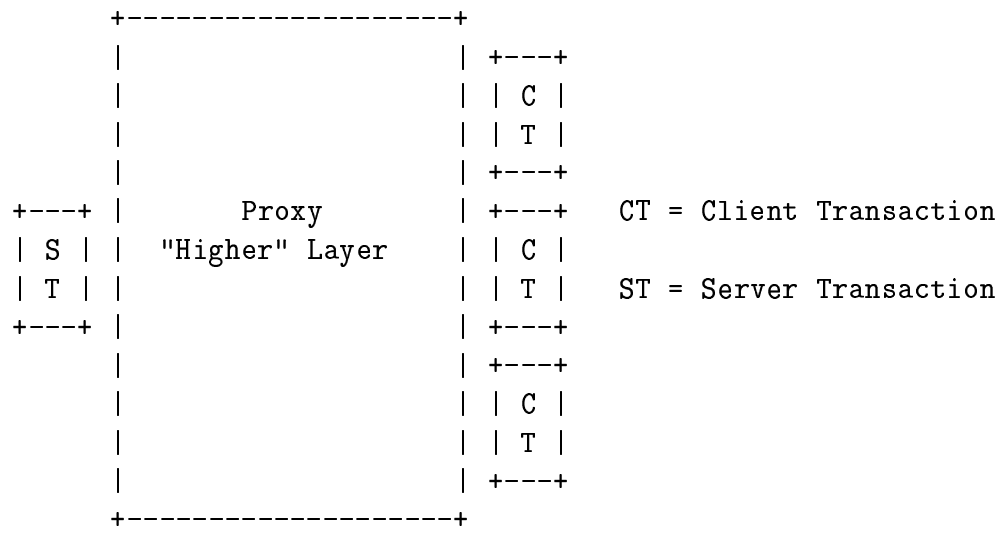

Abbildung 15: Stateful Proxy Model - vgl. [R+02] Abbildung 3

- Syntax-Prüfung
- URI-Prüfung
- *Max-Forwards*-Prüfung
- (optional) Erkennung von Kreisen in der Route
- Prüfung der Felder *Proxy-Require* und *Proxy-Authorization*

Schlägt einer dieser Tests fehl, so muss der Proxy als User Agent Server agieren und mit einem entsprechenden Fehler-Code antworten.

2. Vorverarbeitung der Routen-Informationen: Ist der erste Wert des *Route*-Headerfeldes die Adresse des Proxy, muss dieser Eintrag entfernt werden.

3. Bestimmung der anzufragenden Ziele: Ist der Proxy nicht für die Domäne, die in der Anfrage-URI angegeben ist, zuständig, so ist die URI das einzige Ziel. Ansonsten steht dem Proxy(-programmierer) frei, auf welche Weise er die anzufragenden Ziele ermittelt. In dieser Implementierung werden die Ziele durch die *sipLocationService*-Klasse bestimmt.

4. Weiterleitung der Anfrage zu den Zielen: Sobald die Zielliste des Proxy nicht mehr leer ist, kann er mit der Weiterleitung der Anfrage an die Elemente der Zielliste beginnen. Für jedes Ziel, dem der Proxy die Anfrage weiterleitet, müssen folgende Schritte unternommen werden:

   a) Erstellen einer Kopie der empfangenen Anfrage

   b) Aktualisierung der Anfrage-URI

c) Aktualisierung des *Max-Forward*-Headerfeldes

d) (optional) Hinzufügen eines *Record-Route*-Headerfeldes

e) (optional) Hinzufügen anderer Headerfelder

f) Nachbearbeiten der Routen-Informationen

g) Ermittlung der Adresse, des Ports und des Transportprotokolls der nächsten Station

h) Hinzufügen des *Via*-Headerfeldes

i) Wenn nötig, Anhängen des *Content-Length*-Headerfeld

j) Weiterleiten der Nachricht

k) Starten des Zeitgebers C (Timer C)

5. Auswertung der Antworten: wurde die Anfrage an mehrere Hosts weitergeleitet, müssen die erhaltenen Antworten zusammengefasst werden.

6.3. Behandlung von Antworten

Empfangene Antworten müssen von einem Proxy nach folgendem Schema abgearbeitet werden:

1. Suchen des zur Antwort gehörendem Client-Prozesses - konnte keiner ermittelt werden, muss der Proxy wie ein zustandsloser Proxy verfahren.

2. Aktualisierung des Zeitgebers C (Timer C).

3. Entfernung des obersten Via-Header-Feldes.

4. Zuordnung der Antwort zum „response context“ - dem zugehörigen Client-Prozess.

5. Prüfung, ob die Antwort sofort weiter geleitet werden sollte.

6. Wenn nötig, Auswahl der besten (finalen) Antwort aus dem „response context“.

Wurde keine (finale) Antwort weitergeleitet, nachdem alle assoziierten Client-Prozesse beendet wurden, muss der Proxy die beste bis dahin empfangene Antwort weiterleiten.

Es ist möglich, dass je Anfrage mehrere Antworten weiter geleitet werden: mindestens jede provisorische und jede endgültige (finale) Antwort. Die folgenden Aktionen müssen auf jede Antwort, die weiter geleitet wird, angewendet werden:

7. Zusammenfassen der Autorisierungs-Headerfeld Werte.

8. (optional) Überarbeitung der *Record-Route*-Headerfelder.

9. Weiterleitung der Antwort.

10. Erzeugung und Versand nötig gewordener *CANCEL*-Anfragen.

6.4. Implementierung

Um mehrere Kind-Prozesse verwalten zu können benötigt der *sipProxyLayer* weitere Datenstrukturen. Zum einen um die Kind-Prozesse abzubilden, zum anderen, um die Listen der Anfrageziele (Request Set) je Kind-Prozess zu verwalten. Als UML-Klassendiagramm lässt sich diese Struktur wie in Abbildung 16 darstellen.

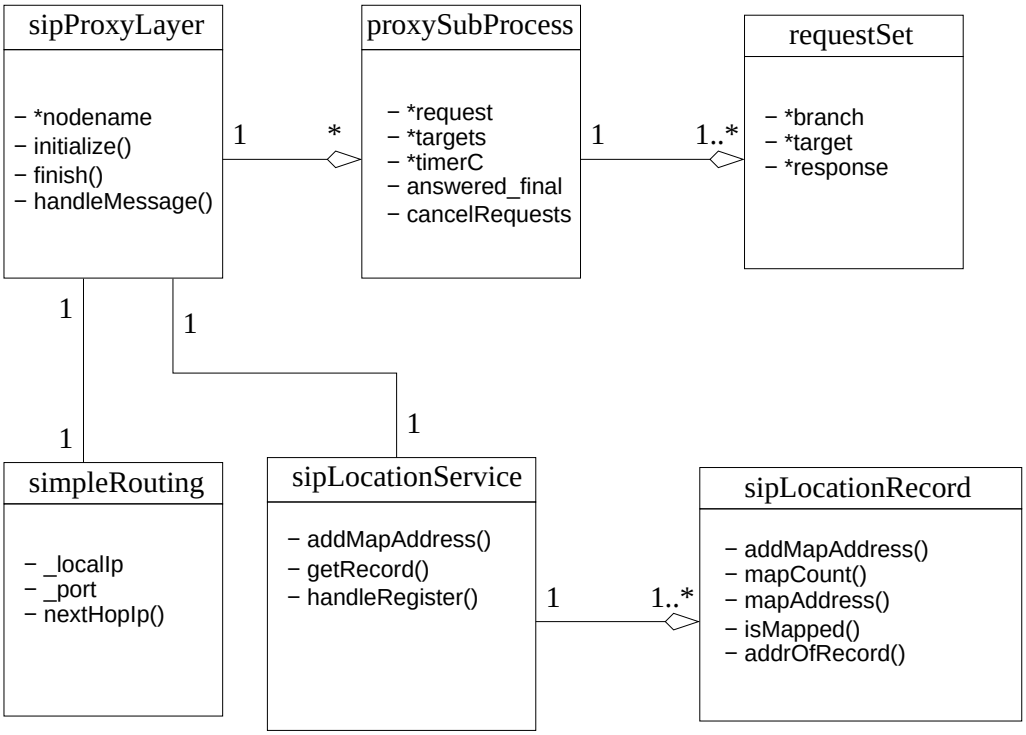

Abbildung 16: Module des sipProxyLayer

Der *sipProxyLayer* ist die Vaterklasse aller weiteren Objekte und auch die Klasse, die an die Stelle der *sipApplication* in der SIP-Anwendung, wie sie in Abbildung 9 auf Seite 9 dargestellt ist, tritt. Sie erledigt alle Aufgaben, die der Proxy zu erfüllen hat, da in den Unterklassen *proxySubProcess* und *requestSet* lediglich die Daten der potenziellen Unterprozesse abgelegt sind. Die *sipLocationService*-Klasse implementiert eine einfache lokale Funktion zur Verwaltung von Weiterleitungsadressen. Da die parallele Abarbeitung der Anfragen durch die diskrete, ereignisorientierte Simulationsumgebung gewährleistet wird sind die *proxySubProcesse* keine „echten“ Unterprozesse, sondern eine erweiterte Datenstruktur, die vom *sipProxyLayer* definiert wird.

Die Abarbeitung eingehender Nachrichten ist für das *sipProxyLayer*-Modul nach Eingangstor der jeweiligen Nachricht unterteilt, also nach Nachrichten, die sich das Modul selbst geschickt hat (selfMessage), Nachrichten, die vom *sipServerModule* empfangen wurden und die, die vom *sipClientModule* geschickt wurden.

6.4.1. Verarbeitung selbst erzeugter Nachrichten

Zu diesen Nachrichten gehört zum einen die Nachricht, die die Abarbeitung der initialen Funktionen anfordert und zum zweiten die Zeitgeber-Nachrichten („Timer C“).

6.4.2. Verarbeitung der Nachrichten vom Server-Modul

Empfängt die Proxy-Ebene eine Anfrage, kann sie sofort mit der Verarbeitung beginnen, da doppelt gesendete Anfragen schon von der Schicht aus *sipServerModule* und *sipServerTransaction* erkannt und verarbeitet werden. Ankommende Anfragen werden nach dem Schema aus Abschnitt 6.2 verarbeitet.

Bei der Ermittlung der Anfrageziele wird zunächst das ursprüngliche Ziel in die Zielliste aufgenommen. Da noch keine DNS-Implementierung existiert, können andere Anfrageziele zur Zeit nur über statische Listen ermittelt werden. Im einfachsten Fall handelt es sich also um genau ein Ziel. Kann kein Ziel ermittelt werden, wird die Verarbeitung mit eine Fehlerantwort „404 not found“ abgebrochen. Alle Antworten werden an die Adresse und den Port gesendet, die das *received*-Tag des obersten *Via*-Headerfeldes der Anfrage trägt. Wurde mindestens ein Ziel ermittelt, wird ein neuer Unterprozess erzeugt - was in unserem Fall nur eine Datenstruktur ist - dem die Anfrage und die Anfrageziele übergeben werden.

Für jedes ermittelte Anfrageziel wird die Adresse für den nächsten Sprung ermittelt. Mangels vorhandener Routing-Implementierungen wird diese Aufgabe vom *simpleRouting*-Modul, das auf Seite 48 beschrieben ist, übernommen. An diese Adresse wird eine modifizierte Kopie der Anfrage weiter geleitet. Das bedeutet, dass die modifizierten Anfragen an das *sipClientModule* gesendet werden, das für jede Anfrage eine *sipClientTransaction* startet, die die Anfrage versendet und Antworten entgegen nimmt.

6.4.3. Verarbeitung der Nachrichten vom Client-Modul

Bei Nachrichten des Client-Moduls handelt es sich um Antworten auf gesendete Anfragen. Da sie von einer *sipClientTransaction* vorverarbeitet wurden, ist garantiert, dass die Applikationsebene nur null bis mehrere provisorische gefolgt von genau einer finalen Antwort erreichen. Deren Verarbeitung geschieht nach dem Schema aus Abschnitt 6.3.

Zu jeder eingegangenen Antwort wird als erstes der zugehörige Unterprozess und das *requestSet* der Anfrage ermittelt. Ist der Unterprozess gefunden, werden die folgenden Punkte nacheinander abgearbeitet:

1. der Zeitgeber „C“ wird aktualisiert - also mit dem Startwert erneut gestartet.

2. der Statuscode der Antwort wird im *requestSet* gesichert.

3. ist die Antwort final, also der Statuscode größer oder gleich 200, wird sie selbst auch im *requestSet* gespeichert, da sie ein Favorit zur Weiterleitung ist.

4. ist der Statuscode im Bereich von 101 bis 299 wird die Antwort weiter geleitet.

5. es wird geprüft, ob schon eine finale Antwort weiter geleitet wurde, wenn nicht, wird weiterhin geprüft, ob alle Ziele des *requestSet* final beantwortet wurden. Ist das der Fall, wird die beste Antwort gewählt und weiter geleitet.

6. Ist der Statuscode der Antwort im „2xx“-Bereich wird an alle Ziele des *requestSet*, die provisorisch beantwortet sind, eine *CANCEL*-Anfrage geschickt, um die laufende Anfrage abzubrechen.

Ein Unterprozess (*proxySubProcess*) darf erst beendet und gelöscht werden, wenn alle von ihm ausgehenden Transaktionen beendet sind. Ist das nicht der Fall, so besteht die Möglichkeit, dass der Socketlayer ein verspätet eintreffendes Paket nicht mehr zuordnen kann und dadurch die gesamte Simulation abbricht.

6.4.4. Das sipLocationService-Modul

Dieses Modul implementiert einen einfachen lokalen Location Service. Das heißt, es speichert zu einer „Ausgangsadresse“ (addr_of_record) mehrere „Weiterleitungsadressen“ (mapAddress) in *sipLocationRecord*s. Gepflegt werden die Daten über die Zugriffsmethoden „handleRegister()“, „addMapAddress()“ und „getRecord()“. Letztere benötigt eine Adresse als Parameter und gibt den *sipLocationRecord* zurück, dessen „addr_of_record“ mit der übergebenen Adresse übereinstimmt. Aus dem LocationRecord können dann die Weiterleitungsadressen ausgelesen werden.

6.4.5. Das simpleRouting-Modul

Das *simpleRouting*-Modul ermittelt die IP-Adresse für den nächsten Sprung in Richtung der Ziel-Adresse auf die folgende Weise: Die ersten drei Teilfelder der IP-Adresse des Ziels werden mit denen der lokalen IP-Adresse des Routing-Moduls der Reihe nach verglichen. Sobald keine Übereinstimmung vorliegt, wird für dieses Teilfeld, das Teilfeld der Zieladresse übernommen und die folgenden Teilfelder auf „0“ gesetzt. Die vierte Stelle bekommt in diesem Fall die „1“ und somit die Adresse des nächst zuständigen Proxy. Im Falle, dass die ersten drei Stellen

übereinstimmen ist die Adresse für den nächsten Sprung die Zieladresse. Dieses System hat den offensichtlichen Nachteil, dass alle IP-Adressen, die auf eins enden zwangsläufig zu einem Proxy gehören.

Dieses Modul ist nötig, da es noch keine Implementierung des Routing-Services unter **OMNeT++** gibt. Es stellt eine stark vereinfachte Übergangslösung zur Ermittlung von Zieladressen und Zuständigkeitsbereichen dar und bedarf in Zukunft dringend Verbesserungen.

7. Simulation des SIP (jb)

In diesem Abschnitt wird ein kleines Simulationsnetzwerk vorgestellt. Es besteht aus den bis hier erwähnten Komponenten und zeigt deren Funktionen.

7.1. Testkonfiguration

Das Netzwerk umfasst, wie auch in Abbildung 17 zu erkennen ist, fünf SIP-Hosts, von denen drei als UACs, und zwei als UAS fungieren. Diese UACs und die UAS sind gruppenweise einem SIP-Proxyserver zugeordnet, der für die jeweilige Gruppe zuständig ist. Verbunden sind diese Elemente durch zwei Router, die aus der *IP-Suite* stammen.

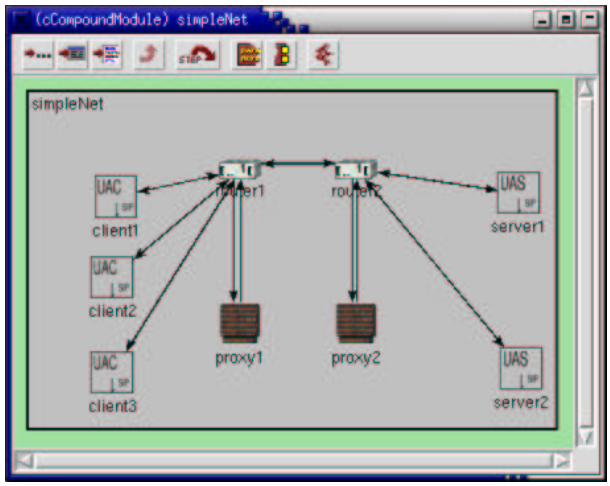

Abbildung 17: Simulationsmodell „simpleNet“

Die Konfiguration der einzelnen Elemente des Simulationsnetzwerkes ist in Tabelle 18 dargestellt. Zusätzlich wurde jedem UAC eine Kommandodatei zugewiesen. Die Kommandodateien enthalten Anweisungen, die die User Agents zu bestimmten Simulationszeiten ausführen sollen. Abbildung 18 auf Seite 51 zeigt als Beispiel einer Kommandodatei die vom „Server1“. In diesem Fall soll der UAS

eine REGISTER-Anfrage an „Proxy2“ schicken (10.0.10.1) um Benutzer Bert von „Server2“ auch auf „Server1“ zu registrieren.

NodeName	IP	User	200-Verz.
Client1	10.0.0.10	Anna	2 Sek
Client2	10.0.0.20	Berta	2 Sek
Client3	10.0.0.30	Christine	2 Sek
Proxy1	10.0.0.1	—	—
Proxy2	10.0.10.1	—	—
Server1	10.0.10.40	Anton	2 Sek
Server2	10.0.10.50	Bert	30 Sek

Tabelle 18: Konfiguration der Elemente

7.2. Testlauf

Im folgenden wird die chronologische Abfolge der SIP-Aktionen der Beispielsimulation vorgestellt:

1. Zum Zeitpunkt $t = 2$ Sekunden schickt „Server1“ eine REGISTER-Anfrage an „Proxy2“ um Benutzer „Bert“, der sich sonst an „Server2“ aufhält, lokal zu registrieren.

2. „Proxy2“ bestätigt die Registrierung.

3. Zum Zeitpunkt $t = 10$ schickt „Client1“ eine INVITE-Anfrage an „Clemens@Server2“.

4. „Server2“ antwortet mit dem Fehlercode „404 - Not Found“, da der Benutzer „Clemens“ dort nicht bekannt ist.

5. Zum selben Zeitpunkt schickt „Client2“ eine INVITE-Anfrage an „Bert@Server2“.

6. „Proxy2“ leitet die Anfrage an „Server1“ und „Server2“ weiter, da sich „Bert@Server2“ auch für „Server1“ registriert hat.

7. Da „Server1“ nach 2 Sekunden mit einer „200 - OK“ Antwort auf die Einladung antwortet, bricht „Proxy2“ den Einladungsversuch an „Server2“ mit einer CANCEL-Anfrage ab und leitet die 200er-Antwort in Richtung „Client2“ weiter.

8. „Client2“ und „Server1“ etablieren eine RTP-Sitzung.

9. Zum Zeitpunkt $t = 17$ schickt „Client2“ eine BYE-Anfrage an „Server1“ um die RTP-Sitzung zu beenden.

10. Auf die BYE-Anfrage hin antwortet „Server1“ mit „200 - OK“ und beendet die RTP-Sitzung, wie auch „Client2“ beim Empfang der Nachricht.

11. Zum Zeitpunkt $t = 29$ registriert sich „Clemens“ lokal auf „Server2“.

12. Bei $t = 30$ bringt „Client3“ zwei INVITE-Anfragen auf den Weg. Eine an „Bert@Server1“ und die zweite an „Clemens@Server2“.

13. Da der Abbau der RTP-Sitzung auf „Server1“ noch nicht abgeschlossen - die RTP-Sitzung somit noch belegt ist, antwortet „Server1“ mit „481 - Temporarily Unavailable“.

14. „Server2“ akzeptiert die Anfrage mit einer 200er Antwort und es wird wieder eine RTP-Sitzung aufgebaut.

15. Zum Zeitpunkt $t = 63$ beendet „Client3“ die RTP-Sitzung mit einer BYE-Anfrage.

```
#
# SIP commandfile (Anton)
#
# map Bert@10.0.10.50 to this Host - use Proxy 10.0.10.1
2s REGISTER Bert 10.0.10.50 10.0.10.1
```

Abbildung 18: Beispiel einer Kommandodatei

7.3. Testergebnis - die Ausgabedatei

Die Ausgabe der Simulationsergebnisse wird mit Hilfe der **OMNeT++** -eigenen *cOutVector*-Klasse geschrieben. Alle Instanzen dieser Klasse schreiben ihre Daten in die selbe Textdatei. Für dieses Modell ist es die Datei „omnetpp.vec“. Ein Beispiel für eine Ausgabedatei findet sich in der Abbildung 19 [11].

An diesem Beispiel kann man den prinzipiellen Aufbau der Ausgabedatei gut erkennen. Jede Instanz des *cOutVector* erzeugt, wenn sie das erste mal in die Ausgabedatei schreibt, eine Zeile, die den Vektor selbst beschreibt. Diese Zeile beinhaltet die Vektorennummer, unter der die Daten des Vektors später eingetragen werden. Im Beispiel also „vector 1“, bzw. „vector 2“. In der nächsten Spalte wird die Simulationszeit der Aufzeichnung als Fließkommazahl festgehalten. Die

[11] vgl. [Varb], Abschnitt 10.3

```
mysim.vec:
vector 1   "subnet[4].term[12]"  "response time"  1
1  12.895  2355.66666666
1  14.126  4577.66664666
vector 2   "subnet[4].srvr"  "queue length"  1
2  16.960  2.00000000000.63663666
1  23.086  2355.66666666
2  24.026  8.00000000000.44766536
```

Abbildung 19: Beispiel einer Ausgabedatei

letzte Spalte enthält den Wert, den man der *cOutVector*-Klasse übergeben hat. Einziger Nachteil dieser Klasse ist, dass ausschließlich Fließkommawerte verarbeitet werden. Aus diesem Grunde bedarf es noch einer Interpretation dieser Werte. Diese lautet wie folgt:

0-49 Werte in diesem Bereich verschlüsseln SIP-Anfragemethoden. Die genauen Werte sind die den Methoden zugeordneten ganzzahligen Werte aus der „sipConstants.h"-Datei der SIP-Implementierung.[12]

50-99 Werte in diesem Bereich verschlüsseln die Typen des *sipInterfacePacket*. Dies geschieht auf die gleiche Weise, wie bei den SIP-Anfragemethoden, mit der Ausnahme, dass die Werte mit 50 addiert werden, um einen Konflikt mit den Werten der Anfragemethoden zu verhindern.[13]

100-699 Diese letzte Gruppe stellt SIP-Antworten durch ihren Statuscode dar.

Anhand dieser Werte kann nachvollzogen werden, welche Nachrichten von welchem SIP-Element gesendet oder empfangen wurden. Außerdem ist es mit einfachen Mitteln, wie z.B. *gnuplot*[14] oder von **OMNeT++** gelieferten Tools, wie *splitvec* oder *plove*, leicht möglich, die Ausgabedatei weiter auszuwerten. So erzeugt *splitvec* aus der Ausgabedatei für jeden enthaltenen Vektor eine eingene Datei, die nur dessen Daten enthält. Da keine Spezifikation bezüglich der Simulationsausgabe vorliegt, wurde die Ausgabe auf die hier genannten Punkte (Zeitpunkte des Sendens und Empfanges von Nachrichten und deren Typen) beschränkt. Es ist allerdings mit wenig Aufwand möglich, die Ausgaben zu verändern oder zu erweitern um spezifischere Aussagen über das Protokollverhalten zu generieren.

[12] siehe Anhang Seite 61
[13] siehe auch Anhang Seite 62
[14] `http://www.gnuplot.info/`

7.4. Die Kommandodatei

Jedem SIP-Host muss eine Kommandodatei (cmdFile) zugeordnet sein. Im einfachsten Fall ist sie leer, was bedeutet, dass der SIP-Host keine selbstständigen Aktionen ausführen soll. Ansonsten hat die Kommandodatei folgenden Syntax:

1. Die Kommandodatei wird zeilenorientiert verarbeitet. D.h. in genau einer Zeile kann genau ein Befehl stehen.

2. Kommentare beginnen mit einem Rautezeichen (#). Alle folgenden Zeichen einer Zeile werden ignoriert.

3. Ein Kommando besteht aus den folgenden fünf Argumenten (Spalten):

   a) Zeit: der Zeitpunkt an dem die Anfrage gestellt werden soll.

   b) Methode: der Methodenname der zu stellenden Anfrage.

   c) Benutzername

   d) IP-Adresse

   e) weiteres, optionales Argument - z.B. für eine *REGISTER*-Anfrage.

4. Die einzelnen Argumente (Spalten) sind durch Leerzeichen (Space) oder Tabulatoren ('\t') zu trennen.

8. Zusammenfassung und Ausblick

SIP ist ein Protokoll zum Aufbau von Multimediasitzungen. Die Funktionalität von SIP kann nur in Verbindung mit weiteren Protokollen vollständig genutzt werden. Das *Session Description Protocol (SDP)*[15] dient der Beschreibung von Sitzungen. Durch SDP können z.B. Sitzungen eindeutig identifiziert werden. Eine Anbindung an SDP wurde nicht vorgenommen, da SDP für **OMNeT++** noch nicht implementiert ist. Schnittstellen zur Netzwerkschicht bieten UDP und TCP. In der vorgestellten Implementierung wurde UDP als Netzwerkprotokoll verwendet, da eine Implementierung für TCP bezüglich **OMNeT++** zur Zeit der Erstellung dieser Arbeit noch nicht existiert hat.
Die Grundfunktionalität des SIP wurde implementiert und soweit möglich getestet. Dazu gehört:

- der Aufbau von Dialogen,
- das Beenden von schwebenden und etablierten Dialogen,
- das Senden von Nachrichten über Proxys und
- das Registrieren eines Benutzers lokal an einem Proxy.

Erweiterungen der Implementierung im Zusammenhang mit weitergehenden Arbeiten können die folgenden Punkte sein:

- Anbindung des SIP an SDP.
- Vollständige Implementierung des Proxys bezüglich der Modi.
- Erweiterung der Funktionalität der Methode REGISTER. Das bedeutet, eine Registrierung kann von mehreren Proxys verwendet werden.
- Erweiterung der Implementierung um DNS, wenn DNS für **OMNeT++** implementiert wurde.

[15] siehe [HJ98]

9. Glossar

Client *Host* in einem Netzwerk, der die Dienste von *Server*n in Anspruch nimmt.

DNS „Domain Name System“ - ein Internetdienst, der *Domain*-Namen in *IP-Adressen* und umgekehrt auflöst.

Domain Domains fassen Organisationseinheiten im Internet zusammen. Z.B. *tu-braunschweig.de*, für die komplette Internetpräsenz der TU-Braunschweig.

Header englische Bezeichnung des Kopfs einer Nachricht. Er enthält die für die Zustellung der Nachricht wichtigen Informationen.

Host Ein Computer, auf dem Programme (Dienste) ausgeführt werden können.

IP-Adresse Adresse, über die jedes Element im Internet identifiziert und angesprochen werden kann. Sie besteht aus vier durch Punkte getrennte Zahlen im Bereich von Null bis 255.

Location Service Funktion eines *Host*, die es ermöglicht, den Aufenthaltsort (z.B. einen *User Agent*) eines Benutzers zu ermitteln.

Port ein logischer Verbindungspunkt eines Computers, über den bestimmte Programme erreicht werden können.

Proxy ein *Server* in einem Netzwerk, der Weiterleitungs- und Zwischenspeicherungsdienste anbietet.

Request englische Bezeichnung für „Anfrage“.

Response englische Bezeichnung für „Antwort“.

Routing Der Prozess der Wahl des Weges in einem Netzwerk zur Erreichung des Bestimmungspunktes.

Server *Host* in einem Netzwerk, der anderen *Host*s Dienste zur Verfügung stellt.

Timer ein Ereignis, das nach einer bestimmten Zeit ausgelöst wird, wenn es initialisiert wurde. Mit Zeitgebern können so Wartezeiten programmiert und simuliert werden.

TU *Transaction User*. Bezeichnet das *sipApplication*-Modul im Zusammenhang mit Transaktionen.

UAC ein *User Agent*, der als *Client* fungiert.

UAS ein *User Agent*, der als *Server* fungiert.

User Agent ein *Host*, der mit Anwendern interagiert.

Zeitgeber Siehe *Timer*.

A. Wichtige Status-Codes

Im Folgenden werden wichtige Status-Codes aufgeführt.

100 Trying wird von der *serverTransaction* generiert

180 Ringing signalisiert das Klingeln beim Benutzer

183 Session Progress signalisiert, dass die Anfrage bearbeitet wird

200 OK die Einladung wurde angenommen

300 Multiple Choices der Adressat besitzt mehrere Adressen

301 Moved Permanently der Adressat ist nicht mehr unter der angegebenen Andresse erreichbar

302 Moved Temporarily der Adressat ist temporär nicht unter der angegebenen Adresse erreichbar

305 Use Proxy es soll ein Proxy benutzt werden

380 Alternative Service die Anfrage wurde nicht erfolgreich bearbeitet, aber andere Services sind verwendbar

401 Unauthorized für die Anfrage wird eine Authentifizierung benötigt

403 Forbidden die Anfrage darf nicht an den adressierten Server gestellt werden

404 Not Found der adressierte Benutzer existiert nicht bei dem adressierten Server

405 Method Not Allowed die geforderte Methode ist nicht erlaubt

407 Proxy Authentication Required für die Anfrage wird eine Proxy-Authentifizierung benötigt

413 Request Entity Too Large der Nachrichteninhalt der Anfrage ist zu lang

415 Unsupported Media Type der geforderte Typ wird nicht unterstützt

416 Unsupported URI Scheme das verwendete URI Schema wurde nicht verstanden

420 Bad Extension die geforderten Erweiterungen wurden nicht verstanden

486 Busy Here die Anfrage kann zur Zeit nicht beantwortet werden

B. Kommandodateien der Testsimulation

Die Kommandodateien der Simulationskonfiguration.

```
#
# SIP commandfile (Anna)
#
10s INVITE Clemens 10.0.10.50
120s INVITE Niemand 10.0.0.20
```

Abbildung 20: Kommandodatei „Client1“

```
#
# SIP commandfile (Berta)
#
10s INVITE Bert 10.0.10.50
17s BYE Bert 10.0.10.50
```

Abbildung 21: Kommandodatei „Client2“

```
#
# SIP commandfile (Christine)
#
30s INVITE Anton 10.0.10.40
30s INVITE Clemens 10.0.10.50
60s BYE X 10.0.10.40
63s BYE X 10.0.10.50
```

Abbildung 22: Kommandodatei „Client3“

```
#
# SIP commandfile (Anton)
#
# map Bert@10.0.10.50 to this Host - use Proxy 10.0.10.1
2s REGISTER Bert 10.0.10.50 10.0.10.1
```

Abbildung 23: Kommandodatei „Server1“

```
#
# SIP commandfile (Bert)
#
# map Clemens@10.0.10.50 to this Host - use Proxy 10.0.10.1
29s REGISTER Clemens 10.0.10.50 10.0.10.1
```

Abbildung 24: Kommandodatei „Server2“

C. Ausgabe der Simulation

Die durch die Simulation erzeugte Ausgabedatei „omnetpp.vec“ wurde mittels des OMNeT++-Skripts „splitvec“ so aufgeteilt, dass jeder Ergebnisvektor in einer eigenen Datei steht. Im Folgenen sind die Ergenbnisvektoren der beiden Proxyserver vorgestellt.

```
# vector 3  "simpleNet.proxy1.sipApplication"  "sipProxy-1st Proxy-incoming"  1
10.0000034 1
10.0000046 1
10.000019 100
10.0000218 100
10.000037 180
10.0000396 404
10.0000408 180
12.0000298 200
30.0000045 1
30.0000057 1
30.0000201 100
30.0000223 100
30.0000365 486
30.0000377 180
60.0000281 200
120.000003 1
120.000012 100
120.000013 404
```

Abbildung 25: Eingangsvector „Proxy1“

```
# vector 0  "simpleNet.proxy1.sipApplication"  "sipProxy-1st Proxy-outgoing"  1
0 54
10.0000034 1
10.0000046 1
10.000037 180
10.0000396 404
10.0000408 180
12.0000298 200
30.0000045 1
30.0000057 1
30.0000365 486
30.0000377 180
60.0000281 200
120.000003 1
120.000013 404
```

Abbildung 26: Ausgangsvector „Proxy1“

```
# vector 2  "simpleNet.proxy2.sipApplication"  "sipProxy-2nd Proxy-incoming"  1
2.0000029 6
10.000012 1
10.0000132 1
10.000025 100
10.0000264 100
10.0000276 180
10.0000288 404
10.0000312 100
10.0000336 180
12.0000248 200
12.0000339 487
12.0000351 200
30.0000131 1
30.0000143 1
30.0000257 100
30.0000269 486
30.0000281 100
30.0000305 180
60.0000247 200
```

Abbildung 27: Eingangsvector „Proxy2“

```
# vector 1  "simpleNet.proxy2.sipApplication"  "sipProxy-2nd Proxy-outgoing"  1
0 54
2.0000029 200
10.000012 1
10.0000132 1
10.0000132 1
10.0000276 180
10.0000288 404
10.0000336 180
12.0000248 200
12.0000248 5
30.0000131 1
30.0000143 1
30.0000269 486
30.0000305 180
60.0000247 200
```

Abbildung 28: Ausgangsvector „Proxy2“

D. Referenzierte Auszüge des Programmcodes

Dieser Abschnitt enthält Auszüge des Programmcodes, auf die in dieser Ausarbeitung direkt verwiesen wurde.

D.1. sipConstants.h

```
//! SIP request methods
enum sipRequestMethod {
UNDEF    = 0,
INVITE   = 1,
ACK      = 2,
OPTIONS  = 3,
BYE      = 4,
CANCEL   = 5,
REGISTER = 6
};
//! strings to the sipRequestMethods
static char *const sipRequestMethodStr[] = {
"UNDEFINED",
"INVITE",
"ACK",
"OPTIONS",
"BYE",
"CANCEL",
"REGISTER"
};
```

Abbildung 29: sipConstants.h, Zeile 62 bis 81

D.2. sipInterfacePacket.h

```
/*!
enumeration of defined sip-actions
*/
enum sipAction {
// SA = SipAction
SA_UNDEF,
SA_REQUEST,
SA_RESPONSE,
SA_TERMINATE,
SA_OPEN_PORT,
SA_TIMEOUT_REQUEST,
SA_TIMEOUT_RESPONSE,
SA_FORWARD,
SA_CLOSE_SESSION,
SA_FROM_SOCKET,
SA_ACK
};
```

Abbildung 30: sipInterfacePacket.h, Zeile 33 bis 46

Literatur

[H+99] Handley et al. SIP: Session Initiation Protocol. RFC 2543, Network Working Group, Mar 1999. http://www.ietf.org/rfc/rfc2543.txt; eingesehen am 2. Januar 2003.

[HJ98] M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327, Network Working Group, Apr 1998. http://www.ietf.org/rfc/rfc2327.txt; eingesehen am 2. Januar 2003.

[Opp02] Matthias Oppitz. Entwurf und Implementierung einer Simulation des Realtime Transport Protocol unter OMNeT++. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Mar 2002. http://cvs-int.etec.uni-karlsruhe.de/omnetpp/model-doc/rtp.html; eingesehen am 5. August 2003.

[R+02] Rosenberg et al. SIP: Session Initiation Protocol. RFC 3261, Network Working Group, Jun 2002. http://www.ietf.org/rfc/rfc3261.txt; eingesehen am 2. Januar 2003.

[S+96] H. Schulzrinne et al. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, Network Working Group, Jan 1996. http://www.ietf.org/rfc/rfc1889.txt?number=1889; eingesehen am 2. Januar 2003.

[Vara] Andreas Varga. *OMNeT++ Community Site.* http://www.omnetpp.org/; eingesehen am 5. August 2003.

[Varb] Andreas Varga. *User Manual - OMNeT++ version 2.3.* http://www.omnetpp.org/external/doc/html/usman.php; eingesehen am 5. August 2003.

[Zhu01] Mei Zhu. Conception, Implementation and Evaluation of a Socket Interface Layer under OMNeT++. Diploma Thesis, Universität Karlsruhe - Institute für Nachrichtentechnik, Dec 2001.