Fachhochschule Ansbach - University of Applied Sciences

# Diplomarbeit

Thema: **VoIP-Konzept für SSL-geschützte Internettelefonie**
Studiengang: Wirtschaftsinformatik
Referent: Prof. Dr. Dieter Schön
Koreferent: Dipl. Ing. (FH) Jürgen-Lorenz Lauenstein

HOB GmbH & Co. KG

vorgelegt von
**Oguzhan Alan**

Oguzhan Alan
Matrikelnummer: 24344
Siegelsdorfer Str. 14
90768 Fürth

# Inhaltsverzeichnis

**Abbildungsverzeichnis** **III**

**Tabellenverzeichnis** **IV**

**Listings** **V**

**1 Einleitung** **1**

**2 Technologischer Hintergrund** **4**
2.1 Protokolle . . . . . 4
2.1.1 Session Initiation Protocol (SIP) . . . . . 4
2.1.1.1 SIP-Komponenten . . . . . 5
2.1.1.2 SIP- Notation . . . . . 7
2.1.1.3 SIP- Verbindungsaufbau . . . . . 9
2.1.2 Session Description Protocol (SDP) . . . . . 13
2.1.3 Real-Time Transport Protocol (RTP) . . . . . 16
2.1.4 Secure Sockets Layer (SSL) . . . . . 17
2.2 Die Telefonanlage Asterisk . . . . . 22
2.3 HOB WebSecureProxy (WSP) . . . . . 25

**3 Realisierung** **28**
3.1 Architektur des Konzeptes . . . . . 28
3.1.1 Definition der Komponenten . . . . . 28
3.1.2 Channel-Mechanismus . . . . . 34
3.1.3 WSP-Protokoll . . . . . 42
3.1.4 Numbers in HOB Abstract Syntax Notation (NHASN) . . . . . 46
3.2 Implementierung eines SIP-Client . . . . . 49
3.2.1 HOBPhone Architektur . . . . . 49
3.2.2 Übertragung von Signalisierungs- und Sprachdaten . . . . . 52
3.2.2.1 UDP-Modus . . . . . 52
3.2.2.2 SSL-Modus . . . . . 56
3.2.3 Verarbeitung von Signalisierungsdaten . . . . . 64
3.2.3.1 Verarbeitung von Signalisierungsdaten (SIP,SDP) . . . . 64
3.2.3.2 Verarbeitung von WSP-Signalisierungsdaten . . . . . 78

3.2.4 Verarbeitung von Sprachdaten (RTP) . . . . . 84
3.2.4.1 Steuerung der Audio-Komponenten . . . . . 84
3.2.4.2 Verarbeitung der Sprachdatenpakete . . . . . 89
3.2.5 Grafische Benutzeroberfläche . . . . . 91
3.3 Implementierung eines SDH . . . . . 97
3.3.1 Architektur des SDH . . . . . 97
3.3.1.1 Aufbau des SDH . . . . . 97
3.3.1.2 Gather-Strukturen . . . . . 101
3.3.2 Verarbeitung von Datenpaketen im SDH . . . . . 102

**4 Fazit und Ausblick** **113**

**Literaturverzeichnis** **VI**

**A Screenshots** **VII**

**B Quellcode** **IX**
B.1 xltclib9.cpp . . . . . IX
B.2 WSPConnection.java . . . . . XXXVII
B.3 PhoneManager.java . . . . . XLVI
B.4 SIPDialog.java . . . . . LV

# Abbildungsverzeichnis


2.1 SIP- Server Architektur . . . . . 7
2.2 Verbindungsaufbau mit SIP . . . . . 10
2.3 RTP- Header . . . . . 16
2.4 SSL-Verbindungsaufbau . . . . . 20
2.5 VoIP-Architektur mit Asterisk . . . . . 22
2.6 Web-Portal HOB RD VPN . . . . . 25
2.7 HOB WSP / Server-Data-Hook Konzept . . . . . 26

3.1 VoIP-Architektur mit WSP . . . . . 30
3.2 Anmeldedialog WSP mit VoIP . . . . . 31
3.3 Informationsrichtungen von Daten . . . . . 33
3.4 SIP-Datenverkehr mit WSP . . . . . 33
3.5 Channel-Mechanismus . . . . . 37
3.6 Fehlinformationen im SIP-Header . . . . . 37
3.7 Aufruf eines SDH . . . . . 39
3.8 Numbers in HOB Abstract Syntax Notation für den Wert 486 . . . . . 47
3.9 Klassendiagramm hob.phone . . . . . 51
3.10 Daten im SSL-Modus . . . . . 60
3.11 Aktivitätsdiagramm 'SIPManager.java -sipReceived()' . . . . . 69
3.12 Aktivitätsdiagramm 'newSIPDialog()' . . . . . 88
3.13 Einstiegspunkt in MainApplet.java im Aktivitätsdiagramm . . . . . 93
3.14 Grafische Benutzeroberfläche 'Standard' . . . . . 96
3.15 Grafische Benutzeroberfläche 'Basic' . . . . . 97
3.16 Informationsrichtung von Datenpaketen . . . . . 102
3.17 Darstellung einer Datensequenz mit Gather-Strukturen . . . . . 110
3.18 Datenfragmentierung im SSL-Modus . . . . . 111
3.19 Verarbeitung fragmentierter Daten im SDH . . . . . 112

4.1 Grafische Benutzeroberfläche 'Futuristic' . . . . . 114

A.1 Wireshark Capture: SSL-Verbindungsaufbau . . . . . VII
A.2 Entwicklungsumgebung eclipse, Package Explorer . . . . . VIII

# Tabellenverzeichnis

2.1 SDP- Parameter . . . . . 14
2.2 Payload- Types . . . . . 15
2.3 Nachrichten beim SSL-Verbindungsaufbau . . . . . 19
2.4 'ClientHello' Parameter im SSL-Verbindungsaufbau . . . . . 21

3.1 Control-Request im WSP-Protokoll . . . . . 44

# Listings


2.1 INVITE- Message . . . 9
2.2 Proxy-Authorization . . . 11
2.3 100 Trying . . . 11
2.4 200 OK . . . 12
2.5 BYE Message . . . 12
2.6 SDP- Content . . . 16

3.1 WSPConf.xml . . . 34
3.2 WSPConf.xml -Mulitple entries . . . 36
3.3 WSP-Dialog CREATE . . . 44
3.4 WSP-Dialog SET . . . 45
3.5 Interface SIPConnection.java . . . 49
3.6 SIPManager.java -sendResponse . . . 50
3.7 SIPConnectionUDP.java . . . 53
3.8 RTPConnectionUDP.java . . . 53
3.9 UDPConnection.java -send . . . 53
3.10 SIPConnectionUDP.java -sendSIP . . . 54
3.11 UDPConnection.java -receivedData . . . 54
3.12 SIPConnectionUDP.java -receivedData . . . 54
3.13 RTPConnectionUDP.java -receivedData . . . 54
3.14 PhoneManager.java -hmp_SIPConnections . . . 55
3.15 SIPClient.java -setClient . . . 55
3.16 PhoneManager.java -new SIPManager(sip_con) . . . 56
3.17 WSPConnection.java -Input/Output-Stream . . . 57
3.18 WSPConnectionListener.java . . . 57
3.19 SIPConnectionWSP.java . . . 57
3.20 RTPConnectionWSP.java . . . 58
3.21 WSPManager.java . . . 58
3.22 WSPConnection.java -addListener . . . 59
3.23 WSPConnection.java -loop to receive data . . . 59
3.24 WSPConnection.java -loop over input received . . . 59
3.25 WSPConnection.java -receive channel . . . 61
3.26 WSPConnection.java -receive NHASN . . . 61
3.27 WSPConnection.java -receive packet . . . 62

3.28 WSPConnection.java -single packet received complete . . . . . 63
3.29 WSPConnection.java -send message . . . . . 64
3.30 SIPMessage.java -Konstruktor . . . . . 64
3.31 SIPHeader.java . . . . . 65
3.32 SIPManager.java -SIPManager.java -sipReceived(SIPMessage msg_in) . 67
3.33 SIPManager.java . . . . . 68
3.34 SIPClient.java -receivedNewSIPRequest . . . . . 70
3.35 SIPClient.java -userAction . . . . . 72
3.36 SIPDialog.java -setNewDialog . . . . . 72
3.37 SIPManager.java -sendRequest . . . . . 73
3.38 SIPManager.java -case INVITE . . . . . 74
3.39 SIPDialog.java -case 407 Proxy Authentication Required . . . . . 75
3.40 SIPRequest INVITE -SIPRespone 407 . . . . . 75
3.41 SIPDialog.java -sendInvite . . . . . 76
3.42 Konsolenausgabe INVITE-Message . . . . . 77
3.43 WSPManager.java -receivedData . . . . . 78
3.44 WSPManager.java -controlReceived . . . . . 79
3.45 WSPManager.java -registryReceived . . . . . 80
3.46 PhoneUserInterface.java -btnSubmitPressed . . . . . 81
3.47 RTPConnectionWSP.java -setRTPConnection . . . . . 82
3.48 WSPManager.java -createChannel . . . . . 82
3.49 WSPManager.java -setChannel . . . . . 83
3.50 AudioManager.java -getPlayer . . . . . 85
3.51 AudioManager.java -getRecorder . . . . . 85
3.52 SDP-Content . . . . . 86
3.53 RTPSender.java . . . . . 89
3.54 RTPManager.java -playPacket . . . . . 90
3.55 MainApplet.java . . . . . 91
3.56 PhoneManager.java . . . . . 92
3.57 PhoneUserInterface.java . . . . . 93
3.58 PhoneUserInterface.java -ua_register . . . . . 94
3.59 PhoneStatusEvent.java . . . . . 94
3.60 PhoneUserInterface.java -setPhoneStatus . . . . . 95
3.61 PhoneUserInterface.java -isActiveClient . . . . . 95
3.62 PhoneUserInterface.java -newPhoneStatusEvent . . . . . 95
3.63 xltclib9.cpp . . . . . 98
3.64 WSPConf.xml -library file name . . . . . 98
3.65 hob-xsclib01.h . . . . . 98
3.66 xltclib9.cpp -switch (adsp_hl_clib_1->inc_func) . . . . . 99
3.67 xltclib9.cpp -struct dsd_clib1_session_9_1 . . . . . 100
3.68 hob-xsclib01.h -struct dsd_gather_i_1 . . . . . 101
3.69 xltclib9.cpp -DEF_AUX_GET_IDENT_SETTINGS . . . . . 103

3.70 xltclib9.cpp -m_extend_gather . . . . . 104
3.71 xltclib9.cpp -REGISTER . . . . . 105
3.72 xltclib9.cpp -m_send_channel . . . . . 106
3.73 xltclib9.cpp -SIP received . . . . . 107
3.74 xltclib9.cpp -m_redirect_udp . . . . . 108
3.75 xltclib9.cpp -loop over all buffers received . . . . . 108

B.1 xltclib9.cpp . . . . . IX
B.2 WSPConnection.java . . . . . XXXVII
B.3 PhoneManager.java . . . . . XLVI
B.4 SIPDialog.java . . . . . LV

# Kapitel 1

# Einleitung

Voice over IP (kurz VoIP) gewinnt mit zunehmender Digitalisierung von Informationen im allgemeinen Sinne schon aus Kompatibilitätsgründen immer mehr an Bedeutung. Das Vorhaben, Computernetzwerke zur Übertragung der menschlichen Sprache als digitalisierten Informationsträger zu nutzen, hat seinen Ursprung bereits in den 1990er Jahren. Die israelische Firma VocalTec stellte hierzu 1995 eine Internet-Phone-Software vor, die lediglich im Halb-Duplex Betrieb von PC zu PC genutzt werden konnte. Nicht zuletzt die Sprachqualität galt jedoch bei VoIP lange Zeit als zu schlecht, vor allem in der Unternehmenskommunikation. Unterschiedliche Systemarchitekturen, keine einheitlichen Protokolle bzw. technische Standards, fehlender Support und Erfahrung, Echos, Verzögerungen und Ausfälle erschwerten den verlässlichen Einsatz von VoIP besonders in der Anfangszeit.

Die erste Norm für multimediale Kommunikation über Paket basierte Netze ohne Quality of Service (kurz QoS) kam 1996 mit dem Standard H.323 der International Telecommunication Union (ITU), welches auf 50 weitere Standards verweist. Zusammen mit dieser Norm wurden erstmals VoIP-Gateways entwickelt, was für VoIP einen entscheidenden Schritt bedeutete. VoIP-Gateways ermöglichten somit die Verbindung von Datennetzen mit dem öffentlichen Telefonnetz, dem Public Switched Telephony Network (kurz PSTN). Mittels einer Schnittstelle zum Telefonnetz, einem so genannten Telefonnetz-Interface, konnten über VoIP-Gateways eingehende Sprachsignale aus dem Telefonnetz ins Datennetz und umgekehrt weitergeleitet werden. Somit ergab sich für VoIP die Möglichkeit den Einsatz von PC zu PC, auf PC zu Telefon und umgekehrt und sogar auf Telefon zu Telefon über VoIP auszuweiten. Ziel war es zunächst, die durch das Telefonnetz entstehenden Kosten vor allem für internationale Gespräche durch die Umleitung auf das Datennetz zu reduzieren.

Die Deregulierung des Telekommunikationsmarktes ab dem 1. Januar 1998 brachte jedoch enorme Preissenkungen in der traditionellen Telefonie mit sich, wobei abgesehen davon die Verfügbarkeit und Qualität von VoIP noch nicht das gleiche Niveau einer klassischen Telefonanlage erreicht hatte. Zu dieser Übergangszeit wurde VoIP in

Unternehmen überwiegend intern, also im Intranet eingesetzt, wobei extern die Telekommunikation weiterhin über das traditionelle Telefonnetz mittels ISDN erfolgte. Mit dieser hybriden Kommunikationsarchitektur beabsichtige man vor allem lokale Netze für VoIP-Anforderungen hinsichtlich Netzwerkkapazitäten vorzubereiten. Bei der Ausweitung der lokalen Kapazitäten spielte allerdings die durch VoIP zu erwartende Kostenersparnis angesichts der erforderlichen Maßnahmen im Vorfeld und der Liberalisierung des Telekommunikationsmarktes nicht mehr die Hauptrolle. Vielmehr wurden Möglichkeiten und Szenarien im VoIP-Umfeld erörtert, die in ihrer Anwendung einen Mehrwert für das Unternehmen schaffen sollten, was zur Definition von neuen Anwendungen und Features führte. Gleichzeitig verzeichnete jedoch der Internet-Zugangsmarkt einen stetigen Wachstum an DSL-Anschlüssen mit zunehmenden Datenverkehrsvolumen aber auch Bandbreiten für Up- und Downstream. Zusammen mit aggressiven Preisstrategien auf dem Markt für DSL-Anbieter und nachfragegerechtem Angebot, entwickelte sich das Verhältnis von Bandbreite, Datenvolumen und Kosten unter anderem auch im Bezug auf die Zielsetzung von Kostenersparnis durch Umleitung von Sprachdaten auf das Datennetz aber auch im Bezug auf Produktivität und Mehrwert, letztendlich immer mehr zugunsten von VoIP-Lösungen.

Inzwischen wird der im Jahre 1999 von der Internet Engineering Task Force (kurz IETF) entwickelte Standard Session Initiation Protocol (kurz SIP) wegen seiner Flexibilität, Transparenz und Einfachheit nach der im Jahre 2002 veröffentlichten zweiten Version von allen deutschen VoIP-Providern eingesetzt und gilt als Durchbruch im VoIP-Umfeld. Zwölf Jahre nach der Vorstellung der Internet-Phone-Software von VocalTec, verkündete 2007 der DSL-Anbieter 1&1 seit dem Start ihres ersten VoIP- Angebotes innerhalb von zwei Jahren mehr als zehn Milliarden Gesprächsminuten über das Internet vermittelt zu haben und prophezeite das Ende der Festnetztelefonie. Vor allem auch immer mehr Unternehmen nutzen inzwischen die neue Form der Kommunikation, laut einer Studie des Wirtschaftsprüfungsunternehmens Deloitte (http://www.deloitte.com) bereits 14% der deutschen Unternehmen, wobei abgesehen von betriebswirtschaftlichen Aspekten die Sprachqualität nach wie vor ein sehr hohes Maß an Bedeutung beinhaltet. Entsprechend den Anforderungen der Unternehmen an Verfügbarkeit, Qualität, Service und Support, der ständig wachsenden Nachfrage und den hinsichtlich VoIP inzwischen ausgereiften technischen Begebenheiten, haben auch Anbieter und Varianten von VoIP-Endgeräten, Software und Telekommunikationsanlagen stark zugenommen.

Durch die technischen Entwicklungen, welche inzwischen den Anforderungen der Internettelefonie gerecht werden und dem ausgereiften Markt für VoIP-Anbieter, können Unternehmen schließlich auch den erwünschten Mehrwert durch den Einsatz von VoIP realisieren. Somit lassen sich oft Arbeitsprozesse optimieren, Medienbrüche vermeiden und Ressourcen effizient einsetzen, was letztendlich zu Kostenersparnissen führt. Angesichts der Kostenvorteile sind allerdings Sicherheitsaspekte im VoIP-Umfeld nicht zu vernachlässigen und gewinnen mit zunehmendem Nutzungsvolumen immer mehr an Bedeutung. Somit kann ein über ein Paket basiertes Netz, basierend auf dem User Datagram Protocol

(kurz UDP) geführtes Gespräch, durch jemanden mit Zugang zum Netz mittels Software regelrecht abgehört werden. Hierbei wird der Datenverkehr z.B. speziell auf Real Time Protocol (kurz RTP) Datenpakete gefiltert. Betroffene Datenpakete können nun kopiert werden und anhand der Informationen im RTP-Header kann eine Audio-Sequenz bzw. ein über IP geführtes Telefongespräch eins zu eins rekonstruiert und abgespielt werden. Mit der freien Software 'Wireshark' (http://www.wireshark.org) für Netzwerkanalyse z.B. sind die erwähnten Funktionalitäten bereits automatisiert verfügbar. Anbetrachts dieser Tatsache sind somit entsprechende Lösungsansätze entwickelt worden, wie z.B. das Secure Real Time Protocol (kurz SRTP) oder Session Initiation Protocol Secure (kurz SIPS), jeweils verschlüsselte Varianten der Protokolle SIP und RTP, welche zusammen als Standards die Hauptelemente in einem VoIP-Gespräch darstellen. In der Praxis erweist sich allerdings die Umsetzung verschiedener Sicherheitskonzepte besonders im Hinblick auf Integration und Kompatibilität bereits vorhandener VoIP-Architekturen, als schwer umzusetzen. Überwiegend sind bei derzeitigen Lösungsansätzen für VoIP-Sicherheit einheitliche Konzepte bzw. Standards aller beteiligten Komponenten erforderlich, was aufgrund der Vielzahl von Anbietern und eingesetzter Hard- und Software bei vielen Unternehmen, die bereits eine bestimmte VoIP-Architektur im Einsatz haben, einen Wechsel aller Komponenten erfordert, was nicht nur mit Kosten sondern auch Abhängigkeit auf spezielle Komponenten in der VoIP-Architektur verbunden ist.

Die vorliegende Arbeit als 'VoIP-Konzept für SSL-geschützte Internettelefonie' soll hierbei im Hinblick auf die derzeitigen Entwicklungen im VoIP-Umfeld ein Konzept darstellen, welches den Einsatz von Internettelefonie um den Aspekt der Sicherheit im VoIP-Datenverkehr ergänzt und zugleich Anforderungen wie vor allem Mehrwert, Wirtschaftlichkeit, Flexibilität und Plattformunabhängigkeit, unter anderem durch eine einfache Integration des Konzeptes in beliebige SIP-basierte VoIP-Architekturen, erfüllt. Ziel ist es, die Möglichkeit von extern auf Web-basierte Dienste und Applikationsserver innerhalb eines (Firmen-)Netzes zuzugreifen, auf die Internettelefonie zu übertragen. Dabei soll Anwendern ein SIP-Client, als JAVA-Applet in einem Web-Browser zur Verfügung stehen, mit welchem die Möglichkeit gegeben ist, über eine SSL-geschützte Verbindung mittels einer als Proxy-Server agierenden VoIP-Komponente, die der zweite Bestandteil des Konzeptes ist, mit dem internen VoIP-Gateway, bzw. dem internen Applikationsserver, zu kommunizieren. Ferner, soll der SIP-Client auch zur internen Nutzung einsetzbar sein, d.h. im internen Netz direkt mit dem internen Applikationsserver, dem vorhandenen bzw. beliebigen VoIP-Gateway auf SIP-Basis kommunizierend. Bei der internen Nutzung erfolgt der Datentransfer über das User Datagram Protocol, (kurz UDP). Hauptkomponenten des Konzeptes stellen somit ein SIP-Client als Applet, dass von extern über TCP/SSL und intern mittels UDP eingesetzt werden kann, und die als Proxy-Server agierende VoIP-Komponente zwischen dem internen Applikationsserver dar.

# Kapitel 2

# Technologischer Hintergrund

## 2.1 Protokolle

In diesem Kapitel sollen für die Ausführungen in der Realisierung des Konzeptes maßgeblichen Protokolle näher erläutert werden. Abgesehen von proprietären Lösungen werden hierzu offene Protokolle und Formate näher betrachtet. Bei proprietären Anwendungen handelt es sich um Software oder allgemein Verfahren die in sich geschlossen und nicht öffentlich sind. Ein bekanntes Beispiel für eine proprietäre VoIP-Anwendung ist 'Skype' [1], dass nicht den allgemein anerkannten Standards entspricht und eigene Übertragungsverfahren bzw. Protokolle verwendet.

Die für das VoIP-Konzept für SSL-geschützte Internettelefonie maßgeblichen Standards basieren auf den allgemeinen Standards des Internets, die durch die IETF (Internet Engineering Task Force) in sog. 'Request for Comments' (kurz RFC) veröffentlicht bzw. definiert sind.

### 2.1.1 Session Initiation Protocol (SIP)

Das Session Initiation Protocol ist nach dem OSI [2] -Schichtenmodell ein Protokoll der Anwendungsebene über das allgemein Multimedia Sitzungen (Sessions) im Internet eingeleitet werden können und ist in RFC 3261 definiert. Im folgenden sollen die Bestandteile des SIP als SIP-Komponenten, die nach RFC 3261 definierte Notation sowie ein Anwendungsbeispiel für einen Verbindungsaufbau mit SIP näher betrachtet werden.

1. freie Telefonie-Software, 2003 eingeführt von Niklas Zennström und Janus Friis, http://www.skype.com
2. Open Systems Interconnection Reference Model beschreibt die Datenübertragung für die Kommunikation informationsverarbeitender Systeme

#### 2.1.1.1 SIP-Komponenten

Das SIP ist ein Klartextprotokoll zur Einleitung von Multimedia-Sitzungen und wird in Form von SIP-Nachrichten mit Parametern und korrespondierten Werten in Textzeilen übertragen. Grundsätzlich werden SIP-Nachrichten unterschieden in SIP-Request´s, also Anfragen, und SIP-Response´s, welche als Antworten zu einer Anfrage korrspondieren.

Die beteiligten Komponenten an über das SIP gesteuerte Multimedia-Sitzungen, lassen sich hierbei grundsätzlich als Client/Server Konzeption darstellen. Ein SIP-Fähiges Endgerät wird in diesem Zusammenhang als User Agent (kurz UA) definiert. Die Bezeichnung User Agent suggeriert hierbei eine Komponente, die Informationen von einem 'User' (zu dt. Benutzer) aufnimmt und in dessen Namen als 'Agent' handelt um Multimedia-Sitzungen auf- oder abzubauen. Weiterhin hat ein UA die Fähigkeit, den Status von Sitzungen, die es einleitet hat oder an denen es teilnimmt, eigenständig zu administrieren. Ein Status einer Sitzung wird hierbei durch die SIP-Parameter Call-ID und CSeq [3] definiert, welche als Werte Informationen über einen korrespondierten SIP-Dialog zwischen Teilnehmern beinhalten (vgl. Kap. 2.1.1.3 'SIP-Verbindungsaufbau').

Hierzu ist eine nähere Betrachtung von sog. SIP-Dialogen erforderlich. Ein SIP-Dialog ist ein Dialog zwischen zwei UA, der zum Austausch von benötigten Informationen für eine bevorstehende Datenübertragung von z.B. Sprachdaten für ein VoIP-Gespräch geführt wird. Dabei werden die Beteiligten Komponenten als User Agent Client (kurz UAC) und User Agent Server (kurz UAS) unterschieden. Die Unterscheidung in UAC und UAS hängt davon ab, ob die jew. Komponente eine Sitzung als UAC initiiert oder als UAS eine Anfrage entgegennimmt und diese beantwortet. In diesem Zusammenhang generiert ein UAC SIP-Requests und ein UAS SIP-Responses. Nachdem ein UA in seiner Funktionalität als Komponente definiert wird, die sowohl eine Sitzung initiieren als auch an einer Sitzung teilnehmen kann, wird für ein Endgerät als UA keine eindeutige Zuordnung als UAC oder UAS vorgenommen, sondern ergibt sich aus der jeweiligen Situation.

Die Definition eines User Agent setzt hierbei als 'User' nicht zwingend einen Benutzer eines Endgerätes voraus. In einer VoIP-Architektur mit SIP kann auch eine technische Komponente, wie z.B. ein SIP-Gateway oder SIP-Proxy, als 'User' eines UA fungieren. Hierbei spricht man von einem SIP-Gateway oder SIP-Proxy als Endgerät. Ein SIP-Gateway wird in diesem Zusammenhang als eine Komponente definiert, die als Schnittstelle einer SIP-Architektur zu einem Netwerk dient, dass auf einem anderen Protokoll für Signalisierungs- und Sprachdaten basiert. Ein SIP-Gateway, dass für beteiligte Komponenten aus den unterschiedlichen Architekturen je als UAC agiert, z.B. zwischen einer VoIP-Architektur und dem konventionellen Telefonnetz, ermöglicht als

3. Command Sequenz, SIP-Parameter als numerischer Wert zur Zuordnung von SIP-Request Abläufen

Endgerät die Kommunikation der Komponenten aus den unterschiedlichen Netzen sowohl für Signalisierungs-, als auch für Sprachdaten.

Ähnlich wie das SIP-Gateway, ist ein SIP-Proxy für beteiligte Komponenten ein UAC, wobei ein SIP-Proxy in Verbindung mit einem SIP-Registrar das Routing von SIP-Nachrichten übernimmt. Dabei kann ein SIP-Proxy auch eigenständig als UAS agieren und stellt den Kommunikationspartner der Teilnehmer dar. Das SIP-Registrar ist für den SIP-Proxy die maßgebliche Informationsquelle zum Weiterleiten von SIP-Anfragen in Form einer klassischen Telefonanlage. Grundsätzlich sind SIP-Server zu unterscheiden in

- Proxy- Server
- Redirect- Server
- Registration- Server

Während ein Proxy-Server eigenständig die Weiterleitung einer Anfrage anhand von Informationen am SIP-Registrar vornimmt, teilt ein Redirect-Server mittels der SIP-Response '302 Moved temporarily' dem anfragenden Teilnehmer lediglich mit, wo er den gewünschten Gesprächspartner anhand der Registrierung derzeit erreichen kann, worauf der anfragende Teilnehmer seine Anfrage an den gewünschten Gesprächspartner versenden kann.

Als Registration-Server wird eine Komponente bezeichnet, die einen SIP-Request 'REGISTER' annimmt und anhand der übermittelten Daten das Registrar aktualisiert. Die Administration des SIP- Registrar erfolgt somit über einen SIP-Registrierungsvorgang. Hierbei leitet ein UAC, also z.B. der Nutzer des IP-Telefons, beim SIP-Registration-Server über einen SIP-Request 'REGISTER' den Registrierungsvorgang ein. Voraussetzung für eine Registrierung ist die Konfiguration von Benutzername und Passwort als SIP-Account am Registrar. Somit hat auch jeder Teilnehmer eine sog. SIP-URI [4], bestehend aus einer Teilnehmerkennung und einer IP-Adresse in der Notation 'Teilnehmerkennung@IP-Adresse', wobei sich der Abschnitt 'IP-Adresse' aus der jeweiligen Registrierung ergibt, so dass der UAC über den einen SIP-Registrierungsdialog mit dem Registration-Server seine für SIP-Anfragen zu verwendende IP-Adresse mitteilt. Die SIP-URI für z.B. einen Teilnehmer mit der IP-Adresse 172.22.80.194 und der Teilnehmerkennung 22 gestaltet sich somit zu <sip:22@172.22.80.141>.

In Abb. 2.1 soll die Funktionalität von SIP-Servern in einer VoIP-Architektur veranschaulicht werden.

4. Uniform Resource Identifier, Notation zur Bezeichnung von Komponenten im Internet

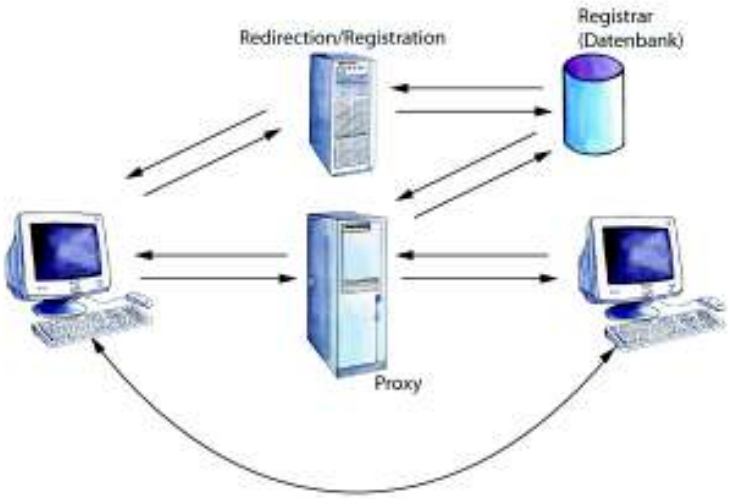


**Abbildung 2.1:** SIP- Server Architektur

#### 2.1.1.2 SIP- Notation

Eine SIP Nachricht besteht allgemein aus einem SIP-Header, wobei optional ein Content als SDP-Teil vorhanden sein kann, der in seinen Einzelheiten in Kap. 2.1.2 'Session Description Protocol (SDP)' beschrieben wird. Die Beschaffenheit der einleitenden Zeile einer SIP-Nachricht ist abhängig davon, ob es sich bei der SIP-Nachricht allgemein um eine Anforderung, also ein SIP- Request oder eine Antwort auf eine Anforderung als SIP-Response handelt. Die einleitende Zeile einer SIP-Request wird definiert durch

Method SP Request-URI SP SIP-Version CRLF

Im weiteren Verlauf werden bei der Darstellung von Definitionen im Bezug auf die Notation zur Vereinfachung der Darstellbarkeit Leerzeichen als 'SP' und die Kombination aus Wagenrücklauf und Zeilenvorschub (\r\n) als CRLF [5] dargestellt. Allgemein gilt für Zeilen mit Paramtern und korrespondierten Werten, dass die Terminierung einer jeden Zeile als Informationseinheit mit CRLF gekennzeichnet wird.

Demnach setzt sich die einleitende Zeile einer SIP-Request aus den Parametern 'Method', einer 'Request-URI' und der SIP-Version getrennt durch je einem Leerzeichen zusammen. Die für SIP-Requests nach RFC 3261 definierten 'Method's, d.h. Schlüsselwörter für eine SIP-Request als Anfrage sind

- REGISTER: Anmeldung beim SIP-Gateway

5. Carriage Return, Line Feed engl. für Wagenrücklauf und Zeilenvorschub

- INVITE: Initiierung einer Verbindung
- ACK: Terminierende Bestätigung über den Empfang einer SIP-Nachricht
- CANCEL: Verbindungsaufbau abbrechen
- BYE: Beenden von Verbindungen
- OPTIONS: Informationen über Endsysteme

Die Request-URI beinhaltet in diesem Zusammenhang die Information über den Empfänger der SIP-Request. Die einleitende Zeile einer SIP-Response wird im Gegensatz zur SIP-Request mit einer dreistelligen Schlüsselnummer für die Art der SIP-Response und einem korrespondierten Response-Text angegeben. Die erste Zeile für eine typische SIP-Response wäre z.B. 'SIP/2.0 200 OK'. Die Schlüsselnummernbereiche für SIP-Responses definieren sich wie folgt:

- 1xx: Provisional: Anfrage empfangen, Prozess wird noch ausgeführt, z.B. 100 Trying
- 2xx: Success: Anfrage Empfangen, Verstanden und Akzeptiert, z.B. 200 OK
- 3xx: Redirection: Gegenstelle hat eine neue Adressierung, z.B. 301 Moved Permanently
- 4xx: Client Error: Die Anfrage ist fehlerhaft, z.B. 404 Not Found, 405 Method Not Allowed
- 5xx: Server Error: Serverfehler, z.B. 503 Service Unavailable
- 6xx: Global Failure: Serverunabhängiger Fehler, z.B. 600 Busy Everywhere (Besetzt)

Die Art der Nachricht (Request/Response) bestimmt den weiteren Verlauf der Handlung, wobei die weiteren Parameter im Header hierfür benötigte Informationen bereitstellen. Insgesamt gibt es 45 verschiedene Parameter zu denen Angaben gemacht werden können oder je nach Handlung gemacht werden müssen. Die Notation für die Zeilen einer SIP-Nachricht ist definiert als

Parameter: SP Wert CRLN

Die Felder Via, From, To und Cseq gehören in nahezu allen Requests oder Responses mit zu den Parametern die mit aufgeführt werden müssen. Während der Parameter 'Via' unter anderem angibt, über welchen Server und Port (z.B. SIP-Standard-Port 5060) die Nachricht versendet wird, geben die Felder From und To die benötigten Werte für Absender und Empfänger der Nachricht an. Diese Parameter können jedoch, z.B. bei Responses, auch die Ursprungswerte einer Anfrage enthalten, d.h. dass nicht immer der

aktuelle Absender im From-Feld angegeben sein muss wie es der Name des Parameter´s suggeriert.

#### 2.1.1.3 SIP- Verbindungsaufbau

Anhand der in Kap. 2.1.1.2 dargestellten Notation, soll im Folgenden im Bezug auf die in Kap. 2.1.1.1 dargestellten SIP-Komponenten anhand eines Beispiels der Ablauf für einen Verbindungsaufbau mit SIP näher betrachtet werden.

Gegenstand des Anwendungs-Beispiels sind die zwei Teilnehmer die je eine VoIP- Software mit SIP als Basis an ihrem Rechner haben und am gleichen SIP-Registrar jeweils mit Benutzername und Passwort als Teilnehmer angelegt sind. Im Verlauf soll TeilnehmerI (zur Vereinfachung im weiteren Verlauf benannt als 'Bob') einen TeilnehmerII (zur Vereinfachung im weiteren Verlauf benannt als 'Alice') anrufen, wobei Alice die Rufnummer 22 hat und Bob die 33. Beide sind beim SIP-Registrar mit jeweils mit ihrer IP-Adresse und der sich ergebenden jeweiligen SIP-URI <sip:22@172.22.80.141> für Alice und <sip:33@172.22.20.197> für Bob registriert. In Abb. 2.2 soll der nachstehend im einzelnen beschriebene Ablauf veranschaulicht werden.

Der SIP-Proxy ist zur Vereinfachung im gleichen Netzwerk und hat die IP-Adresse 172.22.20.190. Um Alice anzurufen versendet Bobs SIP-Client, nachdem Bob die 22 als Zielrufnummer angegeben und auf 'Wählen' geklickt hat, an den SIP-Proxy einen INVITE-Request, der nachstehend dargestellt wird:

**Listing 2.1:** INVITE- Message

```
INVITE sip:22@172.22.20.190 SIP/2.0
Via: SIP/2.0/UDP 172.22.20.197:5060; branch=z9hG4bK-259kf24qcmte; rport
From: "Bob"<sip:33@172.22.20.190>; tag=abewiu0dwg
To: "Alice" <sip:22@172.22.20.190;user=phone>
Call-ID: 3c3274ad04e2-ex0zeriq80aq@172.22.20.197
CSeq: 1 INVITE
Max-Forwards: 70
Contact: <sip:33@172.22.20.197:5060>
Content-Type: application/sdp
Content-Length: 342
```

Bei der in Listing 2.1.1.3 dargestellten SIP-Request INVITE ist zu beachten, dass der Content als SDP-Teil nicht mit aufgeführt ist und in seinen Einzelheiten in Kap. 2.1.2 dargestellt wird. Bevor der SIP-Proxy diesen an Alice weiterleitet, muss Bob den INVITE vorerst authentifizieren. Hierzu erhält er eine SIP-Response '407 Proxy Authentication Required' vom SIP-Proxy, die den Parameter

```
Proxy-Authenticate: Digest realm="proxy", nonce="0e7a2a63"
```

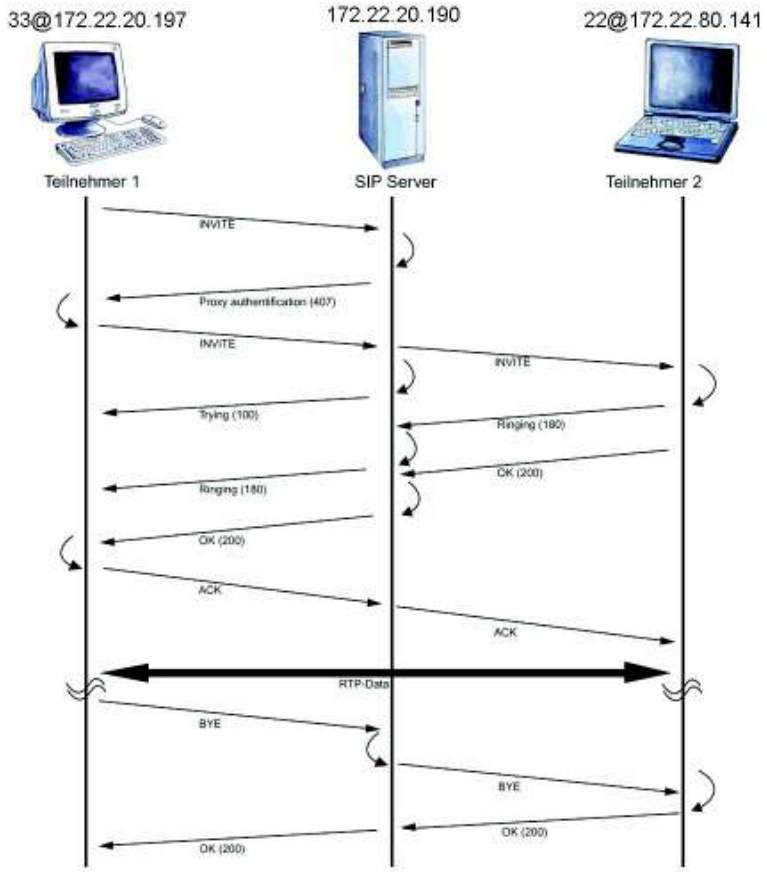


**Abbildung 2.2:** Verbindungsaufbau mit SIP

beinhaltet. Bob versendet erneut die gleiche INVITE, allerdings fügt er diesmal nachstehenden Parameter ein:

**Listing 2.2:** Proxy-Authorization

```
Proxy-Authorization: Digest username="33", realm="asterisk", nonce="0e7a2a
    63", uri="sip:22@172.22.20.190",response="f457802909a6fb5679083751db
    05252a", algorithm=md5
```

Zu beachten ist hierbei der Wert für den Parameter "response". Dieser beinhaltet eine verschlüsselte Zeichenkette die mit dem MD5 [6]- Algorithmus anhand der Teilnehmerparameter von Bob und dem nonce-Wert der SIP-Response '407 Proxy Authentication Required' des SIP-Proxy generiert wird. Der MD5-Algorithmus ist eine kryptographische Hash-Funktion, dessen Anwendung in RFC 1321 'The MD5 Message-Digest Algorithm' beschrieben ist. Nun leitet der SIP-Proxy Bobs Anfrage an Alice weiter und sendet Bob gleichzeitig eine SIP-Response '100 Trying'.

**Listing 2.3:** 100 Trying

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP 172.22.20.197:5060; branch=z9hG4bK-259kf24qcmte; rport
From: "Bob"<sip:33@172.22.20.190>; tag=abewiu0dwg
To: "Alice" <sip:22@172.22.20.190;user=phone>
Call-ID: 3c3274ad04e2-ex0zeriq80aq@172.22.20.197
CSeq: 2 INVITE
Contact: <sip:33@172.22.20.197:5060>
Content-Length: 0
```

Da Bob für die Authentifizierung ein zweites INVITE für die gleiche Session versenden musste, hat sein SIP-Client den CSeq-Wert inkrementiert. Dieser Wert ist nun die Sequenznummer für nachfolgende Nachrichten, die sich auf dieses in diesem Zusammenhang Re-INVITE beziehen, weshalb auch der Parameter CSeq der SIP-Response '100 Trying' des SIP-Proxy den Wert '2' beinhaltet.

Im Bezug auf die Parameter welche den Status einer Sitzung definieren, soll die Funktionsweise von einem 'Re-INVITE' näher betrachtet werden. Hierzu vergibt der UAC dem ursprünglichen INVITE (vgl. Listing 2.1.1.3) eine eindeutige Call-ID. Alle weiteren SIP-Nachrichten die mit dieser Anfragen korrespondieren, müssen die jeweilige Call-ID beinhalten, so dass jede SIP-Nachricht von den beteiligten Komponenten explizit der Sitzung zugeordnet werden kann. In Zusammenhang mit der Call-ID, stellt der Parameter CSeq mit einem numerischen Wert den Ablauf von ggf. mehreren Anfragen als SIP-Request die mit einer Call-ID korrespondieren. Versendet ein UAC eine erneute SIP- Request INVITE mit der gleichen Call-ID, so handelt es sich um eine Re-INVITE. Ein Re-INVITE wird z.B. auch dann versendet, um die Parameter welche für eine aktive

6. Message-Digest Algorithm 5, kryptographische Hash-Funktion die einen 128-Bit-Hashwert erzeugt

oder sich im Aufbau befindende Sitzung definiert wurden zu verändern. Bei einem Re-INVITE wird der numerische Wert des CSeq von der mit der Call-ID korrespondierenden ursprünglichen INVITE inkrementiert.

Falls der SIP-Client von Alice den Anruf entgegennehmen kann und mindestens ein Codec der aus dem SDP des eingehenden INVITE zur Verfügung stehenden Codecs bei Alices SIP-Client auch verfügbar ist, versendet der SIP-Client von Alice an den Server eine SIP-Response '180 Ringing' und signalisiert gleichzeitig Alice durch einen Klingelton den eingehenden Anruf.

Sobald Alice den Anruf entgegennimmt, versendet der SIP-Client nachstehende SIP-Response:

**Listing 2.4:** 200 OK

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 172.22.20.190:5060; branch=z9hG4bK71234e17
From: "Bob"<sip:33@172.22.20.190>; tag=abewiu0dwg
To: "Alice" <sip:22@172.22.20.190;user=phone>
Call-ID: 19832cf960c2362d21e925551058d7b8@172.22.80.141
CSeq: 2 INVITE
Contact: <sip:22@172.22.80.141:5060>
Content-Type: application/sdp
Content-Length: 221
```

Die SIP-Response aus Listing 2.1.1.3 beinhaltet hierbei einen SDP-Teil der in in Kap. 2.1.2 im einzelnen erläutert wird.

Während bei einer INVITE häufig mehrere Codecs zur Verfügung stehen, die der SIP-Client des Anrufers unterstützt, wird beim OK des Angerufenen einer aus diesen ausgewählt und dem Anrufer mitgeteilt. Nun weiß Bobs SIP-Client Alices IP-Adresse und den zu erwartenden Codec, wodurch die Aufgaben des SIP und SDP erledigt sind und nach einem ACK request von Bob, also einer Bestätigung für das Annehmen des Gespräches von Alice, das Gespräch mittels des Real-Time-Protocol direkt zwischen den SIP-Clients aufgebaut werden kann. In Fällen, bei denen die Gesprächspartner keinen gemeinsamen Codec zur Verfügung haben, kann das Gespräch trotzdem aufgebaut werden, indem der Asterisk-Server als Übersetzer fungiert und die RTP-Pakete über diesen versendet werden.

Wenn Bob das Gespräch beendet, sendet sein SIP-Client eine BYE-Request an Alice.

**Listing 2.5:** BYE Message

```
BYE sip:22@172.22.80.141 SIP/2.0
Via: SIP/2.0/UDP 172.22.20.191; rport; branch=z9hG4bKac16508d
From: 'Bob' <sip:33@172.22.20.190>
To: <sip:22@172.22.80.141:5060>
Contact: <sip:33@172.22.20.190>
```

```
6 Call-ID: 19832cf960c2362d21e925551058d7b8@172.22.80.141
7 CSeq: 3 Bye
8 Content-Length:0
```

Nachdem der SIP-Client von Alice mit einem SIP-Response '200 OK' geantwortet hat, ist das Gespräch beendet.

### 2.1.2 Session Description Protocol (SDP)

Das Session Description Protocol wird im Zusammenhang mit SIP für die Festlegung von technischen Parametern einer Multimedia- Sitzung als 'Content' für den SDP- Teil bei SIP- Nachrichten angewandt und ist in RFC 2327 'SDP: Session Description Protocol' definiert. Die für einen Verbindungsaufbau mittels SIP benötigten technischen Parameter werden in Tab. 2.1.2 dargestellt. Das SDP definiert hierbei allgemein Multimedia-Sessions, weshalb auch andere Angaben mit weiteren Schlüsselbezeichnungen möglich sind oder Parameterangaben sich überschneiden können, wobei in diesem Kapitel die für VoIP maßgebliche Parameterangaben behandelt werden.

Für die in Tab. 2.1.2 aufgeführten Parameter ist nach dem SDP folgende Notation definiert:

v=0
o=<username> <session id> <network type> <address type> <address>
s=<session name>
c=<network type> <address type> <connection address>
t=<start time> <stop time>
m=<media> <port> <transport> <fmt list>
a=rtpmap:<payload type> <encoding name>/<clock rate>[/<encoding parameters>]

Das Versionsfeld 'v' deklariert die Version des SDP, derzeit wird diese ausschließlich mit '0' angegeben.

Die Schlüsselbezeichnung 'o' steht für 'Owner' und enthält Angaben über den 'Erzeuger' der Nachricht.

's' steht für 'Session Name', wodurch jeder Session zur Differenzierbarkeit bei parallelen Sessions ein Name zugeteilt wird. Meistens wird hier die Bezeichnung des SIP- Clients oder die Initialen des Herstellers der korrespondierten Software verwendet und bei parallelen Sessions (Konferenzen) numerisch erweitert.

**Tabelle 2.1:** SDP- Parameter

| Parameter | Name | Beschreibung |
|---|---|---|
| v | version | Protokoll Version |
| o | owner | enthält Angaben über den 'Erzeuger' der Nachricht |
| s | session name | Name der Session |
| c | connection information | Netzwerkinformationen |
| t | time | Dauer der Session |
| m | media | Medienangaben |
| a | attribute | Attribute der Medienangaben |

Die Angaben zur Verbindung werden unter 'c' für 'Connection Information' gemacht. 'IN' steht hierbei am häufigsten unter Netzwerktyp für Internet, wobei in diesem Zusammenhang der Adresstyp meistens IP4 ist, möglich wäre für den Netzwerktyp Internet z.B. auch IP6 für Version 6 des Internet Protocol (kurz IPv6).

Falls die Verbindung Zeitbegrenzt sein soll, kann bei der Zeitangabe eine Anfangs- und Endzeit angegeben werden. Dies kann vor allem dann eine Rolle spielen, wenn die Nutzung des SIP- Clients z.B. von einem Guthabenkonto oder Gebühren abhängig ist, ansonsten sind die Werte häufig mit '0 0' besetzt.

Bei den Medienangaben ist für VoIP der Parameter 'media' immer 'audio'. Clients, die z.B. auch Echtzeit- Bildübertragung unterstützen, können für die Übertragung von Bildsignalen mit dem Wert 'video' entsprechende technische angaben machen. Der Parameter 'transport' wird üblicherweise mit dem Wert RTP/AVP belegt, wobei AVP allgemein für Audio/Video Profile angegeben wird. Mit 'fmt list' wird der sog. Payload Type (kurz PT), also die die Beschaffenheit des Payload der RTP- Pakete beschrieben. Bei Audiodaten werden hier als Parameter Schlüsselnummern nach RFC 1890 'RTP Profile for Audio and Video Conferences with Minimal Control' für standardisierte Datenkompressionsverfahren angegeben, wobei auch mehrere Angaben hintereinander gemacht werden können. In Tabelle 2.1.2 werden Datenkompressionsverfahren für Audio- Daten (auch Codec) in Zusammenhang mit den referenzierten Schlüsselnummern als Payload Type aufgeführt.

Für jede der in 'fmt list' stehenden Schlüsselnummern wird mit den Angaben im Feld 'a' für attributes, das jew. Kodierverfahren genau definiert. Stehen in der 'fmt list' mehrere Schlüsselnummern, werden mehrere Felder mit dem Bezeichner 'a' erstellt.

Zur Veranschaulichung wird in Listing 2.1.2 für die in Kap. 2.1.1.3 in Listing 2.1.1.3 dargstellte SIP- Request INVITE der dazugehörige SDP- Teil dargestellt.

**Tabelle 2.2:** Payload- Types

| PT | Codec | Taktfrequenz |
|---|---|---|
| 0 | PCMU | 8000 |
| 1 | 1016 | 8000 |
| 2 | G721 | 8000 |
| 3 | GSM | 8000 |
| 4 | n/a | 8000 |
| 5 | DVI4 | 8000 |
| 6 | DVI4 | |
| 7 | LPC | 8000 |
| 8 | PCMA | 8000 |
| 9 | G722 | 8000 |
| 10 | L16 | 44100 |
| 11 | L16 | 44100 |
| 12 | n/a | - |
| 13 | n/a | - |
| 14 | MPA | 90000 |
| 15 | G728 | 8000 |

**Listing 2.6:** SDP- Content

```
v=0
o=- 3367205336 3367205336 IN IP4 172.22.80.141
s= TestPhone
c= IN IP4 172.22.80.141
t= 0 0
m= audio 49152 RTP/AVP 0
a= rtpmap: 0 PCMU/8000
```

### 2.1.3 Real-Time Transport Protocol (RTP)

Das RTP ist definiert in RFC 3550. Im Gegensatz zu den Protokollen SIP und SDP besteht das RTP aus einem Header mit einer fest definierten Struktur welche nicht als Klartext sondern binär dargestellt wird und einem Payload, der in Zusammenhang mit VoIP digitalisierten Sprachsignale beinhaltet. Mittels des RTP sollen Echtzeitübertragungen in Form eines kontinuierlichen Datenstroms über das Internet ermöglicht werden. Der RTP-Header beinhaltet hierbei für die Interpretation der als RTP-Pakete versendeten Datensequenzen maßgebliche Informationen.

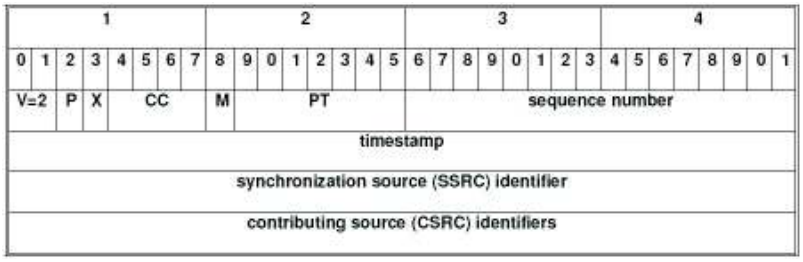


**Abbildung 2.3:** RTP- Header

Das Real-Time Transport Protocol ist ausgehend von den für Echtzeitanwendungen sich ergebenden Voraussetzungen, auf Basis für einen verbindungslosen paketorientieren Einsatz konzipiert. Auschlaggebende Parameter für eine entsprechend einem Audio-Format [7] zu übertragende Sequenz von Audio-Daten sind z.B. Paketgrößen als 'Frame-Size' sowie die Beschaffenheit der Audio-Daten. Bei Datenkompressionsverfahren kann z.B. ein Datenpaket welches nicht in seiner ursprünglichen Reihenfolge empfangen wurde den Abriss der Audio-Wiedergabe bzw. das Aussetzen des Codec [8] hervorrufen.

Die Parameter, des mittels des in Abb. 2.3 dargestellten RTP-Header erlauben mittels entsprechender RTP-Datenpakete die Implementierung von Mechanismen die eine

7. Beschreibt den Aufbau einer Audiodatei als Dateiformat
8. Verfahren, mit dem Daten digital kodiert und dekodiert werden

Echtzeitübertragung über das Internet ermöglichen und werden nachstehend näher beschrieben.

- V- Version (2 Bit) RTP- Version
- P- Padding (1 Bit) wird gesetzt, wenn die Rohdaten des Datenpaketes um zusätzliche 'Füll-Bytes' erweitert wurden, kann z.B. bei Einsatz einer Verschlüsselung bei bestimmten Audio-Formaten erforderlich sein, wenn die Frame-Size des Audio-Formates mit der für die Verschlüsselung definierten Paketgröße nicht als ganzes teilbar ist
- X- Extension- Bit (1 Bit), kennzeichnet ob eine Verlängerung des Header vorliegt, welche optional möglich ist
- CC- CSRC Count (4 Bit) Anzahl der optionalen CSRC Einheiten
- M- Marker- Bit (1 Bit), wird verwendet um bestimmte Ereignisse in innerhalb eines RTP-Datenstroms kennzuzeichnen, wie z.B. das Ende einer 'Stille'- Sequenz, bei der keine Audio-Daten zu akustischen Wiedergabe angefallen sind
- PT- Payload Type (7 Bit), Schlüsselnummer für die Beschaffenheit der Rohdaten, vgl. Tab. 2.1.2 'Payload-Types' Kap. 2.1.2 'Session Description Protocol (SDP)'
- Sequence Number (16 Bit), identifiziert expizit jedes RTP-Datenpaket und wird für jedes RTP-Datenpaket inkrementiert
- Timestamp (32 Bit), dient zur Synchronisierung des Empfangs und der Wiedergabe mit der Laufzeit
- SSRC- Synchronization Source (32 Bit), identifiziert explizit einen RTP- Datenstrom einer Quelle
- CSRC- Contributing Sources (0 - 15 Einheiten á 32 Bits)(optional), kann aus bis zu 15 Einheiten mit je 32 Bits bestehen, wobei die Anzahl der Einheiten im Feld CSRC-Count angegeben wird. 'Contributing Sources' bezeichnen bei einem RTP-Datenstrom der aus mehreren Quellen stammt die beteiligten Komponenten, wobei für Audio-Datenpakete der jeweilige SSRC-Parameter der Komponente angegeben wird

### 2.1.4 Secure Sockets Layer (SSL)

Das SSL als Verschlüsselungsprotokoll zur Datenübertragung im Internet soll in diesem Kapitel als Grundlage für die wesentliche Funktionalität des HOB WebSecureProxy, der in Kap. 2.3 dargestellt wird, in seinen wesentlichen Grundzügen beschrieben werden.

Das SSL-Protokoll besteht aus einer Reihe von Nachrichten (Messages) und Regeln darüber, wann welche Nachricht gesendet oder nicht gesendet werden soll. An dieser Stelle soll erwähnt werden, dass der für das VoIP-Konzept für SSL-geschützte Internettelefonie maßgebliche SSL-Verbindungsaufbau, als vom HOB WebSecureProxy (Kap. 2.3)gegebene Funktionalität angenommen wird. In diesem Zusammenhang werden mögliche Verschlüsselungsmechanismen an sich und die detaillierte Beschaffenheit der erwähnten SSL-Nachrichten nicht näher behandelt.

Im Ablauf eines SSL-Verbindungsaufbaues werden die beteiligten Komponenten als Client und Server unterschieden. Die Unterscheidung in Client und Server der beteiligten Komponenten spielt vor allem bei der Aushandlung der Verschlüsselungsparameter eine entscheidende Rolle. Beim Einsatz von SSL z.B. zur sicheren Übertragung von Daten über das Web, stellt der Web-Browser den Client dar und der Web-Server, der Informatoinen über eine Web-Seite bereitstellt, den Server.

Zur genaueren Betrachtung des SSL-Verbindungsaufbaues sind zunächst allgemein Kryptosysteme in symmetrische und asymmetrische zu unterscheiden. Bei einem sysmmetrischen Kryptosystem sind beide Parteien im Besitz eines geheimen Schlüssels, anhand dessen der Datenverkehr von beiden Parteien ver- und entschlüsselt werden. Ein sysmmetrisches Kryptosystem setzt jedoch voraus, dass beide Parteien im Besitz des geheimen Schlüssels sind. Soll hierzu der geheime Schlüssel selbst mittels des Internet übertragen werden, ergibt sich die Problematik, dass die Übertragung des Schlüssels wiederum verschlüsselt erfolgen müsste. Hierzu wird im Rahmen des SSL-Verbindungsaufbaues ein asymmetrisches Kryptosystem eingesetzt.

Bei einem asymmetrischen Kryptosystem besitzt jeder der kommunizierenden Parteien, seien es hier Client und Server, je einen öffentlichen und privaten Schlüssel. Möchte der Client dem Server eine verschlüsselte Nachricht schicken, in diesem Fall einen privaten, geheimen Schlüssel, so verschlüsselt er die Informationen mit dem öffentlichen Schlüssel des Servers. Dadurch, dass die Nachricht mit dem öffentlichen Schlüssel verschlüsselt jedoch nur mit den privaten Schlüssel entschlüsselt werden kann, kann die Nachricht vom Client der den öffentlichen Schlüssel des Servers angewandt hat sicher versendet werden. Der Server kann dann die Nachricht mit dem geheimen privaten Schlüssel wieder entschlüsseln. Wie aus der Funktionsweise ersichtlich, benötigen Client und Server für eine Übertragung von verschlüsselten Informationen in beide Informationsrichtungen jeweils den öffentlichen Schlüssel der anderen Partei. Dieser Informationsaustausch findet anhand des in Abb. 2.4 dargestellten Ablaufes statt.

In Tabelle 2.1.4 werden die beim Verbindungsaufbau in Frage kommenden Nachrichten in alphabetischer Reihenfolge beschrieben.

Ausgehend von den in Tab. 2.1.4 dargestellten möglichen Nachrichten zwischen Client und Server bei einem SSL-Verbindungsaufbau, wird in Abb. 2.4 der Ablauf eines Verbindungsaufbaus verdeutlicht.

**Tabelle 2.3:** Nachrichten beim SSL-Verbindungsaufbau

| **Nachricht** | **Beschreibung** |
|---|---|
| Alert | Informiert die Gegenstelle über eine mögliche Sicherheistlücke oder einen allgemeinen Fehler |
| ApplicationData | Verschlüsselte Nutzdaten |
| Certificate | Übermittlung des Zertifikates für den öffentlichen Schlüssel |
| CertificateRequest | Serverantwort zur Akzeptanz des öffentlichen Schlüssel Zertifikates |
| CertificateVerify | Bestätigung des Client über die Kenntniss darüber, dass der geheime Schlüssel mit seinem Zertifikat des öffentlichen Schlüssels korrespondiert |
| ChangeCipherSpec | Hinweis zum bevorstehenden Einsatz einer ausgehandelten Verschlüsselung |
| ClientHello | Nachricht vom Client im Bezug auf die Unterstützung der angeforderten Verschlüsselungsart |
| ClientKeyExchange | Kryptographische Schlüssel des Clients für die Datenübertragung |
| Finished | Abschluss aller Aushandlungen und des Verbindungsaufbaus |
| HelloRequest | Anforderung von Server an den Client den SSL-Dialog zur Aushandlung der Verschlüsselungsparameter zu beginnen |
| ServerHello | Servernachricht im Bezug auf die für die bevorstehende Datenübertragung anzuwendende Verschlüsselungsart |
| ServerHelloDone | Servernachricht, dass alle Anfragen vom Client zum Verbindungsaufbau wurden vom Server abgearbeitet |
| ServerKeyExchange | Kryptographische Schlüssel des Servers für die Datenübertragung |

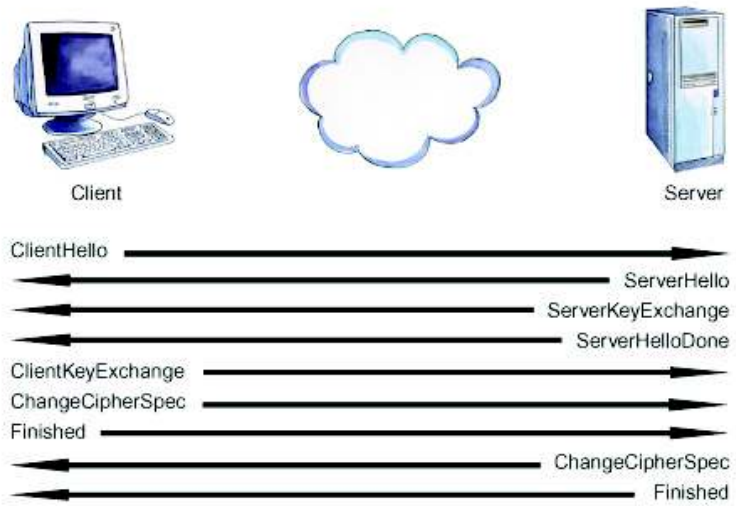


**Abbildung 2.4:** SSL-Verbindungsaufbau

Der in Abb. 2.4 dargestellte Ablauf soll im folgenden näher erläutert werden. Demnach beginnt der Dialog zwischen Client und Server mit einer 'ClientHello' Nachricht, die vom Client an den Server, mit Absicht einen Dialog zur Aushandlung der für eine verschlüsselten Datenübertragung benötigten Parameter zu beginnen. Hierbei handelt es sich um eine Auswahl an Methoden für eine asymmetrische Verschlüsselung um anhand diesem den geheimen Schlüssel zu Übertragen, der im Laufe der Datenübertragung auf Basis einer symmetrischen Verschlüsselung angewendet werden kann. Eine 'ClientHello' Nachricht besteht dabei aus den in Tab. 2.1.4 dargestellten Parametern.

Für das auf ein 'ClientHello' erwartete 'ServerHello' sind die in Tab. 2.1.4 aufgeführten Parameter des Client maßgeblich. Unterstützt der Server die SSL-Version und mindestens ein Verfahren das unter den Parametern 'CipherSuite' und 'CompressionMethods' vom Client vorgeschlagen wurde, so antwortet der Server mit einem 'ServerHello', dass eine ähnliche Beschaffenheit hat wie das 'ClientHello'. Hierbei ist zu Unterscheiden, dass der Sever bei seiner Antwort eine Verfahren für die Erwähnten Parameter auswählt und im 'ServerHello' für den weiteren Verlauf des Verbindungsaufbaus festlegt. Im Bezug auf die Festlegung des asymmetrischen Verfahrens zur Übertragung des geheimen Schlüssels kann hier z.B. das RSA [9]-Verfahren vereinbart werden.

Während die 'ServerHello'-Nachricht des Servers Informationen über die Art der Verschlüsselung wie z.B. den anzuwendenden Verschlüsselungverfahren beinhaltet, wird mit

9. asymmetrisches Kryptosystem benannt nach seinen Erfindern Ronald L. Rivest, Adi Shamir und Leonard Adleman

**Tabelle 2.4:** 'ClientHello' Parameter im SSL-Verbindungsaufbau

| **Parameter** | **Beschreibung** |
|---|---|
| Version | Neueste Version des SSL-Protokoll die der Client unterstützt |
| RandomNumber | 32-byte Zufallszahl zum Einsatz für kryptographische Berechnungen |
| SessionID | Identifiziert explizit eine SSL-Session |
| CipherSuites | Parameter im Bezug auf Kryptographie die vom Client unterstützt werden |
| CompressionMethods | Datenkompressionsverfahren die vom Client unterstützt werden |

der folgenden SSL-Nachricht 'ServerKeyExchange' aus dem in Tab. 2.1.4 dargestelltem Ablauf, der entsprechend den Parametern anzuwendende öffentlichen Schlüssel des Server dem Client übermittelt. Durch den 'ClientKeyExchange' übermittelt wiederum der Client dem Server seinen öffentlichen Schlüssel, wodurch die Aushandlung der Verschlüsselungsverfahrens für asymmetrische Kryptosysteme abgeschlossen ist. Gleichzeitig wird der für die folgende Datenübertragung ('ApplicationData') anzuwendende geheime Schlüssel im Rahmen der symmetrischen Verschlüsselung nach einem Zufallsprinzip vom Client generiert und übertragen. Entscheident ist, dass der geheime Schlüssel mit dem öffentlichen Schlüssel des Server verschlüsselt ist. Empfängt der Server den mit dem öffentlichen Schlüssel verschlüsselten geheimen Schlüssel, so erhält dieser den geheimen Schlüssel indem er die Nachricht mit seinem privaten Schlüssel entschlüsselt. Somit haben beide Parteien einen gemeinsamen geheimen Schlüssel wodurch die Vorraussetzungen für ein symmetrisches Verschlüsselungsverfahren im weiteren Verlauf erfüllt sind.

## 2.2 Die Telefonanlage Asterisk

Asterisk ist eine freie Software, welche die Funktionalität einer Telefonanlage für die Verwendung in einer VoIP-Architektur ermöglicht. Hierzu gehören unter anderem Rufumleitung, Sprachmailbox, Sprachkonferenzen, Warteschleifen, das Halten eines Gespräches usw., wobei diee Telefonanlage Asterisk die in Kap. 2.1.1.1 dargestellten SIP-Komponenten vereint:

- Registration-Server
- Proxy-Server
- Redirection-Server
- SIP-Gateway

Die Administrierung des SIP-Registrars erfolgt über die Datei sip.conf, in welcher SIP-Accounts mit Benutzerdaten, Passwort und weiteren Konfigurationen angelegt werden. Zu den Hauptmerkmalen des Asterisk gehören die sich aus der Integrierung der Komponente als SIP-Proxy ergebende Funktionalitäten. Demnach stellt Asterisk bei einem Verbindungsaufbau zwischen zwei Endgeräten jeweils einen 'user agent' (vgl. Kap. 2.1.1.1 'SIP-Komponenten') und somit für den Signalisierungsdatenverkehr und ggf. auch den Sprachdatenverkehr eine terminierende Komponente dar. Somit kann Asterisk auch als 'Übersetzer' zwischen Endgeräten fungieren und die Kommunikation von zwischen in ihrer Beschaffenheit im Bezug auf verwendete Standards und Protokolle unterschiedlichen Endgeräten für Signalisierungs- oder Sprachdaten gewährleisten. Nachstehende Abbildung soll den Einsatz der Telefonanlage Asterisk in einer VoIP-Architektur verdeutlichen.

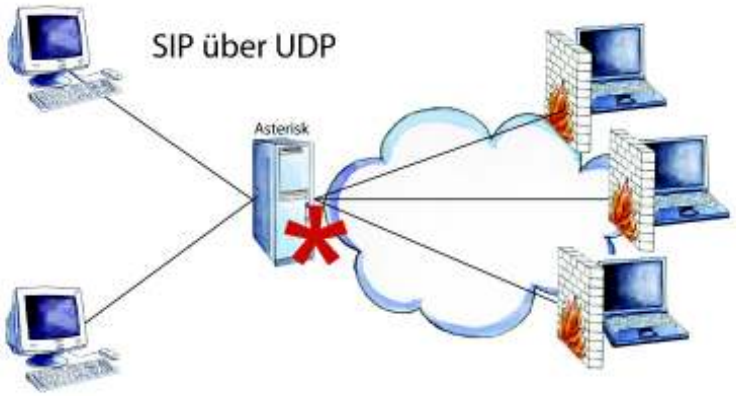


**Abbildung 2.5:** VoIP-Architektur mit Asterisk

Wird seitens eines in Abb. 2.5 dargestellten SIP-Clients ein Verbindungsaufbau initiiert, so vermittelt Asterisk als SIP-Proxy zwischen den beiden Gesprächspartnern die benötigten Informationen für eine Übertragung von Sprachdaten. Ist zwischen den beiden Endgeräten eine Sprachdatenübertragung auf gemeinsamer Basis, in Zusammenhang der in Kap. 2.1.2 'Session Description Protokol (SDP)' dargestellen Datenkompressionsverfahren ('Codec') und dem in Kap. 2.1.3 dargestellten RTP möglich, so kann die Übertragung der Sprachdatenpakete unmittelbar zwischen den beiden Endgeräten erfolgen. Wird nicht mindestens ein gemeinsamer Codec unterstützt, so erfolgt zwischen Asterisk und jeweils den beiden Endgeräten eine Sprachdatenübertragung in sofern, dass Asterisk empfangene Sprachdatenpakete an das jeweils korrespondierte Endgerät in ein für dieses Endgerät anwendbares Format 'übersetzt' und weiterleitet.

Im Bezug auf die Funktionalität des Asterisk als SIP-Proxy, soll hierbei auf den Vorgang des sog. 'Re-INVITE' näher eingegangen werden. Demnach kann der Asterisk als für Signalisierungsdaten terminierende Komponente in der VoIP-Architektur den Verbindungsaufbau situativ steuern. Wird z.B. während einer Sprachdatenübertragung die ohne eine Beteiligung des Asterisk, also unmittelbar zwischen zwei Endgeräten stattfindet, das Gespräch in eine Warteschleife versetzt, so kann Asterisk mittels einer SIP-Request INVITE an die beteiligten Komponenten, die jeweilige Sprachdatenübertragung 'umlenken'. Hierbei korrespondiert die Call-ID dieser SIP-Request INVITE, mit der Call-ID der zu unterbrechenden Sprachdatenübertragung. Beinhaltet eine SIP-Request INVITE eine Call-ID für einen vorgehenden INVITE, so spricht man in diesem Zusammenhang von einem 'Re-INVITE'. Mittels dieser 'Re-INVITE' kann Asterisk die Parameter der laufenden Sprachdatenübertragung in sofern ändern, das für den Teilnehmer der durch die Unterbrechung des Gesprächspartners der neue Gesprächspartner die Telefonanlage Asterisk ist, welche als eine entsprechende Wartemusik als Sprachdatenübertragung versendet.

Nimmt der Gesprächspartner das Gespräch wieder auf, so versendet die Asterisk eine entsprechende SIP-Request INVITE als Re-INVITE, so dass ggf. die Sprachdatenübertragung wieder unmittelbar zwischen den beiden Endgeräten erfolgen kann. Zu beachten ist, dass die Funktionalität 'Re-INVITE' vom jeweiligen Endgerät unterstützt werden muss, was in der weiter oben erwähnten Konfigrationsdatei sip.conf für den jeweiligen SIP-Account konfiguriert werden kann.

Nachdem in Abb. 2.5 dargestelltem Szenario, ist dem Asterisk eine lokale IP-Adresse im privaten Netzwerkadressenbereich entsprechend der Netzwerkkonfiguration zugeteilt, wobei der Standardport für SIP der nach der Definition in RFC 3261 [10] 5060 ist. SIP-fähige Endgeräte können über diese IP-Adresse mit den konfigurierten Benutzerdaten sich über die Asterisk registrieren, der dann die IP-Adresse die für das Endgerät die bei der Registrierung für dieses zugeteilt war, für die Vermittlung von SIP-Signalisierungsdaten verwendet bzw. das SIP-Registrar aktualisiert, wobei die Ziel-

10. RFC 3261 - SIP: Session Initiation Protocol

adressen für die Übertragung der Sprachdaten sich von denen der Signalisierungsdaten unterscheiden kann bzw. beim Gesprächsaufbau im SIP ausgehandelt wird.

Würde man nach dem Szenario in Abb. 2.5 den externen Einsatz von SIP-Clients beabsichtigen, so müsste dem Asterisk eine öffentliche IP-Adresse zugeteilt sein, über die SIP-Clients über eine Internetverbindung kommunizieren könnten. Zu berücksichtigen sind jedoch bei dem externen Einsatz von SIP-Clients abgesehen von Sicherheitsaspekten im Bezug auf den Transport der Signalisierungs- und Sprachdaten über das Internet, unter anderem auch unterschiedliche Hardware-Architekturen sowie lokale Sicherheitsbestimmungen. Hierbei sind Firewalls häufig so konfiguriert, dass der Datenverkehr von UDP [11] Pakten oder eingehenden TCP [12] -Verbindungen verhindert wird, was den externen Einsatz von Web-basierten Softphones im VoIP-Umfeld generell erschwert.

Während die Signalisierungsdaten am Asterisk zentral über den Port 5060 abgehandelt werden, ist beim Einsatz von externen SIP-Clients auch zu berücksichtigen, dass für jede Sprachdatenübertragung mittels dem RTP, die nicht direkt zwischen zwei Endgeräten sondern über die Asterisk geführt wird, der VoIP-Gateway Asterisk für jeden der beiden Gesprächspartner einen separaten bzw. für jedes Gespräch zwei UDP-Ports verwendet, was insbesondere bei einer hohen Anzahl von Gesprächen, bei denen externe SIP-Clients beteiligt sind, zu noch höheren Risiken führt, zumal jeder geöffnete UDP-Port eine potenzielle Angriffsstelle darstellt und das über das Internet transportierte Datenvolumen steigt.

11. User Datagram Protocol (RFC 768), Protokoll zum verbindungslosen Datenaustausch zwischen Rechnersystemen
12. Transmission Control Protocol (RFC 793), Protokoll zum verbindungsorientierten Datenaustausch zwischen Rechnersystemen

## 2.3 HOB WebSecureProxy (WSP)

Die Software-Lösungsfamilie HOB RD VPN der Fa. HOB GmbH & Co. KG bietet Zielsystem-unabhängig Zugriff auf alle Anwendungen und Daten im Firmennetz. Der HOB WebSecureProxy als Produkt der Lösungsfamilie HOB RD VPN ermöglicht hierbei für Clientprodukte den Datenaustausch mit firmeninternen Servern. Ziel ist es, über eine SSL-Verbindung den sicheren Zugriff auf interne Applikationsserver zu gewährleisten.

**Abbildung 2.6:** Web-Portal HOB RD VPN

Abb. 2.6 stellt das Web-Interface des HOB RD VPN wie es als Web-Portal über einen Browser zur Verfügung gestellt wird. Wie aus Abb. 2.6 ersichtlich, wird der Zugriff auf verschiedene interne Applikationsserver angeboten. Während z.B. über den Link 'Start der Sessions' der Datenaustausch mit firmeninternen über z.B. unterschiedliche Terminalserver-Systeme ermöglicht wird, können über den HOB WSP Web-Server-Gate über die SSL-Verbindung Internetseiten über einen externen Web-Server aufgerufen werden.

Die grundsätzliche Funktionsweise des WSP ist hierbei, eine Datenübertragung aus dem Firmennetz mittels einer SSL-Verbindung über das Internet zu gewährleisten. Die Verwaltung von verschiedenen Applikationen erfolgt hierbei über das Konzept der Server-Data-Hook. Ein Server-Data-Hook (kurz SDH), ist eine je nach Bedarf am WSP dynamisch erweiterbare Funktionalität, über welche die erforderliche Verarbeitung von Daten

für unterschiedliche Anforderungen erfolgt. Nachstehende Abbildung soll das allgemeine Prinzip des WSP in Zusammenhang mit dem Konzept der SDH veraunschaulichen.

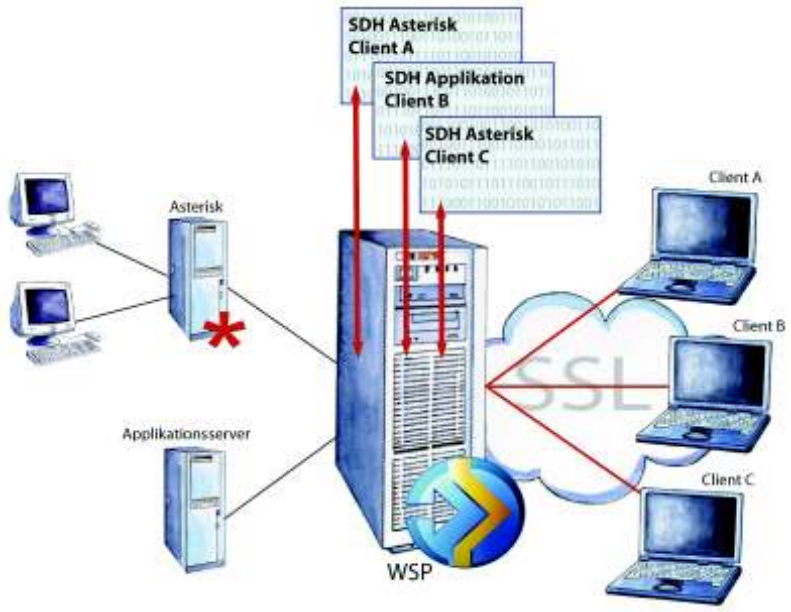


**Abbildung 2.7:** HOB WSP / Server-Data-Hook Konzept

Der Zugriff auf die Funktionalität des WSP über das in Abb. 2.6 dargestellte Web-Portal erfolgt hierbei anhand der Administrierung von Benutzerdaten in einer Konfigurationsdatei am WSP und einem entsprechenden Anmeldedialog des Web-Portals.

Der WSP ist für den Datenaustausch mit einer Clientanwendung als erste und einzige Instanz zuständig und ist eine in den Programmiersprachen C und C++ implementierte Software, welche dynamisch je nach Bedarf SDHs als DLL lädt und die für die Kommunikation zwischen der Clientanwendung und der internen Applikation erforderlichen Vorgänge für die verschiedensten Applikationen, in den SDH 'auslagert'. Dabei werden für die verschiedensten Anwendungen und mit diesen korrespondierten Protokolle SDH´s implementiert und dem WSP als DLL zur Verfügung gestellt.

In Abb. 2.7 werden verschiedene Möglichkeiten für einen Datenaustausch zwischen Clientanwendungen mit firmeninternen Applikationen bzw. Applikationsservern dargestellt. Jeder der Clientanwendung übermittelt hierbei mit dem Aufbau der SSL-Verbindung (vgl. Kap. 2.1.4 'Secure Sockets Layer (SSL)') am WSP erforderliche Parameter im Bezug auf die korrespondierte Anwendung. Anhand dieser Parameter, lädt der WSP dynamisch einen entsprechenden SDH für jeden Client, der die Kommunikation der Clientanwendung mit der firmeninternen Applikation gewährleisten soll.

Ein SDH kann dabei je nach dem für die Kommunikation mit der angesteuerten Applikation sich ergebende Situation Daten umleiten, manipulieren oder auch generieren. Allgemein werden durch den WSP Daten versendet und empfangen, wobei diese durch den korrespondierten SDH in ihrer Beschaffenheit und Informationsrichtung gesteuert werden.

Ausschlaggebend für die Konzeption des WSP ist vor allem die effiziente Verarbeitung der erforderlichen Vorgänge. In diesem Zusammenhang, soll die Kommunikation zwischen dem WSP und dem SDH näher betrachtet werden. Für den Zugriff auf die Funktionalität eines SDH ist zunächst eine entsprechende Konfiguration am WSP erforderlich. Hierzu wird am WSP eine Konfigurationsdatei verwaltet, welche die notwendigen Informationen im Bezug auf den korrespondierten SDH, anhand der von der Clientanwendung übermittelten Parameter ermittelt. Ist die SSL-Verbindung aufgebaut und der korrespondierte SDH geladen, werden sämtliche Daten die mit der Clientanwendung korrespondieren bei Empfang dem SDH übergeben. Der SDH ist hierbei für die Gewährleistung der Kommunikation zwischen Clientanwendung und der firmeninternen Applikation zuständig und wird beim Empfang von Daten unabhängig von der Informationsrichtung mit den empfangenen Daten über eine Funktion gerufen.

Hierbei wird dem SDH für die Dauer der Verarbeitung der empfangenen Daten ein temporärer Speicherbereich seitens dem WSP bereitgestellt. Ist die Verarbeitung der übergebenen Daten am SDH erfolgt, so wird durch den SDH eine Funktion des WSP gerufen, die je nach Situation und Implementierung des SDH variieren kann. Dabei kann der WSP die empfangenen Daten manipulieren oder unverändert weiterleiten, auf neue Daten warten, die empfangenen Daten verändert zurückschicken, neu generierte Daten weiterversenden, eine DNS-Abfrage starten usw.. Das mögliche Verhalten des WSP variiert in diesem Zusammenhang anhand übergebener Daten vom SDH entsprechend der korrespondierten Applikation und den damit zusammenhängenden Protokollen und Standards, wobei die Logik für die erforderlichen situativen Maßnahmen ganz der Implementierung des korrespondierten SDH überlassen wird.

Kern des VoIP-Konzeptes für SSL-geschützte Internettelefonie ist es, die Funktionalität des HOB WSP um VoIP zu erweitern, was in folgenden Kapiteln im einzelnen dargestellt wird.

# Kapitel 3

# Realisierung

## 3.1 Architektur des Konzeptes

In diesem Kapitel sollen die Komponenten des VoIP-Konzeptes für SSL-geschützte Internettelefonie besonders im Hinblick auf die Verarbeitung von Signalisierungs- und Sprachdaten näher beschrieben, sowie eine allgemeine Definition der Architektur der zu implementierenden Komponenten im vorgesehenen VoIP-Szenario erstellt werden. Grundlage für die weiteren Betrachtungen ist das in Kap. 2.3 dargestellte Konzept des HOB WSP und der in Kap. 2.2 dargestellte VoIP-Gateway Asterisk.

### 3.1.1 Definition der Komponenten

Ausgehend von den in Kap. 2.2 näher beschriebenen Problemstellungen unter anderem im Bezug auf Kompatibilität, Wirtschaftlichkeit und Flexibilität beim Einsatz externer SIP-Clients im VoIP-Szenario mit der Telefonanlage Asterisk, soll zunächst ein extern einsetzbarer SIP-Client implementiert werden der den erwähnten Problemstellungen einen Lösungsansatz bietet. Die wesentlichen Hauptmerkmale zum Einsatz des im weiteren Verlauf als HOBPhone benannten SIP-Clients werden wie folgt definiert:

- Plattformunabhängigkeit
- Standortunabhängigkeit
- Kompatibilität
- Sicherheit im Datenverkehr
- Unabhängigkeit von lokalen Sicherheitsbestimmungen
- $n$ SIP-Accounts bei $n$ Telefonanlagen (Asterisk)
- Sprachqualität

- Ausfallsicherheit
- Benutzerfreundlichkeit

Für die Implementierung des HOBPhone wird die Objektorientierte Programmiersprache Java als Komponente der Java-Technologie von Sun Microsystems zugrunde gelegt, wobei das HOBPhone letztendlich als Java-Applet eingesetzt werden soll. Die Projektsprache wird festgelegt in 'Englisch', somit erfolgen die Dokumentation und Kommentierung der Implementierung als auch die Implementierung der Benutzerschnittstelle in englischer Sprache. Als lokale Voraussetzungen für den Einsatz des HOBPhone als Softphone werden nachstehende Merkmale definiert:

- Internetanschluss (minimal 64kbs Upstream)
- Java virtual machine (kurz JVM)
- Soundkarte
- Lautsprecher und Mikrofon

Mit der Zielsetzung, lokale Voraussetzungen für den Einsatz des SIP-Client als Java-Applet zu minimieren und das Konzept ohne Abhängigkeiten zu 'Fremd-'Implementierungen und den damit verbundenen Urheberrechten zu realisieren, wird hierzu weiterhin grundlegend festgelegt, dass sämtliche Implementierungen für die Verarbeitung der Signalisierungs- und Sprachdaten als SIP- und RTP-Datenpakete ausschließlich ohne Verwendung von bereits vorhandenen Implementierungen erfolgen. Zu diesen gehören z.B. das Package `javax.sip` oder `javax.media.rtp` aus dem Java Media Framework (kurz JMF), zumal der Einsatz des JMF die lokale Voraussetzung einer entsprechenden Installation des JMF mit sich bringen würde.

Ein Kernbestandteil des Konzeptes ist es, die Funktionalität des HOBPhone, als Java-Applet über eine SSL-Verbindung VoIP Gespräche auf Basis der in Kap. 2.1 beschriebenen Standards SIP und RTP unter Berücksichtigung der weiteren Hauptmerkmale Sprachqualität, Ausfallsicherheit und Benutzerfreundlichkeit kompatibel zu einem Einsatz in einer VoIP-Architektur mit der Telefonanlage Asterisk zu ermöglichen. Zusätzlich soll die Möglichkeit gegeben sein, das HOBPhone firmenintern bzw. innerhalb eines Intranet durch Datenübertragung mittels UDP einzusetzen, wobei in diesem Kapitel auf die Besonderheiten im Bezug auf die SSL-Funktionalität als Kernbestandteil des Konzeptes eingegangen wird. Als Ausgangsszenario wird die in Kap. 2.2 'Die Telefonanlage Asterisk' in Abb. 2.5 dargestellte VoIP-Architektur angenommen.

Die Komponenten im Ausgangsszenario sind IP-Telefone als Soft- oder Hardphones und die Telefonanlage Asterisk. Mit Kompatibilität zum Ausgangsszenario ist in diesem Zusammenhang gemeint, dass das Konzept externen SIP-Clients über eine SSL-Verbindung

VoIP-Datentransfer zu ermöglichen, ohne eine Implementierung bzw. Änderung an vorhandenen Komponenten im Ausgangsszenario vornehmen zu müssen, eingesetzt werden kann.

Die Möglichkeit, das Konzept ohne eine Implementierung bzw. Änderung an vorhandenen VoIP-Komponenten einsetzen zu können, bedeutet auch, dass in einer VoIP-Architektur, in der das Konzept eingesetzt wird, andere Komponenten schlichtweg nicht betroffen sind und somit weiter verwendet oder auch weitere Komponenten neu hinzugefügt werden können, sofern diese im Rahmen der SIP- und RTP-Standards eingesetzt werden. Diese können IP-Telefone im Netz sein, ein SIP-Proxy oder die Telefonanlage Asterisk, welche auch die Hauptkomponente im Ausgangsszenario darstellt. Erreicht werden soll dies durch die Erweiterung des in Kap. 2.3 dargestellten WSP um die Funktionalität von VoIP, woraus sich das in Abb. 3.1 dargestellte Szenario ergibt.

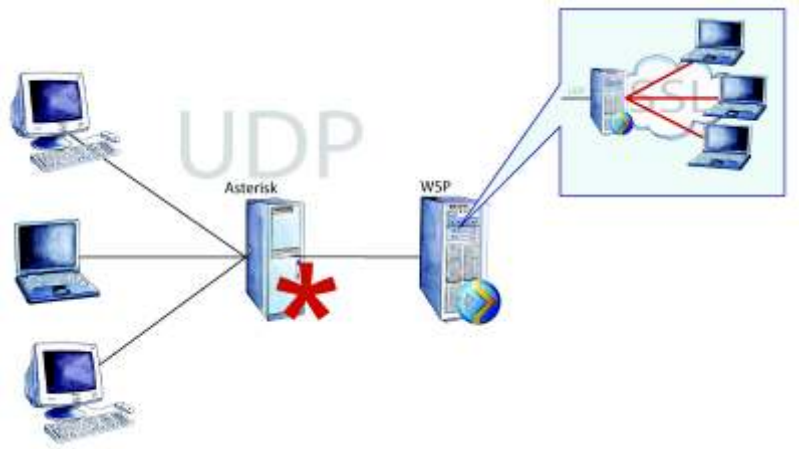


**Abbildung 3.1:** VoIP-Architektur mit WSP

Die Hauptaufgabe des WSP in der vorgesehenen VoIP-Architektur ist es, die Funktionalität des Zugriffs auf interne Applikationsserver von extern auf das VoIP-Szenario zu übertragen. Somit stellt der Asterisk den Applikationsserver dar, auf den der Zugriff von extern, also durch das HOBPhone als Java-Applet ermöglicht werden soll. In Abb. 3.2 ist der Anmelde-Dialog des in Kap. 2.3 dargestellten WSP-Konzeptes abgebildet, wie er um die VoIP-Funktionalität nach erfolgter Anmeldung erweitert ist.

Das HOBPhone agiert hierbei zwar als SIP-Phone am Asterisk, ohne jedoch direkt mit diesem zu kommunizieren, sondern über die sichere SSL-Verbindung mit dem WSP, so dass der externe Datenverkehr über die SSL-Verbindung verläuft und die interne Kommunikation zwischen WSP und Asterisk über UDP abgewickelt wird. Anbetrachts dessen, dass der VoIP-Gateway Asterisk in der momentanen Version SSL nicht unterstützt, wird somit nicht nur die Funktionalität auf Applikationsserver von extern zuzugreifen

**Abbildung 3.2:** Anmeldedialog WSP mit VoIP

auf VoIP übertragen, sondern auch die Zielsetzung von Kompatibilität und Integrierbarkeit in vorhandene VoIP-Architekturen dadurch erfüllt, dass unter Einsatz von SSL (vgl. Kap.2.1.4 'Secure Sockets Layer (SSL)') eine sichere Kommunikation mit Asterisk ermöglicht wird, ohne eine Konfiguration, Implementierung oder Änderung am Asterisk vornehmen zu müssen.

Dadurch soll eine vollständige Unabhängigkeit von Lösungsansätzen für Sicherheit im VoIP-Datenverkehr im Bezug auf deren Kompatibilität und Integrierbarkeit auf vorhandene Ressourcen erreicht werden. Anhand der Voraussetzungen die sich aus den definierten Hauptmerkmalen für den Einsatz des HOBPhone und den jeweiligen Zielsetzungen in der VoIP-Architektur für das Konzept ergeben, stellt neben dem HOBPhone die VoIP-Komponente am WSP den zweiten Baustein des VoIP-Konzeptes für SSL-geschützte Internettelefonie dar.

Die VoIP-Komponente am WSP wird durch das Konzept des Server-Data-Hook (kurz SDH), dass im Zusammenhang mit dem HOB WebSecureProxy in Kap. 2.3 dargestellt wurde, über einen speziellen SDH realisiert. Für die Implementierung des SDH sind hierzu weitere Definitionen und Zielsetzungen erforderlich. Die ermittelte Ausgangssituation legt fest, dass der SDH über UDP intern mit dem Asterisk und extern über die SSL-Verbindung mit dem HOBPhone als Schnittstelle zwischen dem HOBPhone und Asterisk agiert. Die Kommunikation zwischen dem HOBPhone und Asterisk wird, ohne dass etwaige Komponenten in der VoIP-Architektur von dieser Funktionalität betroffen sind, gewährleistet.

Anhand dieser Ausgangssituation ergeben sich auch die wesentlichen Merkmale für den SDH als Komponente in dem beschriebenen Szenario in sofern, dass der SDH die Merkmale und Funktionalität des HOBPhone in der Kommunikation mit dem Asterisk widerzuspiegeln hat, d.h. die Unabhängigkeit des Konzeptes von gegebenen Parametern als Komponenten einer VoIP-Architektur mit Asterisk ist dann gewährleistet, wenn die Verarbeitung der Daten am WSP und die entsprechende Kommunikation mit Asterisk, so erfolgt, als würde die Komponente WSP nicht existieren und Asterisk unmittelbar mit einem klassischen SIP-Phone über UDP kommuniziert (vgl. Abb. 3.1).

Zu beachten ist hierbei, dass die Konzeption des WSP nicht vergleichbar mit der eines SIP-Proxy ist (vgl. Kap. 2.1.1.1 'SIP-Komponenten'). Der WSP ist für beteiligte VoIP-Komponenten zwar für die Signalisierungs- und Sprachdatenübertragung eine terminierende Einheit, jedoch ist diese Funktionalität im Gegensatz zu einem SIP-Proxy als SIP-Komponente nicht vom WSP selbst ausgehend, sondern wird durch ein im SSL-Modus für diese Funktionalität modifiziertes Verhalten des jew. HOBPhone erlangt, was im weiteren Verlauf im einzelnen dargestellt wird (vgl. Abb. 3.6).

Die Übertragung der Rolle des HOBPhone auf den WSP in dieser Konstellation bedeutet auch, dass die Weiterleitung der Datenpakete bei der Kommunikation des WSP mit dem Asterisk unabhängig vom Protokoll (z.B. SIP/SDP bzw. Signalisierungsdaten) oder einer Zeitkritischen Situation (Weiterleitung von RTP-Paketen bzw. Sprachdaten), identisch und somit auch im gleichen Zeitfenster zu erfolgen hat, als würde Asterisk unmittelbar mit einem SIP-Client bzw. dem HOBPhone kommunizieren. Anbetrachts dessen, dass der SDH in diesem Szenario den Datenverkehr von bis zu $n$ HOBPhones verarbeitet, konzentrieren sich somit die Voraussetzungen und entspr. Kapazitäten von $n$ Clients auf den SDH. Dies erfordert für das Verhalten des SDH, dass die Verarbeitung des Datenverkehrs unabhängig vom Protokoll möglichst ohne Zeitverzögerung und sehr effizient zu erfolgen hat.

Um diese Zielsetzung zu erreichen wird festgelegt, dass die Verarbeitung im Sinne von Weiterleitung der Daten, ohne einen zusätzlichen Aufwand zu erfolgen hat, d.h. eingehende Daten vom HOBPhone über SSL müssen an die jeweilige VoIP-Komponente (z.B. Asterisk) und eingehende Daten von anderen VoIP-Komponenten über UDP an das HOBPhone an welches diese gerichtet sind weitergeleitet werden, möglichst ohne eine Interpretation und ausschließlich ohne Modifikation der zu verarbeitenden Datenpakete. Aus der Festlegung, den Datenverkehr zwischen HOBPhone und Asterisk möglichst ohne Interpretation und ausschließlich ohne Modifikation der Datenpakete zu verarbeiten, ergeben sich für das Verhalten der Komponenten HOBPhone und den SDH in verschiedenen Situationen weitere Voraussetzungen.

Um die erforderlichen Maßnahmen zu definieren ist hierzu eine Analyse der Datenpakete der Signalisierungs und Sprachdaten und deren situative Bedeutung in Zusammenhang mit der VoIP-Architektur erforderlich. Ausgehend von der Definition für die SIP-Phone-Komponente HOBPhone, wird auf die beiden einzusetzenden Protokolle näher einge-

gangen, SIP (ggf. in Kombination mit SDP) für Signalisierungsdaten (vgl. Kap. 2.1.1) und RTP für Sprachdaten, (vgl. Kap. 2.1.3). Die Verarbeitung der Datenpakete ist grundsätzlich unabhängig vom Protokoll zunächst in der Informationsrichtung zu Unterscheiden, in Daten vom HOBPhone mit Ziel in die VoIP-Architektur und umgekehrt je zwischen den Beteiligten HOBPhone, WSP und Asterisk (vgl. Abb. 3.3).

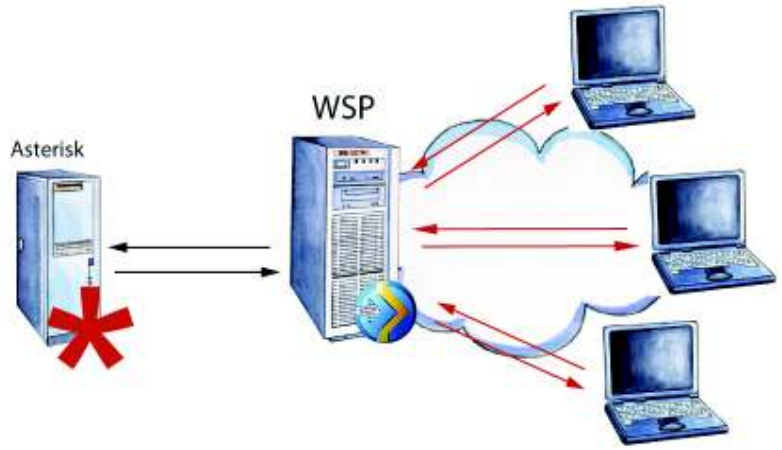


**Abbildung 3.3:** Informationsrichtungen von Daten

Zur Analyse der Begebenheiten im Bezug auf die Weiterleitung der Daten ist zunächst eine Betrachtung über die gegebenen Informationen in einem Datenpaket erforderlich. Diese sind unabhängig vom Protokoll durch die gegebenen Informationen im IP-, UDP- und TCP-Header immer Ziel- und Quell-IP-Adressen sowie Ziel- und Quell-Portnummern.

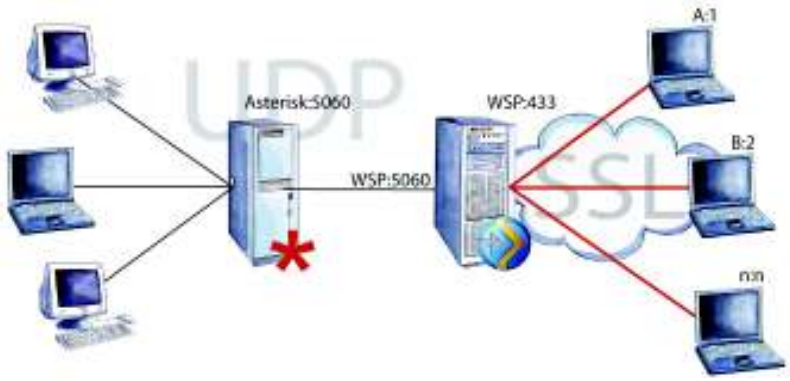


**Abbildung 3.4:** SIP-Datenverkehr mit WSP

Zunächst soll die Verarbeitung von SIP-Datenpaketen vom HOBPhone zur VoIP-Architektur bzw. zur Telefonanlage Asterisk näher betrachtet werden. In den weiteren Ausführungen werden hierzu die in Abb. 3.4 dargestellten IP-Adressen und Port-Nummern der

beteiligten Komponenten verwendet. Ein SIP-Paket hätte demnach die Quell-IP und Port-Konstellation in der Notation 'IP:Port' von $n : n$ und als Ziel WSP:443. Aus diesem Beispiel ist ersichtlich, dass ohne eine Interpretation der Daten der Anwendungsschicht, also der Informationen im SIP-Header, oder eine andere Festlegung für die Weiterleitung, noch nicht ersichtlich ist, wohin das Paket am WSP weitergeleitet werden soll.

Demzufolge wird zunächst festgelegt, dass die SIP-Kommunikation zwischen den SIP-Clients und der VoIP-Architektur ausschließlich über eine Telefonanlage Asterisk zu erfolgen hat, was, wie im weiteren Verlauf bei der Verarbeitung von Sprachdaten dargestellt wird, sich bei RTP-Paketen grundsätzlich unterscheiden kann. Zu beachten ist, das jegliche Festlegung am Verhalten der Komponenten stets konform zu den beteiligten Standards zu erfolgen hat. In diesem Zusammenhang steht auch die Festlegung dass die Kommunikation zwischen SIP-Clients und der VoIP-Architektur stets über die Telefonanlage Asterisk zu erfolgen hat, in keinem Widerspruch zu dem in Kap. 2.1.1 dargestellten Standard SIP. Anhand dieser Festlegung wird am SDH jedoch immer noch die Information benötigt, ob es sich zum einen überhaupt um ein SIP-Paket handelt und wenn ja, ist die Information 'IP:Port' des Asterisk erforderlich, an welche das SIP-Paket gerichtet ist bzw. weitergeleitet werden muss. Diesbezüglich ist sind die Konfigurationsmöglichkeiten des WSP zu den SDH näher zu betrachten.

Der SDH kann bestimmte Informationen vom WSP über eine xml-Konfigurationsdatei erhalten. Diese werden nach Aufbau der SSL-Verbindung, der für jede Verbindung erzeugten jeweiligen Instanz des SDH übergeben. Die für die SSL-Verbindung mit dem WSP benötigten Benutzerdaten werden hierbei über den in Abb. 3.2 dargestellten Anmeldedialog vom Benutzer übermittelt und der Webseite auf die das HOBPhone als Java-Applet Integriert ist als Applet-Parameter übergeben. Wird das Applet gestartet, kann die SSL-Verbindung für das HOBPhone mit dem WSP mittels den über die HTML-Seite übergebenen Parametern Benutzername, Passwort und der URL des WSP, die SSL-Verbindung mit dem WSP aufgebaut werden. Im Bezug zu den benötigten Informationen zur Weiterleitung der SIP-Pakete am WSP wird der Abschnitt <User> in der WSP-Konfigurationsdatei um nachstehende Zeilen ergänzt:

**Listing 3.1:** WSPConf.xml

```
<SIP>
  <ineta>172.22.20.194</ineta>
  <ident>22</ident>
  <shared-secret>22</shared-secret>
</SIP>
```

### 3.1.2 Channel-Mechanismus

Durch die Administration der SIP-Accounts am WSP hat der SDH nach dem SSL-Verbindungsaufbau die Information, dass für den Client mit dem die SSL-Verbindung

soeben aufgebaut wurde, ein Benutzerkonto an einem Asterisk im Firmennetz mit den gegebenen Daten für die IP-Adresse des Asterisk und den Benutzerinformationen existiert.

Somit ist zwar die IP-Adresse für den Empfänger von SIP-Pakten die vom HOBPhone versendet werden am SDH bekannt, jedoch fehlt noch die Unterscheidung des Protokolls in der Anwendungsschicht, d.h. die Interpretation des Paketes als SIP-Paket an sich. Da sich diese Problematik auch bei RTP-Paketen bzw. unabhängig vom Protokoll ergeben würde, wird für die Datenpakete in der Kommunikation zwischen HOBPhone und dem WSP ein eigener HOB-Header definiert. Der HOB-Header soll anhand eines Channel-Mechanismus eine Übertragung von Signalisierungs- und Sprachdatenpaketen über eine vorgesehene SSL-Verbindung ohne eine Interpretation oder Manipulation der übertragenen Roh-Daten ermöglichen, so dass die Datenpakete den entsprechenden Kategorien als Signalisierungs- oder Sprachdaten anhand der Channel-Nummer zugeordnet werden. Demzufolge wird jedes zu übertragende Datenpaket zwischen dem HOBPhone und dem SDH unabhängig von der Informationsrichtung um den HOB-Header erweitert.

Der SDH ordnet den Informationen eine bestimmte Channel-Nummer zu, im weiteren Verlauf des Beispiels sei für SIP-Datenpakete die Channel-Nummer 1. SIP-Datenpakete, die das HOBPhone an die für diese Channel-Nummer definierte Asterisk versenden möchte, müssen nun durch das HOBPhone mit dem HOB-Header versehen werden, der die Channel-Nummer 1 beinhaltet. Hierzu ist wiederum erforderlich, dass das HOB-Phone über die Zuordnung in Kenntnis gesetzt wird, welche Channel-Nummer bei welchem Paket zu verwenden ist, noch bevor überhaupt eine Kommunikation mit der VoIP-Architektur möglich ist bzw. seitens des HOBPhone in Erwägung gezogen werden kann. Die weiter oben dargestellten Zeilen, um welche die xml-Konfiguration des WSP ergänzt wird, beinhalten nun nicht nur die IP-Adresse des korrespondierten Asterisk, sondern auch die Benutzerdaten des SIP-Accounts. Um überhaupt eine Kommunikation beginnen zu können, muss nach diesem Szenario das HOBPhone zunächst die SSL-Verbindung aufbauen, was bei Aufruf des Applet nach dem in Abb. 3.2 dargestellten Anmeldedialog am WSP erfolgt und wodurch der SDH die erwähnten Informationen in der xml-Konfiguration für das jew. HOBPhone bereits hat.

Ausgehend von der SIP-Architektur wie sie in Kap. 2.1.1 dargestellt wurde, dass eine Teilnahme in diesem VoIP-Szenario mit der Telefonanlage Asterisk erst über die Anmeldung beim Asterisk mit den Benutzerdaten möglich ist und der vorhergehenden Festlegung, dass die Kommunikation von SIP-Clients zur VoIP-Architektur ausschließlich über die Telefonanlage Asterisk erfolgt, wird hier nun nach dem Aufbau der SSL Verbindung zunächst ein Dialog zwischen dem SDH und dem HOBPhone ausgeführt. In diesem Dialog, teilt der SDH dem HOBPhone die für diesen Client am WSP Administrierten Informationen der Einwahldaten am Asterisk und gleichzeitig die mit diesen Einwahldaten am SDH assoziierte Channel-Nummer mit. Somit hat das HOBPhone

unmittelbar nach dem Start des Java-Applet und erfolgter SSL-Verbindung alle notwendigen Informationen, um den SIP-Registrierungsdialog durch SIP-Pakete mit dem HOB-Header, der die Channel-nummer beinhaltet, zu beginnen, der auch je nach individueller Benutzerkonfiguration automatisch erfolgen kann.

Unter den wesentlichen Hauptmerkmalen des HOBPhone wurde weiter oben unter anderem die Funktionalität $n$ SIP-Accounts bei $n$ Telefonanlagen definiert. Hierzu ist zu beachten, dass der definierte Ablauf für die Weiterleitung von SIP-Datenpaketen kommend vom HOBPhone hin zur VoIP-Architektur bzw. Telefonanlage Asterisk bei mehreren Accounts eines HOBPhone an einer Asterisk oder bei mehreren Accounts an verschiedenen im Firmennetz lokalen Asterisk identisch ist und sich lediglich vervielfältigt. Die Konfiguration für ein HOBPhone, welches über mehrere Accounts verfügt, erfordert in der Konfigurationsdatei am WSP lediglich mehrere Einträge nach dem gleichen Prinzip:

**Listing 3.2:** WSPConf.xml -Mulitple entries

```
<SIP>
  <ineta>172.22.20.194</ineta>
  <ident>11</ident>
  <shared-secret>11</shared-secret>
</SIP>
<SIP>
  <ineta>172.22.20.194</ineta>
  <ident>22</ident>
  <shared-secret>22</shared-secret>

</SIP>
<SIP>
  <ineta>172.22.5.111</ineta>
  <ident>11</ident>
  <shared-secret>11</shared-secret>
</SIP>
```

Stellt man die Accounts in der Notation Benutzername@IP-Asterisk dar, würde die obige Konfiguration die Accounts 11@172.22.20.194, 22@172.22.20.194 und 11@172.22.5.111 definieren. Hierzu erhält das HOBPhone nach Aufbau der SSL-Verbindung nacheinander die Informationen für jeden Account mit je einer eigenen Channel-Nummer. Agiert das HOBPhone für den Account 22@172.22.20.194, so versieht es das zu versendende SIP-Paket mit einem HOB-Header mit der vom SDH zugeordneten Channel-Nummer für diesen Account (vgl. Abb. 3.5).

Für die Verarbeitung von eingehenden SIP-Paketen über UDP, ist vorerst noch eine weitergehende Überlegung im Bezug auf das SIP-Protokoll und die generierten SIP-Pakete des HOBPhone erforderlich. Nach den Darstellungen in Kap. 2.1.1 sind die Informationen im SIP-Header ausschlaggebend für den Verlauf eines SIP-Dialoges. Vom Erzeuger

**Abbildung 3.5:** Channel-Mechanismus

einer SIP-Request wird z.B. im 'Via'-Feld angegeben, an welche IP-Adresse und Port eine entsprechende SIP-Response erwartet wird. Ausgehend vom weiter oben festgelegten Verhalten des SDH, eingehende SIP-Datenpakete vom HOBPhone ohne eine Modifikation weiterzuleiten, sind demnach eingehende SIP-Datenpakete für das HOBPhone auf die IP-Adresse und Port zu erwarten, die im 'Via'-Feld der ursprünglichen Nachricht stand. Würde somit das HOBPhone seine tatsächliche IP-Adresse und Port im 'Via'-Feld hinterlegen, kämen die erwarteten SIP-Responses nicht an, da diese IP-Adresse und Port die SSL-Verbindung mit dem WSP darstellt (vgl. Abb. 3.6).

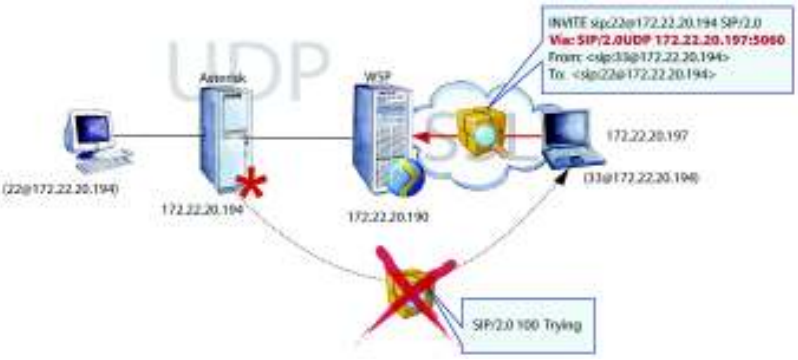


**Abbildung 3.6:** Fehlinformationen im SIP-Header

Da am SDH keine Modifikationen von Datenpaketen vorgenommen werden sollen, bedeutet dies für HOBPhone, die SIP-Header so zu erzeugen, dass diese ein SIP-Phone simulieren, welches im Firmennetz direkt mit dem Asterisk kommuniziert. Für alle andere Beteiligten VoIP-Komponenten ist dadurch lediglich die IP-Adresse und Portnummer des WSP auf UDP-Seite bekannt, was dem in Abb. 3.1 dargestelltem Szenario entspricht.

Ähnlich wie bei der Channel-Nummer für einen SIP-Account, wird hier wiederum am HOBPhone, noch bevor eine SIP-Kommunikation stattfinden kann, die Information benötigt, über welche interne IP-Adresse und Portnummer der WSP erwartet, SIP-Pakete zu empfangen, damit die zu generierenden SIP-Header entsprechend 'manipuliert' werden können. Dabei wird am WSP die Portnummer für die Verarbeitung von SIP-Datenpaketen mittels UPD die für SIP als Standard definierte Portnummer 5060 festgelegt. Ferner gelten bei SIP-Datenpaketen die einen SDP-Content beinhalten die gleichen Bedingungen. Hierbei ist zu unterscheiden, dass im SDP-Teil einer SIP-Nachricht nicht die Parameter für den weiteren Verlauf des SIP-Dialoges, sondern die für eine bevorstehende Sprachdatenübertragung mittels dem RTP definiert werden, wobei sich die erforderlichen Informationen der RTP-Datenverarbeitung von dem Szenario im Bezug auf SIP-Datenpakete unterscheiden. Hierzu ist allerdings eine nähere Betrachtung der RTP-Datenverarbeitung notwendig, welche im späteren Verlauf bei der Verarbeitung von RTP-Datenpaketen behandelt wird.

Während bei der SSL-Verbindung für jedes HOBPhone eine Instanz des SDH zugeordnet werden kann (siehe Kap. 2.3 'HOB WebSecureProxy (WSP)'), haben SIP-Pakete ankommend vom Asterisk über UDP, die an ein bestimmtes HOBPhone gerichtet sind, anders als z.B. bei RTP-Paketen wie im späteren Verlauf dargestellt wird, lediglich eine Quell-IP-Adresse, die Asterisk IP-Adresse und eine Quell-Portnummer, den Standardport 5060. Eingehende SIP-Pakete können allein aus den Header-Daten der IP- und UDP-Protokolle nicht zu einem SDH bzw. HOBPhone zugeordnet werden, was eine Interpretation des SIP-Headers auf Anwendungsschicht des Datenpaketes unumgänglich macht. Hierzu wird der für die Ermittlung des Korrespondierten HOBPhone erforderliche Algorithmus so einfach wie möglich gehalten, so dass so wenig wie möglich Informationen aus den eingehenden SIP-Datenpaketen Interpretiert werden müssen.

Jedoch ist vor der Verarbeitung des Datenpaketes die grundsätzliche Zuordnung in SIP- oder RTP-Datenpakete erforderlich. Zu dieser grundsätzlichen Zuordnung ist keine Interpretation der Daten auf Anwendungsschicht erforderlich, da sich diese aus den Quellinformationen der IP- und UDP-Header in Form von Quell-IP-Adresse und Quell-Portnummer zuordnen lassen. Datenpakete die von einer Quell-IP-Adresse stammen, welche bereits am WSP bei einem SIP-Account als SIP-Gateway eingetragen sind und die Standard-Portnummer 5060 für SIP als Quell-Portnummer haben, können somit eindeutig als SIP-Datenpakete zugeordnet werden. Ist das Datenpaket einmal eindeutig als SIP-Datenpaket zugeordnet, kann durch die in Kap. 2.1.1 beschriebene Funktionsweise des SIP anhand des nachstehend näher beschreibenen Algorithmus das Korrespondierte HOBPhone als Ziel ermittelt werden.

Demnach ist zunächst eine Unterscheidung des SIP-Datenpaketes als SIP-Response oder SIP-Request erforderlich. Bei einer SIP-Response handelt es sich um eine Antwort auf einen vorher erzeugten Request, d.h. der Erzeuger des Dialoges, auf die sich diese SIP-Response bezieht, war in diesem Szenario ein HOBPhone. Im Gegensatz zu einem

SIP-Request, beinhaltet eine SIP-Response im 'From'-Teil des SIP-Headers die Information des Erzeugers des ursprünglichen SIP-Request´s und die des Erzeugers der SIP-Nachricht selbst. Da eine SIP-Response immer als Antwort immer an den Erzeuger der ursprünglichen Nachricht gerichtet ist (sofern nichts anderes angeben wurde), ist somit die Information bei einer SIP-Response im 'From'-Teil ausschlaggebend dafür, welches der HOBPhone auf SSL-Seite korrespondiert ist. Wurde ein SIP-Request empfangen, beinhaltet der Parameter 'To', die notwendige Information für die Weiterleitung des Datenpaketes, wobei der Tag 'To' im Gegensatz zur Konstellation bei einer SIP-Response, wörtlich auch im Einklang zur Informationsrichtung steht. Unabhängig davon, ob die benötigten Daten im 'From' oder 'To' Teil einer SIP-Response oder SIP-Request stehen, haben diese in entsprechender Konstellation den gleichen Informationsgehalt in der Notation benutzername@asterisk.

Anhand der Administration der Benutzerdaten am WSP für SIP-Accounts, kann nun das korrespondierte HOBPhone am WSP zusammen mit der dazugehörigen Channel-Nummer ermittelt werden, so dass das Datenpaket zur Verarbeitung an die entsprechende SDH-Instanz übergeben wird. Der SDH vervollständigt dann das SIP-Datenpaket mit dem HOB-Header mit entsprechender der Channel-Nummer und leitet das Datenpaket über die SSL-Verbindung an das HOBPhone weiter, was in Abb. 3.7 dargestellt wird.

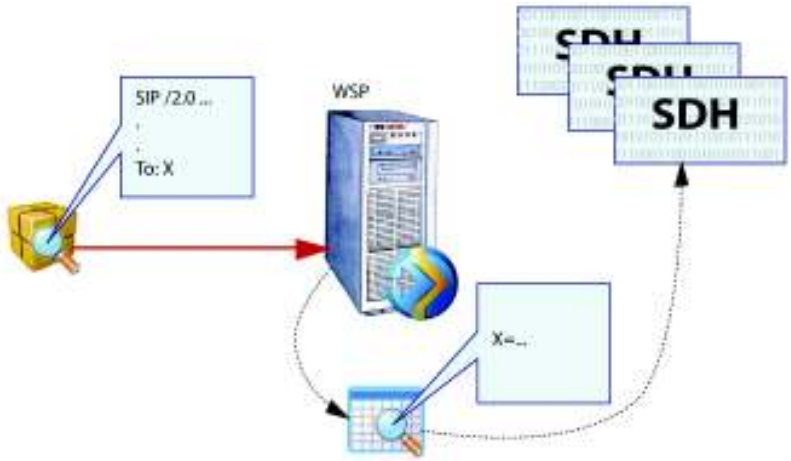


**Abbildung 3.7:** Aufruf eines SDH

Während für die Verarbeitung der Signalisierungsdaten festgelegt werden konnte, dass die SIP-Kommunikation zwischen den SIP-Clients und der VoIP-Architektur ausschließlich über eine Telefonanlage Asterisk zu erfolgen hat, ist bei der Verarbeitung der Sprachdaten bzw. von RTP-Paketen zu unterscheiden, dass sich das Ziel ausgehend vom HOBPhone, abhängig von den im SDP-Teil ausgehandelten Parametern beim Gesprächsaufbau, dynamisch für jede Sprachdatenübertragung und auch für eine aktive Sprachdatenübertragung ändern kann (vgl. Kap. 2.3 'Die Telefonanlage Asterisk').

Konnte demnach am WSP anhand der Administration der SIP-Accounts über die Konfigurationsdatei festgelegt werden, woher mögliche SIP-Datenpakete stammen können bzw. hin zu versenden sind, ist diese grundsätzliche Konfiguration für RTP-Datenpakete nun nicht möglich. Hierzu werden die erforderlichen Parameter zwischen HOBPhone und WSP bzw. der zugehörigen SDH-Instanz, dynamisch für jeden Gesprächsaufbau und entsprechend der Aushandlung der Parameter im jeweiligen SDP-Teil der SIP-Datenpakete konfiguriert. Die Festlegung, dass die Datenverarbeitung am WSP effizient und somit möglichst ohne eine Interpretation und ausschließlich ohne eine Modifikation der zu verarbeitenden Datenpakete zu erfolgen hat, trägt bei der Verarbeitung der RTP-Datenpakete aufgrund der zeitkritischen Umgebung bei Echtzeitanwendungen eine besonders hohe Relevanz.

Abgesehen von den in Kap. 2.1.2 beschriebenen auszuhandelnden Parametern bei einem Gesprächsaufbau im SDP-Teil über einen entsprechenden SIP-Dialog, werden im Bezug auf die erwähnte Festlegung zur Interpretation oder Modifikation der zu verarbeitenden Daten, im folgenden ausschließlich die Informationen für IP-Adresse und Portnummer näher betrachtet. In diesem Zusammen wurde hierzu festgestellt, wie der SIP-Header in Kombination mit dem HOB-Header bei SIP-Datenpaketen in der Informationsrichtung vom HOBPhone ausgehend hin zur VoIP-Architektur seitens des HOBPhone zu erstellen ist, damit im weiteren Verlauf der Kommunikation nicht die in Abb. 3.6 dargestellte Problematik auftritt, so dass durch die Eintragung der tatsächlichen IP-Adresse und Portnummer in den SIP-Header durch das HOBPhone andere VoIP-Komponenten vergeblich versuchen würden, über UDP SIP- Datenpakete an die für die SSL-Verbindung mit WSP genutzte IP-Adresse und Portnummer zu versenden.

Zur Festlegung der Vorgehensweise bei der Verarbeitung von RTP-Datenpaketen ist zunächst eine Unterscheidung von ausgehenden und eingehenden Anrufen bzw. eine Unterscheidung im Bezug auf den Erzeuger eines Gesprächsaufbaus gemäß SIP erforderlich. Zu unterscheiden ist hierbei, dass die IP-Adresse und Portnummer für die Übertragung von SIP-Datenpaketen im Gegensatz zu RTP-Datenpaketen mittels UDP, sich nicht individuell nach einem HOBPhone richtet, sondern nach den am WSP Konfigurierten Angaben für die Telefonanlage Asterisk, d.h. dass die erwähnte IP-Adresse und Portnummer auf UDP-Seite bei SIP-Datenpaketen für $n$ HOBPhone verwendet wird (vgl. Abb. 3.4), während bei einer Übertragung von RTP-Datenpaketen für jeden Gesprächsaufbau eine eigene Kombination von IP-Adresse und Portnummer benötigt wird bzw. eine eigene Portnummer bei explizit einer IP-Adresse des WSP für die UDP-Übertragung im Intranet. In diesem Zusammenhang ist zu beachten, dass der WSP auch Multihomed [1] eingesetzt werden kann, wodurch für die Kommunikation mit firmeninternen Applikationsservern unterschiedliche IP-Adressen verwendet werden können.

Während für die Verarbeitung der SIP-Datenpakete die IP-Adresse und Portnummer am WSP für die Übertragung mittels UDP bereits nach dem Aufbau der SSL-Verbindung

1. in diesem Zusammenhang Serversystem mit mehreren NIC(Network Interface Card)

festgelegt sind und dem HOBPhone durch eine entsprechende Instanz des SDH mitgeteilt werden können, sollen aufgrund dessen, dass die Dauer und Anzahl von RTP-Datenübertragungen abhängig vom jeweiligen Anwender des HOBPhone ist, IP-Adresse und Portnummer für die Übertragung von RTP-Datenpaketen mittels UDP, erst bei Bedarf am WSP festgelegt werden. Ausgehend von dieser Definition soll zunächst der Verlauf, in dem das HOBPhone der Erzeuger eines Gesprächsaufbaus nach SIP ist, näher betrachtet werden.

Das HOBPhone benötigt nach diesem Szenario noch bevor der Gesprächsaufbau über SIP stattfindet, die Information, an welche IP-Adresse und Portnummer der Gesprächspartner RTP-Pakete zu versenden hat, bzw. wo der WSP RTP-Datenpakete zur Übergabe an die entsprechende Instanz des SDH erwartet, der dann die RTP-Datenpakete mit einem HOB-Header und somit einer entsprechenden Channel-Nummer an das HOBPhone über SSL-versendet. Hierzu versendet das HOBPhone noch vor einem ausgehenden Gesprächsaufbau nach SIP eine Anfrage an den WSP der diese an den jew. SDH weiterleitet, eine Portnummer für eine bevorstehende Übertragung von RTP-Datenpaketen mittels UDP am WSP zu reservieren, die das HOBPhone im entsprechenden SDP-Teil des SIP-Datenpaketes einsetzen kann. Anders, als bei dem Dialog zur Festlegung der Parameter für die Verarbeitung von SIP-Datenpaketen unmittelbar nach dem SSL-Verbindungsaufbau, definiert hierbei das HOBPhone die für die Anforderung im weiteren Verlauf zu korrespondierende Channel-Nummer, anhand welcher am SDH bei eingehenden Datenpaketen vom HOBPhone entschieden wird, wohin diese mittels UDP weitergeleitet werden sollen.

Ist dem HOBPhone die lokale IP-Adresse und Portnummer für eine UDP-Datenübertragung am WSP bekannt, eine Channel-Nummer für die Übertragung von Daten im Bezug auf diese Portnummer definiert, fehlt noch am WSP bzw. dem jeweiligen SDH der mit dem WSP intern kommuniziert, die Information über das Ziel welches für diese in diesem Szenario temporäre Channel-Nummer korrespondiert werden soll. Die noch benötigte Information über das Ziel für die bevorstehende RTP-Datenübertragung ergibt sich hierbei aus dem im SIP definierten Verlauf für den Gesprächsaufbau ausgehend vom HOBPhone als Erzeuger. Demnach führt die ursprüngliche INVITE-Nachricht des HOBPhone dazu, dass durch die Vermittlung durch die Telefonanlage Astersik, eine Akzeptanz oder Ablehnung des Ersuchens des HOBPhone durch den jeweiligen Gesprächspartner erfolgt bzw. zu erwarten ist, wobei als Ablehnung in diesem Sinne auch eine fehlende Reaktion innerhalb eines definierten Zeitraumes interpretiert wird.

Nimmt nun der Gesprächspartner das Gespräch an, so sind in der für diesen Gesprächsaufbau terminierenden Nachricht die erforderlichen Informationen darüber erhalten, welche IP-Adresse und Portnummer als Gegenstelle für die Übertragung der RTP-Datenpakete mittels UDP zu verwenden sind (siehe auch Kap. 2.1.1.3 'Verbindungsaufbau mit SIP'). Bevor nun das HOBPhone mit der Übertragung der eigenen RTP-Datenpakete oder

dem Empfang von RTP-Datenpaketen der Gegenstelle beginnt und nach SIP eine Bestätigung als SIP-Request 'ACK' für die erwähnte terminierende Nachricht versendet, um beidseitig die Sprachdatenübertragung zu veranlassen, sendet es mit einem weiteren Befehl an den WSP die dort benötigten Informationen für die Verarbeitung der RTP-Datenpakete der bevorstehenden Sprachdatenübertragung, was als Vorgang in Kap. 3.1.3'WSP-Protokoll' im einzelnen näher dargestellt wird.

Empfängt der WSP Datenpakete von dieser IP-Adresse und Portnummer, können nun die Daten anhand der Informationen über IP-Adresse und Portnummer des Versenders mittels UDP durch den UDP- und IP-Header in Zusammenhang mit dem vorgehenden Dialog und dem HOBPhone, dem zugehörigen SDH übergeben werden, der dann ohne jegliche Interpretation oder Modifikation der RTP-Daten den vom HOBPhone erwarteten HOB-Header mit entsprechender Channel-Nummer hinzufügt und die Datenpakete über die SSL-Verbindung zum HOBPhone weiterleitet.

Bei eingehenden Datenpaketen über SSL, kann der SDH anhand der Channel-Nummer die Datenpakete an das jeweilige Ziel mittels UDP weiterleiten bzw. das Versenden über den WSP veranlassen. Die Vorgehensweise für die Verarbeitung der Sprachdatenübertragung am SDH bei einem Gesprächsaufbau ausgehend von einer Komponente in der VoIP-Architektur gestaltet sich nach dem gleichen Prinzip. Wenn der Erzeuger der initiierenden SIP-Nachricht für einen Gesprächsaufbau, der SIP-Request 'INVITE', nicht das HOBPhone ist, sind alle erforderlichen Parameter für einen Gesprächsaufbau durch die SIP-Request bereits vorhanden. Somit werden noch vor dem Status, dem Anwender eine Akzeptanz oder Ablehnung des eingehenden Anrufes überlassen wird, die Parameter im SDP-Teil der eingehenden SIP-Request 'INVITE' mit den lokal Verfügbaren Ressourcen im Hinblick auf zu verwendende Codecs abgeglichen, was in Kap. 3.2.3.1'Verarbeitung von Singalisierungsdaten (SIP,SDP)' im einzelnen dargestellt wird.

### 3.1.3 WSP-Protokoll

Kann auf Basis der Parameter im SDP-Teil der SIP-Request und den lokalen Begebenheiten eine Sprachdatenübertragung begonnen werden, müssen die für die Verarbeitung der RTP-Datenpakete am SDH benötigten Informationen für den durch das HOBPhone zu erstellenden SDP-Teil der terminierenden Nachricht durch aufeinander folgende Dialoge zwischen HOBPhone und WSP in Form von Port-Anforderung und Zielangabe definiert werden. Während der Dialog für die Port-Anforderung definiert, über welche IP-Adresse und Portnummer der WSP bereit ist, Sprachdatenpakete mittels UDP zu Empfangen und zu Versenden, wird durch die Zielangabe, welche durch die eingegangene Ursprüngliche SIP-Request angegeben wurde und mit der Channel-Nummer der Port-Anfrage korrespondiert, am SDH festgelegt, zu welcher Komponente in der VoIP-Architektur die über die SSL-Verbindung vom HOBPhone eingehenden Sprachdatenpakete, mittels UDP weitergeleitet werden müssen.

Hierzu sollen die in ihrer Notation und im Ablauf zu definierenden Dialoge zwischen dem HOBPhone und dem SDH die in Frage kommen näher betrachtet werden. Bei Dialogen zwischen dem HOBPhone und dem SDH geht es hauptsächlich um einen Informationsaustausch zwischen den beiden Komponenten, um bei der Datenübertragung im weiteren Verlauf eine möglichst effiziente Weiterleitung von Datenpaketen am SDH zu ermöglichen, wie weiter oben bereits dargestellt wurde. Für die Interpretation der im weiteren Verlauf als Control-Message benannten Datenpakete, welche in diesem Zusammenhang WSP-Signalisierungsdaten darstellen, wird die Channel-Schlüsselnummer '0' definiert. Wird also ein Datenpaket vom SDH oder einem HOBPhone mit der Channel-Nummer '0' empfangen, so handelt es sich um das im weiteren Verlauf zu definierende Protokoll und somit um einen zwischen dem SDH und dem HOBPhone verlaufenden Dialog, wobei die Verarbeitung von WSP-Signalisierungsdaten in Kap. 3.2.3.2'Verarbeitung von WSP-Signalisierungsdaten' im einzelnen dargestellt wird. Dieses zu definierende Protokoll wird als WSP-Protokoll benannt. Die Notation des WSP-Protokoll wird ähnlich der SIP-Notation festgelegt. Demnach soll das Protokoll in Klartext mittels dem Zeichensatz UTF-8 angewandt werden. Informationseinheiten werden als durch ein Doppelpunkt getrennte Einheiten von Parameter und Wert dargestellt, wobei jede Informationseinheit eine mit Wagenrücklauf und Zeilenvorschub (Carriage Return, Line Feed als \r \n) endende Zeile darstellt. Eine Control-Message gemäß dem WSP-Protokoll, kann aus einer Request oder einer Response bestehen.

Wie bereits weiter oben dargestellt, erfolgt unmittelbar nach dem Aufbau einer SSL-Verbindung und Login am WSP (vgl. Abb. 3.2)ein Dialog, bei dem der SDH mittels der Administration von Asterisk-Accounts am WSP, die Daten für Asterisk-Accounts und evtl. individuelle Benutzereinstellungen wie z.B. Standardaccount, Auto-Login etc. dem HOBPhone zusammen mit einer im späteren Verlauf für SIP-Datenpakete im HOB-Header zu korrespondierenden Channel-Schlüsselnummer als Control-Request 'REGISTER' dem HOBPhone versendet. Nachstehend wird der Aufbau einer solchen Control-Request veranschaulicht.

REGISTER
config-name:test1
channel:1
SIP-Ident:11
INETA-SIP-gateway:172.5.0.1
SIP-shared-secred:geheim
INET A-this:172.5.0.1

An dieser Stelle soll erwähnt werden, dass die Implementierung des SDH in dieser Version auf Basis von ausschließlich einem Asterisk-Account pro HOBPhone und eine Asterisk als VoIP-Komponente in der VoIP-Architektur erfolgt. Die Implementierung des HOBPhone im Bezug auf die Datenübertragung hingegen erfolgt bereits auf Basis der definierten Architektur mit $n$ möglichen Accounts an $n$ verschiedenen Asterisk in der

**Tabelle 3.1:** Control-Request im WSP-Protokoll

| Befehl | Beschreibung |
|---|---|
| CREATE | Channel erzeugen und somit für gültig erklären |
| SET | Channel parametrisieren |
| REMOVE | Channel ungültig erklären, bzw. entfernen |

VoIP-Architektur, was im einzelnen in Kap. 3.2.1'HOBPhone Architektur' dargestellt wird. Die Funktionalität des SDH Datenpakete von mehreren Asterisk-Accounts über mehrere Asterisk in der VoIP-Architektur zu verarbeiten, ist eine Erweiterung des für diese Version implementierten SDH und wird in Kap. 4'Fazit und Ausblick' erwähnt.

Als Request werden allgemein die Methoden CREATE, SET, REMOVE und REGISTER definiert. Der Aufbau einer Control-Request besteht aus dem Methodennamen und mindestens dem Parameter für die korrespondierte Channel-Nummer. Während für die Verarbeitung von SIP-Datenpaketen maßgebliche Informationen durch die Administration der entsprechenden Informationen am WSP durch die Control-Request 'REGISTER' lediglich dem HOBPhone mitgeteilt werden, ergeben sich für die Verarbeitung von RTP-Datenpaketen relevante Informationen, die am WSP benötigt werden dynamisch je nach den Gesprächsverbindungen. Somit wird z.B. am WSP bei einer bevorstehenden Übertragung von RTP-Datenpaketen eines Gesprächs die Information benötigt, welcher IP-Adresse und Portnummer mittels UDP die RTP-Datenpakete des HOBPhone weitergeleitet bzw. das HOBPhone RTP-Datenpakete von dieser erwartet. In Tab. 3.1.3 werden die Möglichkeiten für eine Control-Request im Bezug auf eine bevorstehende RTP-Datenübertragung dargestellt.

Die sich aus der VoIP-Architektur ergebende Situation, in welcher ein Dialog zum erzeugen und parametrisieren einer Channel-Nummer im Bezug auf eine RTP-Datenübertragung erforderlich ist bzw. stattfindet, wird in Zusammenhang mit der Steuerung von Audio-Komponenten des HOBPhone in Kap. 3.2.4.1'Steuerung der Audio-Komponenten' im einzelnen dargestellt. Der Dialog bei dem vom HOBPhone ausgehenden Portanfrage und der empfangen Antwort vom SDH lässt sich mittels des WSP-Protokoll wie folgt darstellen:

**Listing 3.3:** WSP-Dialog CREATE

```
CREATE\\
channel:2

200 OK\\
channel:2\\
INETA/local:172.5.0.1\\
port/local:15123
```

Nachdem das HOBPhone mit der Control-Response die Information, welche Portnummer am WSP für das HOBPhone für die bevorstehende RTP-Datenübertragung mittels UDP reserviert wurde erhält, erfolgt der Dialog zur Parametrisierung der korrespondierten Channel-Nummer, der im Aktivitätsdiagramm als 'Zielangaben' und 'Gesetzt' veranschaulicht wird und nachstehend entsprechend dem WSP-Protokoll dargestellt wird.

**Listing 3.4:** WSP-Dialog SET

```
HOBPhone --> WSP:
SET
channel:2
INETA/target:172.5.0.9
port/target:15987

WSP --> HOBPhone:
200 OK
channel:2
INETA/target:172.5.0.9
port/target:15987
```

Control-Responses bestehen aus einem Response-Code mit einem korrespondierenden Response-Text und identisch zur Control-Request mit mindestens dem Parameter für die korrespondierte Channel-Nummer. Für den durch das WSP-Protokoll zu steuernden Dialog, wird hierbei vorausgesetzt, dass zwischen einem HOBPhone und dem SDH immer nur explizit ein Dialog existieren kann. Somit muss eine Control-Response nicht explizit zu einer Control-Request, wie bei SIP mittels einer Call-ID, zugeordnet werden und bezieht sich immer auf eine Channel-Nummer.

Wie aus dem obigen Ablauf für des Dialoges zur Anforderung einer Portnummer am WSP für eine bevorstehende Sprachdatenübertragung und Parametrisierung der korrespondierten Channel-Nummer aus der Portanfrage, ersichtlich, wird bei der Portanfrage lediglich eine Channel-Nummer dem WSP übergeben. Dieser Channel-Nummer ordnet dann der WSP einen Port für eine Datenübertragung zwischen VoIP-Komponenten im Intranet mittels UDP zu. Somit wird alles was auf diesem Port an Daten ankommt, der Channel-Nummer zugeordnet und dem jeweiligen HOBPhone mittels der SSL-Verbindung weitergeleitet. Um jedoch die Informationsrichtung vom HOBPhone in die VoIP-Architektur möglichst effizient zu verarbeiten, wird am WSP noch die Information benötigt, wohin bzw. zu welcher Komponente in der VoIP-Architektur die Datenpakete für diese Channel-Nummer die vom HOBPhone über die SSL-Verbindung an den WSP gesendet werden über die korrespondierte Portnummer mittels UDP weitergeleitet werden sollen. Durch die Parametrisierung der Channel-Nummer, werden dem WSP die benötigten Informationen als Zielangabe mitgeteilt. Die benötigten Informationen erhält das HOBPhone über den in Kap. 2.1.1.3'Verbindungsaufbau mit SIP' dargestellten Vorgang mittels des SDP in Verbindung mit SIP. Die Notation des WSP-Protokoll für Parameter wird allgemein definiert als

PARAMETER(/LOCATION):WERT\r\n

wobei die 'LOCATION' als 'target' oder 'local' optional ist (der Parameter 'channel' benötigt z.B. wird ohne eine 'LOCATION' angegeben).

Empfängt nun der WSP das Datenpaket und übergibt es dieses an den SDH, ist die einzige Information die der SDH in dieser Informationsrichtung aus den Daten in Anwendungsschicht interpretieren muss die Channel-Nummer im erwarteten HOB-Header.

### 3.1.4 Numbers in HOB Abstract Syntax Notation (NHASN)

Anhand der Channel-Nummer und den in der Konfigurationsdatei hinterlegten Informationen, kann nun das Datenpaket weitergeleitet werden, ohne dass sich der SDH mit den Daten im SIP-Teil des Datenpaketes auseinander setzen muss. Die zu übertragenden Datenpakete zwischen dem HOBPhone und dem SDH sind, unabhängig davon, ob es sich um Signalisierungs- oder Sprachdaten handelt, mit einem HOB-Header zu versehen. Der HOB-Header besteht hierbei aus zwei Komponenten, der Channel-Nummer dessen Anwendung bereits in Kap. 3.1.3 dargestellt wurde und einem numerischen Wert für die Länge des mit der Channel-Nummer korrespondierten Datenpaketes. Dieser Wert wird in der Notation 'Numbers in HOB Abstract Syntax Notation' (kurz NHASN) dargestellt. Während der Einsatz des NHASN in Zusammenhang mit der Übertragung von Signalisierungs- und Sprachdaten in den Kap. 3.2.2 im einzelnen dargestellt wird, soll an dieser Stelle als grundlage für die Architektur des HOBPhone die Notation des NHASN näher dargestellt werden.

Der NHASN ist eine Notation zur effizienten Darstellung von numerischen Werten und soll in diesem Zusammenhang als Information über die Länge der Transportierten Roh-Daten dienen. Wird z.B. ein SIP-Datenpaket der Länge 486 Bytes das mit der Channel-Schlüsselnummer 1 korrespondiert unabhängig von der Informationsrichtung mittels der SSL-Verbindung zwischen dem HOBPhone und dem SDH verschickt, so würde sich der HOB-Header aus den Informationen 1 für die Channel-Schlüsselnummer und 486 für die Länge des SIP-Datenpaketes zusammensetzen. Während für die Channel-Nummer das Werteintervall [0;255] ausreichend ist und somit bei der Zusammensetzung des HOBHeader für die Channel-Nummer ein Byte reserviert werden kann, werden für den Wert der die Paketlänge der Nutzdaten darstellen soll, welcher sich maximal innerhalb des Werteintervalls eines Integer befindet, nicht pauschal 4 Bytes reserviert, sondern der NHASN angewendet. Bei der Darstellung von numerischen Werten nach NHASN wird für den darzustellenden Wert nicht pauschal eine Größe wie z.B. Integer (4 Bytes) verwendet. Stattdessen sollen für den darzustellenden Wert gerade so viel Bytes verwendet werden, mit denen sich die Zahl darstellen lässt.

Da durch ein solches Vorgehen jedoch bei der Interpretation des übertragenen Wertes die Information fehlen würde, mit wie vielen Bytes der Wert nun dargestellt wird, soll durch NHASN ermöglicht werden, die Länge der zu interpretierenden Zahl (die wiederum selbst eine Länge als numerischen Wert darstellt) zu ermitteln. Hierzu wird von 8 Bits eines Byte das erste Bit (Most Signifacant Bit, kurz MSB) als Informationsträger reserviert. Ist dieses erste Bit gesetzt, hat dies die Bedeutung, dass der Wert, der dargestellt werden soll, sich nicht durch die restlichen 7 Bit dieses Byte darstellen lässt und ein zweites Byte verwendet wurde. Ist beim zweiten Byte das erste Bit gesetzt, würde dies bedeuten, dass der Wert der dargestellt werden soll sich nicht durch die 7 Bits des ersten Byte zusammen mit den restlichen 7 Bit des zweiten Byte, also nicht mit insgesamt 14 Bits darstellen lässt und ein drittes Byte verwendet wurde usw. Erst wenn das erste Bit mit 0 gesetzt ist, ist das letzte Glied einer aus sich dieser Notation ergebenden Kette erreicht. Indem ein Bit aus jedem Byte als Informationsträger ob die verwendeten Bits für die darzustellende Zahl sich mindestens auf das folgendes Byte erstrecken reserviert wird, können folglich nicht alle 8 Bits eines Byte für die Darstellung eines Wertes genutzt werden , sondern immer nur verbleibende 7 Bits. Die Nutzung von 7 Bits für in diesem Zusammenhang Nutzdaten bedeutet, bei einem Byte in NHASN die Möglichkeit zur Darstellung des Wertintervalls [0;127], mit zwei Byte [0;16383] usw., d.h. allgemein lässt sich bei n verwendeten Bytes in NHASN das Wertintervall

$$[0; (2^{n*7}) - 1]$$

darstellen, wobei zur Darstellung von numerischen Werten von z.B. Paketlängen in NHASN Werte größer als zwei Byte nicht zu erwarten sind. Zur Veranschaulichung soll im nachfolgenden Beispiel die Zahl 486 binär in NHASN dargestellt werden.

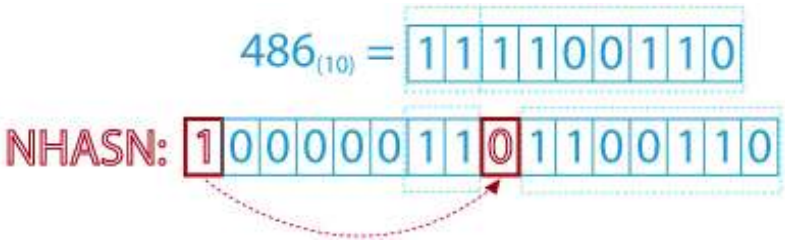


**Abbildung 3.8:** Numbers in HOB Abstract Syntax Notation für den Wert 486

Wie aus dem Beispiel in Abb. 3.8 ersichtlich, werden für die Darstellung des numerischen Wertes 486 anhand des NHASN zwei Bytes benötigt. Ausgehend von einer pauschalen Reservierung von einem Integer mit 4 Bytes für die Darstellung des benötigten Wertes, ergibt dies für das obige Beispiel eine Volumenersparnis von zwei Bytes pro Datenpaket. RTP-Datenpakete mit Sprachsequenzen, die mittels des Codec GSM610 komprimiert

wurden, haben zusammen mit 12 Bytes für den RTP-Header und 33 Bytes für den GSM610 Payload eine Paketgröße von 45 Bytes. Für die Darstellung des numerischen Wertes 45 würde man mittels NHASN lediglich 1 Byte benötigen, was im Gegensatz zur pauschalen Reservierung für die Darstellung der Paketgröße mit einem Integer von 4 Bytes eine Volumenersparnis von 3 Byte pro Paket bedeutet. Ausgehend von der üblichen Anzahl von 50 Paketen in der Sekunde bei einer Übertragung von Sprachdaten mittels des RTP, würde dies eine Senkung der Datenrate um 1,2 kbit/s bedeuten, was sich aus 50 * 3 Bytes = 150 B/s = 1200 bit/s = 1,2 kbit/s ergibt. Im Vergleich zur Datenrate des Codecs GSM610 von 13 kbit/s, entspricht dies einer Senkung der Datenrate um ca. 9,2 % bei Anwendung des NHASN anstatt der pauschalen Zuordnung von 4 Bytes für die Darstellung der Paketgröße.

## 3.2 Implementierung eines SIP-Client

Für die Festlegungen bei der Architektur des SIP-Clients als HOBPhone wird die in Kap. 3.1 definierte Architektur des Konzeptes als Grundlage genommen, wobei die Implementierung des HOBPhone als Java-Client sich in drei Hauptbereiche untergliedert:

- Übertragung der Signalisierungs- und Sprachdaten
- Verarbeitung der Signalisierungs- und Sprachdaten
- Benutzerschnittstelle

### 3.2.1 HOBPhone Architektur

Die Architektur des HOBPhone im Bezug zur Verarbeitung der Signalisierungs- und Sprachdaten wird unabhängig von der Art der Datenübertragung (UDP oder SSL) definiert. Hierzu wird für SIP- und RTP-Datenpakete jeweils ein interface implementiert, welches einen Zugriff auf Empfangene Daten und den Versand von Daten, unabhängig von der Art der Datenübertragung, welche durch die weiter oben erwähnte Einstiegsmarke der Anwendung definiert wird, ermöglichen soll. Die jeweilige Schnittstelle zum Empfang und Versand von SIP- oder RTP-Datenpaketen wird in den Klassen `SIPConnection.java` und `RTPConnection.java` definiert, wobei im Folgenden zunächst die Verarbeitung von Signalisierungsdaten über SIP-Datenpakete näher behandelt werden soll.

**Listing 3.5:** Interface SIPConnection.java

```
1  public interface SIPConnection {
2    public void init(SIPManager listenerp);
3    public void terminateSIPConnection();
4    public boolean sendSIP(byte[] byr_data, int im_offset, int im_len);
5    public String getLocalINETA();
6    public int getSIPLocalPort();
7    public String getSIPGatewayINETA();
8    public SIPManager getSIPManager();
9    public boolean connect();
10 }
```

Das oben aufgeführte Interface, soll in seiner vordefinierten Funktionalität bzw. Methodik als Schnittstelle zwischen Übertragung und Verarbeitung für die entsprechenden Daten für SIP dienen, indem wie z.B. in Kap. 3.2.2.1 'UDP-Modus' dargestellt wird, eine Instanz der Klasse `SIPConnectionUDP.java`, welche das Interface SIPConnection implementiert, somit als `SIPConnection` verwendet werden kann und eine Unabhängigkeit von der tatsächlichen Bereitstellung der zu verarbeitenden Daten gewährleistet ist. Die nachstehende Funktion `sendResponse` der Klasse `SIPManager.java` zeigt als Beispiel, wie

mittels einer Instanz einer Klasse die das Interface `SIPConnection.java` implementiert, hier `sip_con`, eine SIP-Response als Datenpaket versendet werden kann:

**Listing 3.6:** SIPManager.java -sendResponse

```
1   public boolean sendResponse(SIPClient sip_client, int imp_msgKey,
2                   String str_content, SIPMessage.SIPHeader sip_hdr_in) {
3      byte[] byr_msg = prepareResponse(sip_client, imp_msgKey, str_content
          ,
4                                       sip_hdr_in);
5       if ( byr_msg != null) {
6         return sip_con.sendSIP(byr_msg,0,byr_msg.length);
7       } else {
8         return false;
9       }
10    }
```

Entscheidend ist bei der Verarbeitung von SIP-Daten zunächst die Beziehung von SIP-Clients zur Telefonanlage Asterisk. Hierbei kann ein SIP-Client $n$ Accounts bei jeweils $n$ Asterisk besitzen. Durch die weiter oben erwähnte Schnittstelle soll hierbei für jede erforderliche Möglichkeit einer differenzierbaren Datenübertragung die ein SIP-Client zu unterschiedlichen Astersik hat bzw. benötigt, unabhängig von der Anzahl der jeweilig korrespondierten SIP-Accounts, zu $n$ Objekten der Klasse `SIPConnection.java` instantiiert werden. Ein SIP-Client in seiner grundlegenden Funktionalität, wird durch die Klasse `SIPClient.java` definiert.

Zu jeder Instanz der Klasse `SIPConnection.java` stehen $n$ Instanzen der Klasse `SIPClient .java` Beziehung, welche im folgenden als `sip_client` benannt werden. Die Verwaltung der $n$ `sip_client`´s korrespondierend zu einer Instanz der Klasse `SIPConnection.java`, erfolgt durch die weiter oben erwähnte Klasse `SIPManager.java` die in einer 1:1 Beziehung zu der Klasse `SIPConnection.java steht`. Die Verwaltung von $n$ möglichen Instanzen der Klasse `SIPConnection.java` wiederum erfolgt durch eine Instanz der Klasse `PhoneManager.java`, über welche auch die jeweilige Zuordnung aller Instanzen der Klassen `SIPConnection.java` und `RTPConnection.java` zu einer expliziten Datenübertragungsart erfolgt.

Ausgehend von den Beziehungen von SIP-Clients im Bezug zur Telefonanlage Asterisk, werden sämtliche Zusammenhänge zur Verwaltung der entsprechenden Schnittstellen für die Datenbereitstellung und Datenverarbeitung von Signalisierungs- und Sprachdaten auf die Klasse `PhoneManager.java` als zentrale Verwaltungseinheit konzentriert. In Abb. 3.9 werden die Beziehungen der die Klassen im Package hob.phone im Klassendiagramm dargestellt.

Bei der Verarbeitung von Sprachdaten, welche mittels RTP-Datenpakete übermittelt werden, ist ausgehend von der definierten Funktionalität des HOBPhone in Kap. 3.1 zu beachten, dass in dieser Version des Konzeptes , die Möglichkeit mehrere Gespräche

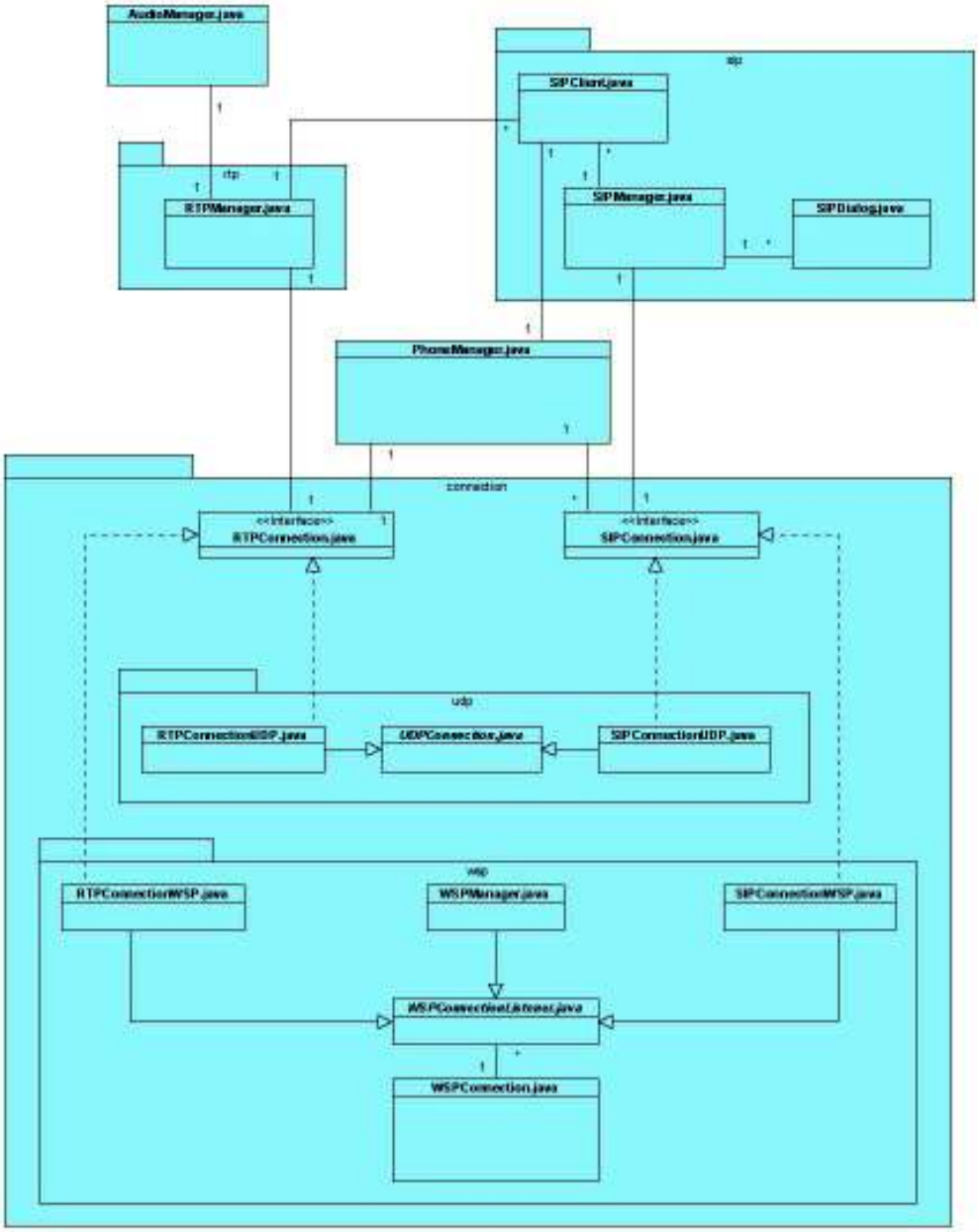


**Abbildung 3.9:** Klassendiagramm hob.phone

zur gleichen Zeit zu führen (Konferenzen) nicht implementiert wird. Folglich ist die Anzahl der möglichen Sprachdatenübertragungen zum gleichen Zeitpunkt im Gegensatz zur Verarbeitung von Signalisierungsdaten auf eine beschränkt.

Wie aus Abb. 3.9 ersichtlich, wird das Konzept für die Verarbeitung von Signalisierungsdaten für die Verarbeitung der Sprachdaten lediglich mit der Unterscheidung der 1:1 Beziehung der Klassen `RTPConnection.java` zu `PhoneManager.java` übernommen. Zwar erfolgt die Verwaltung von $n$ `sip_client`´s im Bezug auf mehrere mögliche Sprachdatenübertragungen über eine Instanz der Klasse `RTPManager.java`, jedoch stellt die Klasse `PhoneManager.java` aufgrund der nur einen möglichen Sprachdatenübertragung zum gleichen Zeitpunkt in dieser Version lediglich eine Instanz der Klasse `RTPConnection .java` zur Verfügung, welche sich $n$ `sip_client`´s für mögliche Sprachdatenübertragungen entsprechend zeit versetzt teilen müssen.

Ausgehend von der voneinander unabhängigen Konzeption der drei Hauptbereiche wird durch die abstrakte Klasse `PhoneUserInterface.java` die Implementierung möglicher Benutzeroberflächen definiert, wobei zu beachten ist, dass durch den Klassennamenszusatz 'Interface' ist in diesem Zusammenhang kein Interface als Schnittstelle im Sinne der Implementierung eine Schnittstelle in Java gemeint ist, sondern der Bezug auf eine Benutzerschnittstelle hergestellt werden soll. Durch die Abstrahierung der Benutzerschnittstelle wird eine unabhängige Implementierung der Komponenten an sich und möglicher Benutzerschnittstellen voneinander gewährleistet. Die erforderliche Kommunikation bei der Verarbeitung von Signalisierungs- und Sprachdaten und der Benutzerschnittstelle erfolgt hierbei mittels definierter Events, was in Kap. 3.2.5'Grafische Benutzeroberfläche' im einzelnen näher erläutert wird.

### 3.2.2 Übertragung von Signalisierungs- und Sprachdaten

Für die Art der Übertragung von Signalisierungs- und Sprachdaten als Modus in dem das HOBPhone eingesetzt wird, ist die jeweilige Einstiegsmarke anhand der über die HTML-Seite übergebenen Parameter im Bezug auf den Modus des HOBPhone als `WSP` oder `SSL` in der Klasse `MainApplet.java` ausschlaggebend, was in Kap. 3.2.5 im einzelnen dargestellt wird.

#### 3.2.2.1 UDP-Modus

Das für eine Datenübertragung mittels UDP maßgebliche Vorgehen des HOBPhone wird unabhängig davon, ob es sich um die Übertragung von SIP- oder RTP-Datenpaketen handelt als Basis die abstrakte Klasse `UDPConnection.java` implementiert. Ausgehend von

der in Kap. 3.2.1 definierten Architektur des HOBPhone, eine unabhängige Implementierung der Komponenten für die Übertragung und Verarbeitung von Daten zu gewährleisten, werden die Klassen `SIPConnectionUDP.java` und `RTPConnectionUDP.java` im Bezug auf die Datenübertragung mittels UDP einerseits von der Klasse `UDPConnection.java` abgeleitet, implementieren jedoch gleichzeitig das Interface `SIPConnection.java` im Bezug auf die allgemeine Definition einer SIP- oder RTP-Connetion zur von der Art der Datenübertragung unabhängigen Verarbeitung der Daten:

**Listing 3.7:** SIPConnectionUDP.java

```
public class SIPConnectionUDP extends UDPConnection implements
    SIPConnection
```

**Listing 3.8:** RTPConnectionUDP.java

```
public class RTPConnectionUDP extends UDPConnection implements
    RTPConnection
```

Im Bezug zu den folgenden Ausführungen ist zu beachten, dass der UDP-Modus ausschließlich für einen internen Einsatz innerhalb einer VoIP-Architektur konzipiert ist.

Innerhalb der Klasse `UDPConnection.java` werden Methoden und Funktionen die allgemein für eine Übertragung von Daten mittels UDP maßgeblich sind ausformuliert. Diese sind vor allem die Erzeugung einer Instanz der Klasse `DatagramSocket.java` aus dem Package `java.net` und der Versand und Empfang von UDP-Datenpaketen mittels dieser Instanz. Nachstehender Quellcode veranschaulicht die Funktion `send()` der abstrakten Klasse `UDPConnection.java`, welche für alle Instanzen von Klassen die von dieser Klasse erben gerufen wird.

**Listing 3.9:** UDPConnection.java -send

```
public boolean send (byte[]byr_data,
  int im_offset, int im_len){
  try {
    dgs_udp.send(new DatagramPacket(byr_data,im_offset , im_len,
        ina_udp_dest, im_destPort));
    return true;
  } catch (Exception ex2) {
  Debug.err(this.getClass().toString(),"Error: Cannot send packet " + ex2)
      ;
  return false;
  }
}
```

In Zusammenhang zu der oben aufgeführten Funktion, soll anhand der nachstehend dargestellten Funktion der Klasse `SIPConnectionUDP.java` der Anwendung der abstrakten Klasse `UDPConnection.java` und dem Interface `SIPConnection.java` durch der Klasse `SIPConnectionUDP.java` verdeutlicht werden:

**Listing 3.10:** SIPConnectionUDP.java -sendSIP

```
public boolean sendSIP(byte[] byr_data, int im_offset, int im_len) {
  return send(byr_data, im_offset, im_len);
}
```

Die Funktion `sendSIP()` ist im Interface `SIPConnection.java` für alle Klassen die das Interface implementieren definiert(vgl. Listing 3.2.1 in Kap. 3.2.1'HOBPhone Architektur'). Somit muss die Funktion `sendSIP()` in `SIPConnectionUDP.java` implementiert werden, nachdem alle Instanzen der Klasse `SIPConnectionUDP.java` anhand des Interface eine `SIPConnection.java` darstellen bzw. repräsentieren. Die Funktion `send()` steht hierbei einer Instanz der Klasse `SIPConnectionUDP.java` durch die Vererbung der `UDPConnection.java` auch zur Verfügung. Wird also die Funktion `sendSIP()` über ein Objekt welches als `SIPConnection.java` definiert ist, gerufen, so werden die Daten im Falle einer Instanz der Klasse `SIPConnectionUDP.java` über die Funktion `send()` der Klasse `UDPConnection.java` und somit über eine Instanz der Klasse `DatagramSocket.java` mittels UDP übertragen. Methoden und Funktionen, die zwar eine Übertragung von Daten mittels UDP gemeinsam haben, jedoch im Vorgehen bei SIP- und RTP-Datenpaketen zu unterscheiden sind, werden abstrakt definiert.

**Listing 3.11:** UDPConnection.java -receivedData

```
public abstract void receivedData(byte[] byrp_data,int imp_offset, int imp
    _len);
```

Das Vorgehen innerhalb der Methode `receivedData`, ist somit für mittels UDP empfangene Datenpakete je nachdem, ob es sich um eine Instanz der Klasse `SIPConnectionUDP.java` oder `RTPConnectionUDP.java` handelt, unterschiedlich. Nachstehender Quellcode soll die unterschiedlichen Vorgehensweisen der abstrakt definierten Methode verdeutlichen.

**Listing 3.12:** SIPConnectionUDP.java -receivedData

```
public void receivedData(byte[] byrp_data, int imp_offset, int imp_len) {
  sip_mngr.sipReceived(
    new SIPMessage(new DatagramPacket(byrp_data,imp_offset,imp_len))
  );
}
```

**Listing 3.13:** RTPConnectionUDP.java -receivedData

```
public void receivedData(byte[] byrp_data, int imp_offset, int imp_len) {
     rtp_mngr.playPacket(byrp_data,imp_offset, imp_len);
}
```

Wie aus dem dargestellten Quellcode ersichtlich, werden empfangene SIP-Datenpakete zur Verarbeitung einer Instanz der Klasse `SIPManager.java` übergeben, wobei empfangene RTP-Datenpakete der Instanz der Klasse `RTPManager.java` zur Wiedergabe der

Sprachsignale übergeben werden. Die möglichen Szenarien zum Empfang und Versand von Datenpaketen im Bezug auf die Beziehungen der Instanzen der betroffenen Klassen, wurden in Kap. 3.1 'Architektur des Konzeptes' dargestellt bzw. definiert. Der Einsatz von Instanzen der Klasse `SIPConnectionUDP.java` richtet sich nach den jeweiligen Benutzereingaben über eine Benutzerschnittstelle. Demnach wird für jede Datenübertragung mit $n$ möglichen Asterisk welche abhängig von den Benutzereingaben ist, eine Instanz der Klasse `SIPConnectionUDP.java` erzeugt. Dies soll anhand dem nachstehenden Ausschnitt der Funktion `setClient` der Klasse `PhoneManager.java` verdeutlicht werden.

**Listing 3.14:** PhoneManager.java -hmp_SIPConnections

```
if (hmp_SIPConnections.containsKey(str_sip_host)) {
    if (this.bo_phone_DEBUG)
  Debug.out(this.getClass().toString(),
    "Using CURRENT SIPConnectionUDP to set Client: "+ str_sip_host);
  sip_con = ((SIPConnection) hmp_SIPConnections.get(str_sip_host));
  sip_client.setClient(sip_con, sip_con.getSIPManager(), rtp_mngr,mp_ua_
      input);

} else {

if (bo_phone_DEBUG) {
    Debug.out(this.getClass().toString(),
    "NEW SIPConnectionUDP pending...");
  }
  sip_con = new SIPConnectionUDP(str_sip_host, Constants.SIP_PORT);
}
```

Die Funktion `setClient` der Klasse PhoneManager.java wird über eine jeweilige Instanz immer dann gerufen, wenn über die Benutzerschnittstelle Informationen bezüglich eines SIP-Accounts eingegeben werden.

Mit der Überprüfung in Zeile eins in Listing 3.2.2.1 wird zunächst festgestellt, ob die IP-Adresse des Asterisk der für diesen SIP-Account eingegeben wurde, bereits verwendet wird. Ist dies der Fall, wird dem für den neu eingegebenen SIP-Account erstellten Instanz der Klasse `SIPClient.java`, die bereits für diesen Astersik vorhandene Instanz der Klasse `SIPConnection.java` zur Datenübertragung übergeben(Listing 3.2.2.1 Z.5-6).

Der `sip_client` trägt sich bei dem `sip_mngr` der übergebenen `sip_con` als neuer `sip_client` für die Weiterleitung von für diesen `sip_client` korrespondierten SIP-Datenpaketen ein, was in Zeile 15 im nachstehenden Quellcode der Methode `setClient()` aus der Klasse `SIPClient.java` dargestellt wird:

**Listing 3.15:** SIPClient.java -setClient

```
public void setClient(SIPConnection conp, SIPManager sip_mngrp,
      RTPManager rtp_mngrp, Map<Integer, String> mp_client) {
```

```
3     this.sip_con = conp;
4     this.sip_mngr = sip_mngrp;
5     this.rtp_mngr = rtp_mngrp;
6     str_localINETA = sip_con.getLocalINETA();
7
8     str_fullName = (String) mp_client.get(Constants.HMP_FULLNAME);
9     str_userName = (String) mp_client.get(Constants.HMP_USERNAME);
10    str_sip_host = (String) mp_client.get(Constants.HMP_HOST);
11    str_password = (String) mp_client.get(Constants.HMP_PWD);
12    str_contact = "<sip:" + str_userName + "@" + str_localINETA + ":"
13        + sip_con.getSIPLocalPort() + ";" + "transport="
14        + Constants.strf_transport + ">";
15    sip_mngr.addSIPClient(this);
16    bo_clientIsSet = true;
17  }
```

Handelt es sich bei den Benutzereingaben um eine neue Asterisk die bisher noch nicht verwendet wurde bzw. für die bisher noch kein Account vorhanden war, so wird eine neue Instanz der Klasse `SIPConnectionUDP.java` erstellt (Z. 14 3.2.2.1).

Diese Instanz ist allgemein als `SIPConnection sip_con` definiert. Aufgrund der Beziehung von 1:1 von Instanzen der Klassen `SIPManager.java` und `SIPConnection.java`, ist eine Instanz der Klasse `SIPManager.java` für die Verwaltung von Daten bzw. der Kommunikation mit diesem Asterisk zu erzeugen, der die `sip_con` übergeben wird:

**Listing 3.16:** PhoneManager.java -new SIPManager(sip_con)

```
1 if (bo_connection) {
2   sip_mngr = new SIPManager(sip_con);
3   sip_con.init(sip_mngr);
4   addSIPConnection(str_sip_host, sip_con);
5   sip_client.setClient(sip_con, sip_mngr, rtp_mngr, mp_ua_input);
6 }
```

#### 3.2.2.2 SSL-Modus

Die Klasse `WSPConnection.java` definiert den Empfang und Versand von Daten über eine SSL-Verbindung auf Basis der in Kap. 3.1 definierten Architektur. Bevor eine Datenübertragung stattfinden kann, ist zunächst der Aufbau einer SSL-Verbindung erforderlich. Die SSL-Verbindung zwischen dem HOBPhone und dem WSP wird mittels einer Implementierung von HOB realisiert. Hierzu wird das Package `hob.ssl` in die für den Empfang und Versand von Daten über eine SSL-Verbindung definierte Klasse `WSPConnection.java` integriert und stellt bei erfolgter SSL-Verbindung eine Instanz der Klasse `java.net.Socket`, die im weiteren als `dsl_sock_01` dargestellt wird, bereit vgl. Listing 3.2.2.2). Ist die SSL Verbindung mit dem WSP anhand dem über die HTML-Seite übergebenen Parameter 'URL' des WSP aufgebaut, so wird über die Funktion

`connect()` ein Thread zum Empfang von Daten gestartet, wobei für den Versand und Empfang von Daten je eine Instanz der Klassen `Inputstream.java` und `Outputstream.java` aus dem Package java.io erstellt wird:

**Listing 3.17:** WSPConnection.java -Input/Output-Stream

```
1  dsc_sos_input = dsl_sock_01.getInputStream(); /* incoming data */
2  dsc_sos_output = dsl_sock_01.getOutputStream(); /* outgoing data */
```

Zunächst soll der Empfang von Daten näher betrachtet werden. Das für den Empfang von Datenpaketen maßgebliche Vorgehen wird anhand der zu erwartenden Datenpakete definiert. Die Beschaffenheit von möglichen Datenpaketen im Bezug auf den HOB-Header und die Beziehungen der durch die Channel-Nummer korrespondierten Instanzen entsprechender Klassen, wurden in Kap. 3.1.2 'Channel-Mechanismus' dargestellt. Während beim Empfang von SIP-Datenpaketen mittels UDP die Beziehung einer Instanz der Klasse `UDPConnection.java` zu einer Instanz der Klasse `SIPConnectionUDP.java` 1:1 ist, was bedeutet, dass jede Instanz der Klasse `SIPConnectionUDP.java` eine eigene Instanz der Klasse `DatagramSocket.java` zum Versand und Empfang von Daten hat, konzentrieren sich die Datenpakete von korrespondierten $n$ Instanzen der Klasse `SIPConnectionWSP.java` auf eine SSL-Verbindung und somit auf eine Instanz der Klasse `WSPConnection.java`.

Die Zuordnung von Datenpaketen zu den korrespondierten Instanzen einer Klasse innerhalb der Klasse `WSPConnection.java` erfolgt unabhängig davon, ob es sich um SIP-, RTP- oder Control-Datenpakete gemäß dem WSP-Protokoll handelt, anhand der im HOB-Header übermittelten Channel-Nummer. Dadurch, dass Instanzen von Klassen die Daten von der Klasse `WSPConnection.java` erwarten sich lediglich durch eine Channel-Nummer unterscheiden und der Vorgang des Empfangs von Daten allen gleich ist, wird die abstrakte Klasse `WSPConnectionListener.java` implementiert, deren Quellcode nachstehend dargestellt wird.

**Listing 3.18:** WSPConnectionListener.java

```
1 public abstract class WSPConnectionListener {
2 public abstract void receivedData(byte[] byrp_data,int imp_offset, int imp
      _len);
3 }
```

In Zusammenhang dazu, wird im folgenden die Konstruktion der möglichen Klassen dargestellt, deren Instanzen zur Verarbeitung Daten seitens der Klasse `WSPConnection.java` erwarten.

**Listing 3.19:** SIPConnectionWSP.java

```
1 public class SIPConnectionWSP extends WSPConnectionListener implements
2     SIPConnection{
3 public SIPConnectionWSP(WSPConnection ssl_conp, int imp_channel){
```

```
4   this.wsp_con = ssl_conp;
5   this.im_channel = imp_channel;
6 }
7 .
8 .
9 .
10 }
```

**Listing 3.20:** RTPConnectionWSP.java

```
1 public class RTPConnectionWSP extends WSPConnectionListener implements
      RTPConnection {
2 .
3 .
4 .
5
6 }
```

**Listing 3.21:** WSPManager.java

```
1 public class WSPManager extends WSPConnectionListener{
2 .
3 .
4 .
5
6 }
```

Die Klassen `SIPConnectionWSP.java` und `RTPConnectionWSP.java` haben somit den Empfang von Datenpaketen als `WSPConnectionListener.java` gemeinsam und repräsentieren gleichzeitig jeweils über das jeweilige Interface Instanzen der Klassen `SIPConnection .java` und `RTPConnection.java`. Im Gegensatz zum UDP-Modus des HOBPhone, erfolgt die Verwaltung von $n$ SIP-Accounts an $n$ möglichen Asterisk nicht durch Benutzereingaben über die Benutzerschnittstelle, sondern über die Administration der Benutzerdaten am WSP die dem HOBPhone unmittelbar nach Aufbau der SSL-Verbindung mittels des WSP-Protokoll mitgeteilt werden (vgl. Kap. 3.1.3 'WSP-Protokoll').

Demnach erfolgt eine Zuteilung einer Channel-Nummer für eine Instanz der Klasse `SIPConnectionWSP.java` für die Verarbeitung von SIP-Datenpaketen anhand der Informationen der Signalisierungsdaten zwischen dem HOBPhone und dem SDH über den WSP anhand des WSP-Protokoll. Für die Verarbeitung der Signalisierungsdaten und somit das Vorgehen bei der Erzeugung entsprechender Instanzen der Klassen `SIPConnectionWSP.java` und `RTPConnection.java` ist die Klasse `WSPManager.java` definiert, welche in Kap. 3.2.3.2 'Verarbeitung von WSP-Signalisierungsdaten' im einzelnen dargestellt wird. Um die Verarbeitung der erwähnten Signalisierungsdaten zu gewährleisten, wird nach Aufbau der SSL-Verbindung eine Instanz der Klasse `WSPManager.java` erzeugt und die in Kap. 3.1 definierte Channel-Nummer '0' für Control-Pakete des WSP-Protokoll zugeordnet:

**Listing 3.22:** WSPConnection.java -addListener

```
  addListener(Constants.WSP_CONTROL_CHANNEL + "",
              WSPManager.getWSPManager(ui, this));
```

Wie aus dem oben dargestellten Quellcode ersichtlich, wird die Instanz der Klasse `WSPManager.java` mittels `addListener` als `WSPConnectionListener.java` dem Container hinzugefügt, aus dem anhand der im HOB-Header empfangenen Channel-Nummern die korrespondierten Instanzen ermittelt und die Datenpakete weitergeleitet werden, was im weiteren Verlauf dargestellt wird. Die für eine Weiterleitung eines Datenpaketes als Informationseinheit erforderliche Interpretation des HOB-Header soll hierzu im einzelnen näher betrachtet werden. In Abb. 3.10 wird die Beschaffenheit der durch den Input-Stream empfangenen Daten verdeutlicht, welche für das weitere Vorgehen maßgeblich sein wird.

Die in Abb. 3.10 in seiner Beschaffenheit verdeutlichte Sequenz von Datenpaketen wird innerhalb einer `while`-Schleife anhand der Funktion `read` der weiter oben dargestellten Instanz der Klasse `InputStream.java` verarbeitet:

**Listing 3.23:** WSPConnection.java -loop to receive data

```
while (true) { /* loop to receive data */
  try {
  iml_len_rec = dsc_sos_input.read(byrl_recv); /* receive data */
  } catch
 (IOException eh1) {
  Debug.err(this.getClass().toString(),
  "jbthvoipc1-l00000FI;-E Socket read Error : " + eh1);
  // terminateConnection();
  break;
  }
.
.
.
}
```

Die Verarbeitung der als Stream empfangenen Datenpakete erfolgt innerhalb einer verschachtelten `while`-Schleife:

**Listing 3.24:** WSPConnection.java -loop over input received

```
while (true) { /* loop over input received */
  iml_rem = iml_len_rec -iml_rp; /* remaining part of record */
  if (iml_rem <= 0)
  break; /* end of record */
  switch (iml_state_1) { /* part one of state */
.
.
.
```

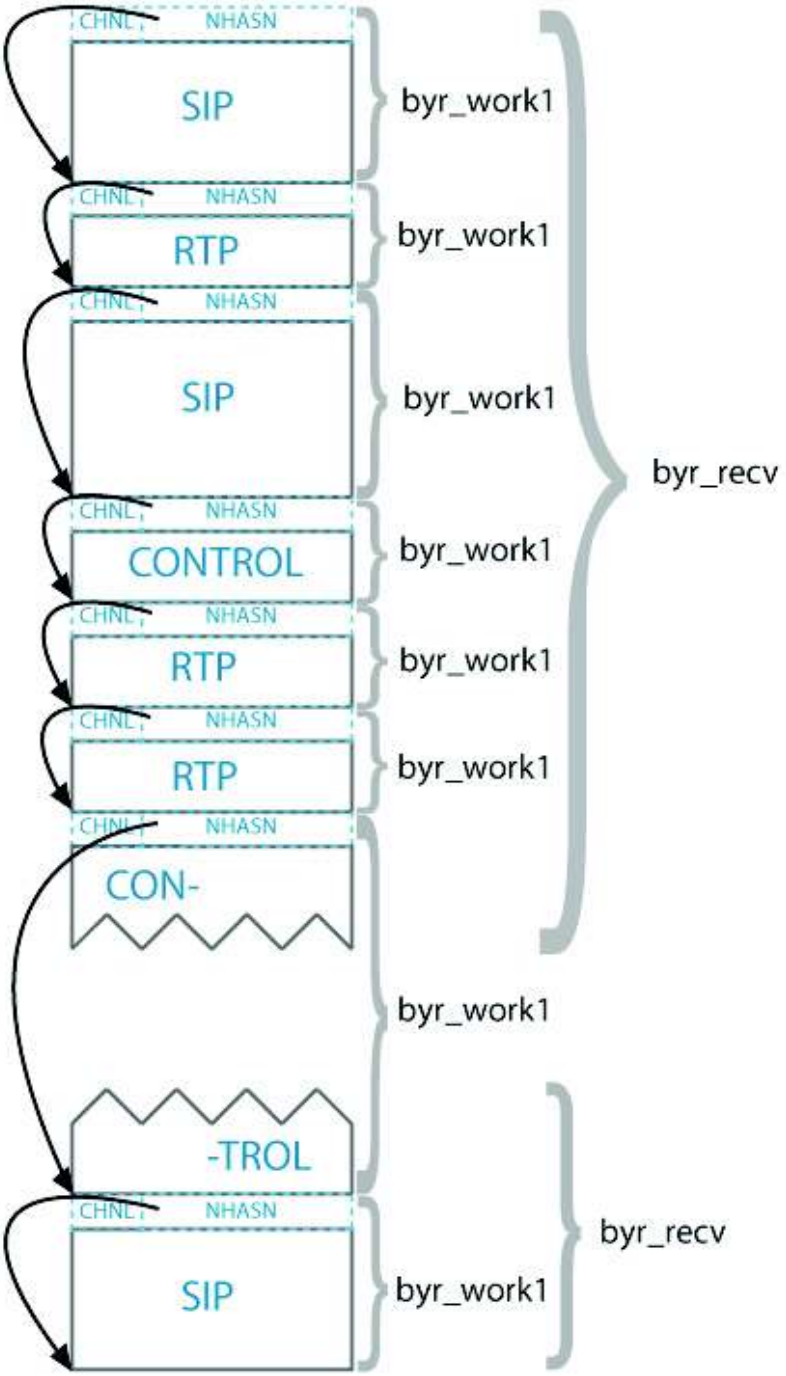


**Abbildung 3.10:** Daten im SSL-Modus

```
9   }
10 }
```

Diese verschachtelte `while`-Schleife innerhalb der Schleife für den Empfang stellt eine sog. 'State-Machine' dar. Demnach wird die zu verarbeitende Sequenz von Daten am Status der Bearbeitung in Form von

- Channel empfangen
- NHASN ermittelt
- Datenpaket empfangen

Der temporäre Status der Verarbeitung wird in der Varialbe `iml_state_1` zwischengespeichert, der mittels einer `switch`-Konstruktion bei jedem Durchlauf der verschachtelten Schleife unterschieden wird (vgl. Listing 3.2.2.2). Dabei wird für jedes interpretierte Byte aus dem Stream ein Zähler inkrementiert, der durch die Variable `iml_rp` dargestellt wird. Dieser 'Read-Pointer' ist in Zusammenhang mit dem NHASN-Wert für die zu erwartende Paketlänge und der verbliebenen Daten im Stream maßgeblich für die Feststellung von fragmentierten Datenpaketen, worauf im weiteren Verlauf näher eingegangen wird. Zunächst wird anhand des HOB-Header die übermittelte Channel-Nummer für ein Datenpaket interpretiert:

**Listing 3.25:** WSPConnection.java -receive channel

```
1
2 case by_STATE_RECV_CHANNEL: /* receive channel */
3   byl_channel = byrl_recv[iml_rp++]; /* channel received */
4   iml_state_1 = by_STATE_RECV_NHASN; /* receive length NHASN */
5   iml_p_len = 0; /* clear length packet */
6   break; /* all done */
```

Aufgrund der in Abb. 3.10 abgebildeten Beschaffenheit der Daten, wonach Datenpakete die mit unterschiedlichen Channel-Nummer korrespondieren als Stream empfangen werden, ist die Paketlänge des Datenpaketes, welches im Stream zu der ermittelten Channel-Nummer gehört erforderlich, welche wie in NHASN dargestellt ist:

**Listing 3.26:** WSPConnection.java -receive NHASN

```
1 case by_STATE_RECV_NHASN: /* receive length NHASN */
2   iml_p_len <<= 7; /* shift old bits length packet */
3   byl1 = byrl_recv[iml_rp++]; /* get new data */
4   iml_p_len |= (byl1 & 0X7F); /* apply new bits */
5   if (iml_p_len > imf_maxlen) { /* length too big */
6     Debug.err(this.getClass().toString(),
7 "jbthvoipc1-l00000FI;-E Error received single packet too big "+ iml_p_len)
      ;
8 /* abend of client / communication */
9   }
```

```
10  if ((byl1 & 0X80) != 0)
11  break; /* more bit was set */
12  byrl_work1 = null; /* no packet yet */
13  iml_state_1 = by_STATE_RECV_PACKET; /* receive packet */
14  break; /* all done */
```

Ist die korrespondierte Paketlänge des Datenpaketes als Informationseinheit ermittelt, wird der Status auf Empfang des Datenpaketes bzw. der Rohdaten gesetzt:

**Listing 3.27:** WSPConnection.java -receive packet

```
1  case by_STATE_RECV_PACKET: /* receive packet */
2    if (byrl_work1 == null) { /* no packet yet */
3      if (iml_rem < iml_p_len) { /* needs more data */
4        byrl_work1 = new byte[iml_p_len]; /* create new buffer */
5        iml_p_len -= iml_rem; /* compute area not filled yet */
6        iml1 = iml_rp; /* save read-pointer */
7        while (iml_rp < iml_len_rec) {
8        /*loop to copy input data to buffer*/
9        byrl_work1[iml_rp -iml1] = byrl_recv[iml_rp++];
10       }
11       break; /* receive more data */
12     }
13     byrl_work1 = byrl_recv; /* use receive buffer */
14     iml1 = iml_rp; /* this is offset */
15     iml2 = iml_p_len; /* length of packet */
16     iml_rp += iml_p_len; /* read pointer after packet */
17   } else { /* copy to buffer */
18     iml1 = iml_rem;
19     if (iml1 > iml_p_len)
20     iml1 = iml_p_len;
21     /* displacement in buffer */
22     iml2 = iml_rp -(byrl_work1.length -iml_p_len);
23     iml_p_len -= iml1; /* area not filled later */
24     iml1 += iml_rp; /* compute end of loop */
25     while (iml_rp < iml1) { /* loop to copy input data to buffer */
26     byrl_work1[iml_rp -iml2] = byrl_recv[iml_rp++];
27     }
28     if (iml_p_len > 0)
29     break; /* wait for more data */
30     iml1 = 0; /* this is offset */
31     iml2 = byrl_work1.length; /* length of packet */
32     }
33     /* one single packet received complete */
```

Wie bereits weiter oben erwähnt, kann es hierbei vorkommen, dass im empfangenen Stream nicht mehr genug Daten vorhanden sind, um das Datenpaket zu vervollständigen, und somit das letzte Datenpaket als Fragment einer Informationseinheit empfangen wurde.

Wie in Abb. 3.10 abb dargestellt, werden bei der Verarbeitung des Stream zwei Byte-Arrays verwendet. Während das Byte-Array `byrl_recv` die Empfangenen Daten des InputStream beinhaltet, wird in das Byte-Array `byrl_work1` die Datensequenz aus dem Stream kopiert, welche nach dem NHASN aus dem HOB-Header ein Datenpaket als Informationseinheit darstellt. Dieser Vorgang wird solange wiederholt, bis ein aus dem jeweiligen HOB-Header korrespondiertes Datenpaket vollständig empfangen wurde oder der Empfangs-Puffer ausgeschöpft ist. Ist eine Informationseinheit fragmentiert, so wird die verschachtelte Schleife mit dem Status 'Datenpaket Empfangen ' verlassen. Nach dem in der äußeren Schleife wieder neue Daten empfangen wurden, erfolgt die Verarbeitung erneut in der verschachtelte Schleife mit der `switch`-Konstruktion, diesmal mit dem Status 'Datenpaket Empfangen '. Nun wird das Byte-Array `byrl_work1` dass eine Datensequenz einer Informationseinheit als Fragment des letzten Datenpaketes beinhaltet, mit den verbleibenden Daten ergänzt, wobei die erforderliche Restlänge und der korrespondierte Channel aus den zwischengespeicherten letzten Werten des HOB-Headers als Channel-Nummer und NHASN gegeben sind.

Ist die Informationseinheit vervollständigt, erfolgt die Weitergabe des Datenpaketes an eine Instanz der Klasse `WSPConnectionReceiver.java`, die mit der Channel-Nummer korrespondiert, und der Status wird wieder auf 'Channel empfangen' gesetzt:

**Listing 3.28:** WSPConnection.java -single packet received complete

```
/* one single packet received complete */

  if (hmp_listeners.containsKey(byl_channel + "")) {
  ((WSPConnectionListener) hmp_listeners.get(byl_channel + ""))
  .receivedData(byrl_work1, iml1, iml2);
  } else {
  Debug.m_console_out(this.getClass().toString(),
  "jbthvoipc1-l00000FI;-E Error received single packet invalid channel: "
          + byl_channel + " offset: " + iml1 + " length: "
          + iml2, byrl_work1, iml1, iml2, true);
  }

  iml_state_1 = by_STATE_RECV_CHANNEL; /* receive channel */
  break; /* all done */
```

Wie bereits in Kap.3.1 'Architektur des Konzeptes' definiert, müssen sämtliche Datenpakete, die zwischen dem HOBPhone und dem WSP über den SDH versendet und empfangen werden, unabhängig von der Informationsrichtung, einen HOB-Header beinhalten. Die Generierung des HOB-Header erfolgt hierzu innerhalb der Funktion `m_send_1`, die über die jeweiligen Schnittstellen `SIPConnection.java` und `RTPConnection.java` mittels der Funktionen `sendSIP` und `sendRTP` gerufen wird. Hierzu wird der Funktion `m_send _1` die Channel-Nummer der Instanz übergeben, die den Versand von Daten veranlasst hat, also eine Instanz der Klassen `SIPConnectionWSP.java`, `RTPConnectionWSP.java`

oder `WSPManager.java`. Anhand der übergebenen Channel-Nummer und der tatsächlichen Paketlänge wird ein entsprechender HOB-Header generiert und zusammen mit dem Datenpaket in den OutputStream geschrieben:

**Listing 3.29:** WSPConnection.java -send message

```
/* send message */
dsc_sos_output.write(byrlwork1, iml_off, (8 -iml_off) + iml_len);
```

Die weitere Verarbeitung des Datenpaketes anhand der Channel-Nummer erfolgt im SDH, was in Kap. 3.3.2 'Verarbeitung von Datenpaketen im SDH' dargestellt wird.

### 3.2.3 Verarbeitung von Signalisierungsdaten

In den folgenden Kapiteln wird die Implementierung des HOBPhone im Bezug auf die notwendige Verarbeitung von Signalisierungsdaten dargestellt.

#### 3.2.3.1 Verarbeitung von Signalisierungsdaten (SIP,SDP)

Bei der Verarbeitung der Signalisierungsdaten, welche als SIP-Datenpakete von einer Instanz der Klasse `SIPConnection.java` bereitgestellt werden, wird im folgenden auf die Verwaltung von $n$ Instanzen der Klasse `SIPClient.java` als Empfänger und Erzeuger der Signalisierungsdaten über die Instanz der Klasse `SIPManager.java` näher eingegangen (vgl. Abb.3.9 'Klassendiagramm hob.phone'). Entsprechend den Darstellungen in Kap. 3.2.1 ist die Verwaltung und somit die Zuordnung der empfangenen SIP-Datenpakete zu den korrespondierten SIP-Clients in der Klasse `SIPManager.java` definiert, wobei im folgenden zunächst der Datenempfang dargestellt wird. Bevor eine Verarbeitung des Datenpaketes als SIP-Datenpaket erfolgt, werden die von einer Instanz der als Schnittstelle definierten Klasse `SIPConnection.java` empfangenen Daten durch die Klasse `SIPMessage.java` Interpretiert und als SIPMessage-Objekt der weiteren Verarbeitung übergeben. Die Klasse `SIPMessage.java` erwartet hierbei im Konstruktor die Rohdaten der Anwendungsschicht als Instanz der Klasse `DatagramPacket.java` und unterteilt sich in einen SIP-Header der durch die Klasse `SIPHeader.java` definiert wird und falls vorhanden in einen SDP-Teil, definiert durch die Klasse `SDPContent.java`. Im Konstruktor des `SIPMessage.java` werden zunächst allgemeine Werte festgelegt, wie z.B. SIP-Version oder ob es sich um eine SIP-Response oder einen SIP-Request handelt, was im nachfolgenden Ausschnitt des Quellcodes verdeutlicht wird:

**Listing 3.30:** SIPMessage.java -Konstruktor

```
if (str_sip.equals(Constants.SIP)) {
  im_bytePos++;/* seperator for version in SIP/2.0 is ->/<-*/
 .
```

```
 .
 .
} else {/* it is a request:METHOD|URI|SIPVERSION */
 bo_isRequest = true;
 .
 .
 .
}
```

Dabei wird die Notation des SIP für SIP-Responses zugrunde gelegt, nach der eine SIP-Response stets mit den Parametern Protokoll/Version, also bei einer SIP-Response konkret mit 'SIP/2.0' beginnt. Gleichen die ersten Zeichen des Datenpaketes somit nicht dem Schlüsselwort 'SIP', so handelt es sich um eine SIP-Request, die wiederum die Notation

'Methode URI Protokoll/Version'

hat, also konkret am Beispiel der SIP-Request

'INVITE username@domain.com SIP/2.0'

Ausgehend davon, dass eine SIP-Nachricht von einem SIP-Header und optional einem SDP-Content besteht, werden nach der in Kap. 2.1.1 dargestellten Notation für SIP die Daten interpretiert und Werte für Parameter in entsprechende Variablen hinterlegt, die durch die Klasse `SIPHeader.java` definiert sind. Eine Instanz der Klasse `SIPMessage.java` beinhaltet hierfür ein Objekt der Klasse `SIPHeader.java` als `SIPHeader header` und ggf. ein Objekt der Klasse `SDPContent.java` als `SDPContent content`, welche einen SDP-Teil definiert. Die Interpretation anhand der Notation der Daten des SIP-Header soll anhand dem nachstehenden Ausschnitt des entsprechendem Quellcodes verdeutlicht werden.

**Listing 3.31:** SIPHeader.java

```
while (!(str_pre.endsWith(": ")) \&\& im_byteCount < dgp_in.getLength()
             \&\& !(str_pre.equals("\r\n"))) {
      str_pre = str_pre + (char) dgp_in.getData()[im_byteCount];
      im_byteCount++;
}
```

Würde die Variable `im_byteCount` des Datentyps Integer die Nummer für das Element im Character-Array der Rohdaten für den ersten Buchstaben der Sequenz

From: "HOBPhone TEST«sip:81@172.22.20.194>;tag=114-1940908523-459 \r\n

des Character-Arrays repräsentieren, so erhielte man für die Variable `str_pre` des zusammengesetzten Datentyps `String` den Wert 'From:' nach fünf Durchläufen. Nach einer Unterscheidung des Typs als Parameter, wird im weiteren Verlauf einer dem Parameter entsprechend benannten Variable der entsprechende Wert rechts vom Trennzeichen zugeordnet. Ist eine Instanz der Klasse `SIPMessage.java` als `sip_msg` definiert, kann nun für das oben aufgeführte Beispiel mit dem Aufruf `sip_msg.header.str_from` auf den Wert für den Parameter 'From' zugegriffen werden, wobei sich je nach Parameter der Wert nach weiteren Unterscheidungen untergliedern kann. Der Wert für den Parameter 'From' im Bezug auf die oben aufgeführte Sequenz wird z.B. in nachstehende Elemente untergliedert:

- From_Fullname = 'HOBPhone Test'
- From_URI = 'sip:81'
- From_Tag = '114-1940908523-459'

Die Verarbeitung von SIP-Datenpaketen nach erfolgter Interpretation der Rohdaten erfolgt durch die Instanz der der Klasse `SIPManager.java` durch Aufruf der Methode `sipReceived`, welche als Parameter eine Instanz der Klasse `SIPMessage.java` erwartet. Zur Vereinfachung werden die Instanzen der Klassen `SIPClient.java`, `SIPManager.java` und `SIPConnection.java` im weiteren Verlauf als `sip_client`, `sip_mngr` und `sip_con` benannt.

Das in Abb. 3.9 dargestellte Klassendiagramm definiert hierzu jeder von $n$ möglichen `sip_con` je einen `sip_mngr`. Der Aufruf der Methode `sipReceived` des `sip_mngr` erfolgt über eine `sip_con`, welche Rohdaten empfängt und als `SIPMessage.java`-Objekt übergibt. Für die Definition des Vorgehens innerhalb der Methode `sipReceived` ist eine weitergehende Betrachtung der gegebenen Situation anhand der definierten Architektur des Konzeptes erforderlich. Demnach werden einem `sip_mngr` wiederum $n$ `sip_client`´s zugeordnet, wobei diese zu einer bestimmten `sip_con` über den `sip_mngr` in Beziehung stehen.

Anhand eines Beispieles soll nun eine gegebene Situation für den Aufruf der Methode `sipReceived` eines `sip_mngr` näher erläutert werden. Ein `sip_client` würde hierbei einen SIP-Client eines SIP-Accounts an einer Telefonanlage Asterisk repräsentieren. Gegeben sei der Fall, dass der Anwender über diesen `sip_client` jemanden anruft und somit einen Verbindungsaufbau mit SIP initiiert.

Nun es kann es vorkommen, dass während dem Dialog für den Verbindungsaufbau für diesen `sip_client` z.B. eine Auto-Registrierung seitens des HOBPhone erfolgt, da die bei der letzten Registrierung am Asterisk angegebene Gültigkeitsdauer für die derzeitige Registrierung überschritten ist. Demnach würden der Methode `sipReceived` aufeinander folgend SIP-Responses zum einen für den Verbindungsaufbau und zum anderen für den Registrierungsvorgang von der `sip_con` übergeben werden.

Würde anhand der empfangenen SIP-Response das weitere Vorgehen innerhalb einer Methode des `sip_client` erfolgen und die Methode `sipReceived` die SIP-Responses dieser übergeben, so müsste bei jeder empfangenen SIP-Response der Zusammenhang überprüft werden. Damit ausgehende Anfragen, oder allgemein Dialoge im SIP eines SIP-Clients voneinander unabhängig verarbeitet und allgemein definiert werden können, wird hierfür die Klasse `SIPDialog.java` definiert, dessen Instanzen im weiteren Verlauf als `sip_dialog` benannt werden.

Dia Klasse `SIPDialog.java` wird über einen `sip_client` immer dann instantiiert, wenn aufgrund einer aus- oder eingehenden SIP-Request die Generierung oder der Empfang mehrerer SIP-Requests oder SIP-Respones als ein SIP-Dialog zu erwarten ist. Dabei ist zu beachten, dass ein SIP-Dialog in diesem Sinne anhand einer ursprünglichen SIP-Request erzeugt wird, wobei der Erzeuger des SIP-Dialoges, der Erzeuger der SIP-Request ist.

Wie in Kap. 2.1.1.3 'SIP-Verbindungsaufbau' dargestellt wurde, hat jede SIP-Request eine eindeutige Call-ID, auf die sich alle SIP-Responses auf diese SIP-Request im SIP-Header beziehen müssen. Ein SIP-Request 'INVITE', der sich auf eine gültige Call-ID bezieht, stellt hierbei einen SIP-Re-Request bzw. einen SIP-Request 'RE-INVITE' dar. Eine Call-ID wird in diesem Zusammenhang solange als gültig definiert, bis der Dialog mit einer terminierenden SIP-Nachricht beendet ist. Eine Call-ID für einen ausgehenden SIP-Request 'INVITE' wird zum Beispiel mit Beendigung des Dialoges für diesen Verbindungsaufbau wieder ungültig.

Die Methode `sipReceived` ordnet somit primär eingehende SIP-Nachrichten als Instanz der Klasse `SIPMessage.java` nach der Call-ID ein. Ist die korrespondierte Call-ID gültig, so handelt es sich um eine SIP- Re-Request oder eine SIP-Response zu einem vorhergehenden SIP-Request und wird dem entsprechenden `sip_dialog` zur Verarbeitung übergeben. Ist die Call-ID der eingehenden SIP-Nachricht nicht vorhanden bzw. unbekannt, so darf es sich nur um eine SIP-Request handeln und somit um eine neue gültige Call-ID, denn eine SIP-Response muss immer mit einem SIP-Request korrespondieren, deren Call-ID nach dem beschriebenen Vorgang vorhanden sein müsste bzw. nach der Definition davon ausgegangen wird. Nachstehender Quellcode stellt die Methode `sipReceived(SIPMessage msg_in)` der Klasse `SIPManager.java` dar:

**Listing 3.32:** SIPManager.java -SIPManager.java -sipReceived(SIPMessage msg_in)

```
1 public void sipReceived(SIPMessage msg_in) {
2   if (msg_in.validVersion()) {
3     if (hmp_sipDialog.containsKey(msg_in.header.str_callID)) {
4       if (msg_in.isRequest()) {/*RE-REQUEST*/
5       ((SIPDialog) hmp_sipDialog.get(msg_in.header.str_callID))
6         .requestReceived(msg_in);
7       } else {/*RESPONSE*/
8       ((SIPDialog) hmp_sipDialog.get(msg_in.header.str_callID))
9               .responseReceived(msg_in);
```

```
10        }
11      } else {/* Call-ID not found */
12      if (msg_in.isRequest()) {/*NEW REQUEST*/
13        if (hmp_sipClient.containsKey(msg_in.header.str_toName)) {
14        ((SIPClient) hmp_sipClient.get(msg_in.header.str_toName))
15         receivedNewSIPRequest(msg_in);
16        } else {
17        Debug.err(this.getClass().toString(),
18          "SIPClient of a new Request not found: "
19          + msg_in.header.str_toName);
20        }
21      } else {
22      Debug.err(this.getClass().toString(),
23          "Call-ID of a response not found :"
24          + msg_in.header.str_callID + " Response: "
25          + msg_in.im_sipStatus);
26          }
27      }/* end else hmp_sipDialog.containsKey(CallID)*/
28    } else {
29    Debug.err(this.getClass().toString(), "Received INVALID SIPVERSION");
30    }
31  }
```

Die Instantiierung der Klasse `SIPDialog.java` erfolgt allein durch den `sip_client`, weshalb durch den `sip_mngr` einen SIP-Request der sich nicht auf eine gültige Call-ID bezieht und somit einen neuen Dialog darstellt, zunächst dem dazugehörigen `sip_client` übergeben wird, der dann einen neuen `sip_dialog erzeugt`. Nachdem für die richtige Zuordnung von SIP-Nachrichten zu $n$ SIP-Dialogen von $n$ `sip_client`´s der jeweilige `sip_mngr` mit der Methode `sipReceived` zuständig ist, werden dem `sip_mngr` zur Verwaltung entsprechend $n$ Objekte der Klassen `SIPClient.java` und `SIPDialog .java` bereitgestellt. Hierfür, wird durch jeden der $n$ `sip_client`´s jeder erzeugte neue `sip_dialog` in einem entsprechenden Container des `sip_mngr` hinterlegt, wobei der `sip_client` auch sich selbst bei seiner Erzeugung in einem entsprechenden Container hinterlegen muss. Als zur Verwaltung bestimmte Container werden Objekte der Klasse `java.util.HashMap` verwendet. Der Nachstehende Quellcode stellt den Konstruktor der Klasse SIPManager.java dar.

**Listing 3.33:** SIPManager.java

```
1
2 public SIPManager(SIPConnection sip_conp) {
3   hmp_sipDialog = new HashMap<String, SIPDialog>();/* storage for dialogs
        */
4   hmp_sipClient = new HashMap<String, SIPClient>();/* client storage */
5   this.sip_con = sip_conp;
6 }
```

Das beschriebene Vorgehen für eingehende SIP-Datenpakete durch die Mehtode `sipReceived` soll durch das in Abb. 3.11 dargestellte Aktivitätsdiagramm verdeutlicht werden.

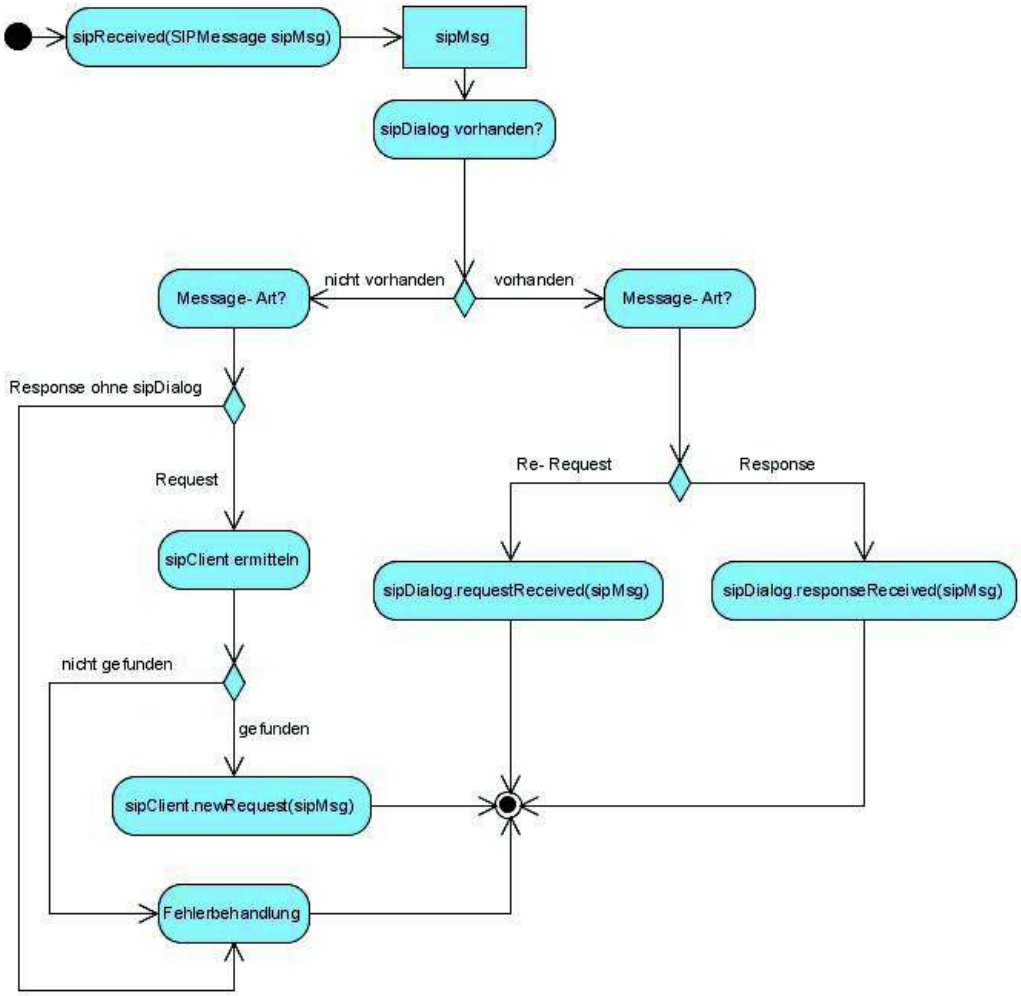


**Abbildung 3.11:** Aktivitätsdiagramm 'SIPManager.java -sipReceived()'

Soll der `sip_client` jeden erzeugten neuen `sip_dialog` in einem Container des `sip_mngr` hinterlegen, ist eine Erfassung für möglichen Situationen für eine Erstellung eines `sip_dialog` erforderlich. Hierbei kommen zwei Situationen in Frage, eine eingehende SIP-Request, die nicht einer gültigen Call-ID zugeordnet werden kann oder eine Handlung des Benutzers. Zunächst soll der Fall eines eingehenden neuen SIP-Request näher betrachtet werden. Ausgehend von der Definition eines SIP-Dialoges in diesem Kontext, ist die einzige Mögliche Situation für einen eingehenden neuen SIP-Request, der einen neuen SIP-Dialog darstellt, der SIP-Request 'INVITE', also ein eingehender Anruf. SIP-

Requests, wie z.B. 'ACK' oder 'BYE' müssen sich nach dem SIP mittels Ihrer Call-ID auf eine vorhergehende SIP-Request, also einer 'INVITE' beziehen und stellen somit eine SIP-'Re-Request' dar. Bei verbleibenden SIP-Requests, wie z.B. 'OPTIONS', findet kein Dialog ist diesem Sinne statt, so dass diese SIP-Requests mit einer SIP-Response terminiert werden, weshalb auch kein neuer `sip_dialog` erzeugt wird, sondern diese über den `sip_client` verarbeitet werden. Der Ausschnitt aus dem Quellcode der Klasse `SIPClient.java` stellt einen Teil der Methode `receivedNewSIPRequest(SIPMessage msg_in)` dar, die seitens des `sip_mngr` über die Methode `sipReceived`, wie aus dem in 3.11 dargestellten Aktivitätsdiagramm ersichtlich ist, immer dann gerufen wird, wenn ein eingehender SIP-Request keiner vorhanden Call-ID zugeordnet werden kann und somit einen neuen SIP-Request darstellt.

**Listing 3.34:** SIPClient.java -receivedNewSIPRequest

```
1  public void receivedNewSIPRequest(SIPMessage msg_in) {
2    switch (Constants.getKey(msg_in.str_phoneStatus)) {
3    case Constants.byf_INVITE:
4    // TODO:send "305"Use Proxy when INVITE coming from wrong addr in
       TCPmode
5    if (msg_in.header.str_callID != null) {
6      if (phone_mngr.isActiveClient(this)) {
7      /* Check if an INVITE Dialog already exists */
8      if (!bo_invite) {
9        SIPDialog.get().setNewDialog(this, msg_in);
10     } else {
11     /*
12      * INVITE-Dialog already exists: Re-INVITE not possible here
13      * as SIPManager will redirect to existing Call-ID as re-invite
14      * MUST use Call-ID of origin INVITE. Handle as new incoming Call.
15      * while an INVITE-Dialog exists.
16      */
17         if (bo_callWaiting)
18          }
19       }
20     }
21   }
22 }
```

Der Aufruf in Zeile 9 für den Fall einer empfangenen SIP-Request 'INVITE' erfolgt im Falle eines eingehenden SIP-Request immer dann, wenn der SIP-Client im Moment nicht einen gültigen Dialog für eine 'INVITE'-Request führt. Das ist z.B. dann der Fall, wenn während eines Gespräches oder einem Verbindungsaufbau ein anderer Anruf eingeht. Hier ist eine gesonderte Behandlung erforderlich, die sich aus der Funktionalität und den individuellen Benutzereinstellungen des SIP-Clients ergibt. Ist zum Beispiel die Funktionalität 'Makeln' oder 'Anklopfen' erwünscht bzw. konfiguriert, so würde der `sip_client` zwar einen weiteren Dialog erzeugen, müsste aber gleichzeitig einen Mechanismus in Gang setzten, der aktive Gespräche in den Status 'hold' versetzt. Dies

kann ferner in der Konfiguration des SIP-Accounts vorgenommen werden, wodurch Asterisk die Übertragung von Sprachdatenpaketen des aktiven Gespräches mittels eines 'RE-INVITE' auf sich 'umlenkt' und der wartenden Gegenstelle für die Dauer des Status 'hold' z.B. Sprachdatenpakete in Form von Wartemusik sendet (vgl. Kap. 2.2 'Die Telefonanlage Asterisk'). Durch den erzeugten neuen `sip_dialog` für den eingehenden weiteren Anruf, kann der in den Status 'hold' zu versetzende `sip_dialog` völlig unabhängig von diesem Vorgang behandelt werden.

Der Benutzer muss bei Verwendung von mehreren SIP-Accounts über die Benutzerschnittstelle immer einen SIP-Client als den aktiven einstellen bzw. kennzeichnen, was über die Klasse `PhoneManager.java` verwaltet wird. Eingehende Anrufe für den aktiven SIP-Client werden unmittelbar durch den Status 'Ringing' angezeigt und akustisch durch einen Klingelton bemerkbar gemacht, wobei ausgehende Anrufe automatisch über den jeweiligen SIP-Account des aktiven SIP-Client erfolgen. Der Mechanismus für 'hold' wird in diesem Zusammenhang auch dann verwendet, wenn der angerufene SIP-Client nicht aktiv bzw. selektiert ist.

Die Funktionalität des 'hold' ist jedoch in dieser Version des Konzeptes nicht implementiert worden und wird in Kap. 4 'Fazit und Ausblick' als weiteres Feature aufgeführt. Demnach werden in dieser Situation eingehende Anrufe für nicht Aktive SIP-Clients oder bei einem laufenden Gesprächsaufbau bzw. Gespräch mittels der SIP-Response 'Busy Here' abgewiesen. Ausgehend von den beiden Möglichkeiten die bei einer eingehenden SIP-Request in Frage kommen, soll nun, abgesehen von einer eingehenden SIP-Request die nicht einer gültigen Call-ID zugeordnet werden kann und eine neue SIP-Request darstellt, eine mögliche Handlung des Benutzers näher betrachtet werden. Als Handlungen des Benutzers werden in diesem Kontext folgende Möglichkeiten betrachtet:

- eingehenden Anruf annehmen (ACCEPT)
- eingehenden Anruf ablehnen (DECLINE)
- ausgehenden Anruf initiieren (DIAL)
- Gespräch beenden (HANGUP)

Wie bereits weiter oben definiert wurde, wird im Falle eines eingehenden Anrufes ein neuer `sip_dialog` erzeugt und dem `sip_mngr` zur Verwaltung übergeben. Demnach setzt die Handlung 'Anruf annehmen' oder 'Anruf ablehnen' eine gültige Call-ID voraus. Ebenfalls, setzt die Handlung 'Auflegen' eine gütlige Call-ID insofern voraus, dass, unabhängig davon, ob es sich um einen eingehenden oder ausgehende Anruf handelt, ein `sip\_dialog` vorhanden sein muss. Allein die Handlung 'Anrufen' ist somit die einzige Möglichkeit, einen neuen `sip_dialog` mit einer neuen Call-ID zu initiieren. Nachstehend wir ein Ausschnitt der Methode `userAction()` des `sip\_client` dargestellt, die über den `sip_client` einer Instanz der Klasse `PhoneUserInterface.java` zur Verfügung steht und bei einer Handlung mit den entsprechenden Parametern gerufen wird.

**Listing 3.35:** SIPClient.java -userAction

```
public void userAction(int imp_actionKey, String strp_callID,
    String strp_callTo) {
  if (strp_callID == null) {
    SIPDialog.get().setNewDialog(this, imp_actionKey, strp_callTo);
  } else {
    if (sip_mngr.hmp_sipDialog.containsKey(strp_callID)) {
    ((SIPDialog) sip_mngr.hmp_sipDialog.get(strp_callID)).userAction(
            imp_actionKey, strp_callTo);
    } else {
      /* TODO:invalid user action from GUI: missing callID */
      Debug.err(this.getClass().toString(), "[" + getSettingName()
        + "] Invalid user action from GUI: callID not found:"
        + strp_callID);
    }
  }
}
```

Somit können der Methode `userAction()` wahlweise eine Call-ID oder eine Information für ein zu Erreichendes Ziel in Verbindung mit einer Schlüsselnummer für die auszuführende Handlung übergeben werden. Wird keine Call-ID übergeben, so wird von einem neuen ausgehenden Anruf ausgegangen und die Erzeugung einer neuen Instanz der Klasse `SIPDialog.java` veranlasst (Zeile 4, Listing 3.2.3.1). Zu beachten ist der Unterschied der übergebenen Parameter bei der Erzeugung eines neuen `sip_dialog`. Bei einem eingehenden Anruf innerhalb der Methode `receivedNewSIPRequest(SIPMessage msg_in)` des `sip_client` wird der Mehtode `setNewDialog` der empfangene SIP-Request übergeben, wobei bei einem ausgehenden Anruf die Zielinformation `strp_callTo` übergeben wird. Durch den Aufruf der überschriebenen Methode mittels der Zielinformation, wird eine neue Call-ID generiert , während bei Übergabe einer empfangenen SIP-Request 'INVITE' die Call-ID der empfangenen SIP-Request übernommen wird.

Die Erzeugung und Übergabe des `sip_dialog` Objektes wird hierzu der Klasse `SIPDialog .java` selbst überlassen. Aufgrund dessen, dass im weiteren Verlauf für die Kommunikation innerhalb des erzeugten Dialoges der `sip_dialog` unabhängig und selbstständig agieren soll, wird durch den `sip_client` über die Methode `setNewDialog` der Dialog in diesem Sinne 'angestoßen', dann aber sich selbst überlassen. Im nachstehenden Quellcode werden die unterschiedlichen Konstruktoren und die überschriebene Mehtode `setNewDialog` der Klasse `SIPDialog.java dargestellt`.

**Listing 3.36:** SIPDialog.java -setNewDialog

```

  private SIPDialog(SIPClient sip_clientp, SIPMessage msg_in) {
    init(sip_clientp, msg_in);
  }

  private SIPDialog(SIPClient sip_clientp, int imp_actionKey,
```

```
7       String strp_callTo) {
8     init(sip_clientp, imp_actionKey, strp_callTo);
9   }
10
11  public void setNewDialog(SIPClient sip_clientp, SIPMessage msg_in) {
12    new SIPDialog(sip_clientp, msg_in);
13  }
14
15  public void setNewDialog(SIPClient sip_clientp, int imp_actionKey,
16      String strp_callTo) {
17    new SIPDialog(sip_clientp, imp_actionKey, strp_callTo);
18  }
```

Bezieht sich eine Handlung des Benutzers auf einen bereits existierenden `sip_dialog`, so wird durch

```
((SIPDialog)sip_mngr.hmp_sipDialog.get(strp_callID)).userAction(imp_actionKey
 , strp_callTo);
```

innerhalb der Methode `userAction` des `sip_client` der entsprechende `sip_dialog` über den Container des `sip_mngr` mittels der Call-ID angesteuert. Eingehende `sip_message` am `sip_mngr` werden, sofern sich diese auf einen `sip_dialog` beziehen bzw. eine gültige Call-ID haben, nach dem gleichen Prinzip weitergeleitet, wie der nachstehende Ausschnitt aus der Mehtode `sipReceived` des `sip_mngr` verdeutlicht:

```
((SIPDialog)hmp_sipDialog.get(msg_in.header.str_callID)).responseReceived(msg
 _in);
```

Das Versenden von SIP-Datenpaketen erfolgt über entsprechende Funktionen, welche die Klasse `SIPManager.java` bereitstellt. Hierbei wird je eine Funktion für das Versenden eines SIP-Requests und einer SIP-Response bereitgestellt. Die Funktion `sendRequest` mit dem boolean-Rückgabewert, der den Erfolg des Sendevorganges ausdrückt, erwartet als Parameter den versendenden `sip_client`, eine Schlüsselnummer für die Nachrichtenart und den korrespondierenden `sip_dialog`. Der nachstehende Quellcode stellt die Funktion `sendRequest` der Klasse SIPManager dar:

**Listing 3.37:** SIPManager.java -sendRequest

```
1
2 public boolean sendRequest(SIPClient sip_client, int imp_msgKey, SIPDialog
      sip_dialog) {
3     byte[] byr_msg = prepareRequest(sip_client, imp_msgKey,sip_dialog);
4          if ( byr_msg != null) {
5                     return sip_con.sendSIP(byr_msg,0,byr_msg.length);
6
7          } else {
8              return false;
9          }
10 }
```

Wie aus dem Quellcode ersichtlich, wird durch Aufruf der Funktion `prepareRequest` innerhalb der Funktion ein byte-Array zurückgegeben, dass dann über die `sip_con` mittels der Funktion `sendSIP` an das Ziel als SIP-Datenpaket versendet wird und gleichzeitig den Rückgabewert der Funktion `sendRequest` als Erfolg des Sendevorganges darstellt. Die Generierung der Nachricht mit den gegebenen Informationen in der Art der zu generierenden Nachricht und dem Erzeuger erfolgt schließlich durch die Funktion `prepareRequest`. Hierbei wird je nach Art der SIP-Request entsprechend den Vorgaben für das SIP eine jeweilige Vorlage auf Basis der in Kap. 2.1.1.2 beschriebenen Notation für SIP verwendet, die anhand der übergebenen Informationen angepasst wird.

Die übergebene Schlüsselnummer für die Art der SIP-Request ist eine in der statischen Hilfs-Klasse `Consants.java` definierte Zahl für die verschiedenen SIP-Requests, welche über eine `switch`-Konstruktion unterschieden werden. Im nachstehenden Ausschnitt des Quellcodes der Funktion `prepareRequest`, soll die Generierung einer SIP-Request für die Schlüsselnummer `Constants.byf_INVITE`, also einer SIP-Request 'INVITE' verdeutlicht werden.

**Listing 3.38:** SIPManager.java -case INVITE

```
1
2  case Constants.byf_INVITE:
3    if (sip_client.mp_digest.get(Constants.HMP_NONCE) != null) {
4    str_digest = "Proxy-Authorization: Digest "
5        + createDigestHeader(sip_client.str_userName,
6        sip_client.str_password, sip_client.mp_digest) + "\r\n";
7    } else {
8      str_digest = "";
9    }
10   try {
11
12   byr_msg = (METHOD + " sip:" + strp_to + "@" + strp_host_to + " "
13   + Constants.strf_sip_vers + "\r\n"
14   + sip_client.str_via+ ";rport;branch=" + sip_client.str_branch + "\r\n"
15   + sip_client.str_from + strp_fromTag + "\r\n"
16   + "To: <sip:"+ strp_to + "@" + str_localIP + ">" + strp_toTag + "\r\n"
17   + "Contact: " + sip_client.str_contact + "\r\n"
18   + "Call-ID: "+ strp_callID + "\r\n"
19   + "CSeq: " + imp_cSeq + " " + METHOD+ "\r\n"
20   + "User-Agent: " + Constants.strf_ua + "\r\n"
21   + str_digest
22   + "Content-Type: application/sdp\r\n"
23   + "Content-Length: " + strp_sdpContent.length() + "\r\n"
24   + "\r\n"
25   + strp_sdpContent).getBytes(str_encoding);
26   } catch (UnsupportedEncodingException uee) {
27   Debug.err(this.getClass().toString(), "["
28   + sip_client.getSettingName() + "] Unsupported Encoding:"
29   + uee.getMessage());
```

```
30   }
31
32 break;
```

Die Generierung von SIP-Responses erfolgt mittels der Funktion `prepareResponse` nach dem gleichen Verfahren, außer dass die Vorlage der Notation für eine SIP-Response entspricht Das letztendliche 'Verhalten ' des `sip_client`, im Sinne von wann und wie auf welche SIP-Nachricht oder Benutzerhandlung reagiert wird, ergibt sich aus den Methoden für den Empfang von SIP-Datenpaketen und die Behandlung von Benutzerhandlungen, die die Klasse `SIPClient.java` bereitstellt, wobei ein Großteil der Handlungen in die erwähnte Konzeption der SIP-Dialoge ausgelagert werden. Der nachstehende Ausschnitt des Quellcodes stellt einen Teil der Methode `responseReceived` eines `sip_dialog` dar. Bei einer empfangenen SIP-Response wird hierbei in einer `switch`-Konstruktion die empfangene SIP-Response nach der allgemeinen Schlüsselnummer für SIP-Responses für das weitere Vorgehen unterschieden, wobei im folgenden der Fall der SIP-Response '407 Proxy Authentication Required' näher betrachtet werden soll.

**Listing 3.39:** SIPDialog.java -case 407 Proxy Authentication Required

```
1  case 407:
2    // "407" ; Proxy Authentication Required
3    String str_nonce = sip_client.mp_digest.get(Constants.HMP_NONCE);
4    boolean bo_nonceSend = false;
5    if (str_nonce != null) {
6      if (str_nonce.equals(msg_in.header.str_nonce)) {
7      // this authentication was already done, duplicate 407
8        bo_nonceSend = true;
9      }
10   }
11   if (!bo_nonceSend) {
12     setPhoneStatus(new PhoneStatusEvent(msg_in.im_sipStatus, phone_mngr
13         .getCurrentStatusKey(), str_callID, phone_mngr
14         .getCurrentInfo()));
15     if (sip_mngr.sendRequest(sip_client, Constants.byf_ACK, this)) {
16       sendInvite(msg_in.header.str_nonce, msg_in.header.str_realm);
17     }
18   }
19 break;
```

Ausgehend von der Definition für SIP, stellt die SIP-Response mit der Schlüsselnummer 407 eine Aufforderung dar, dass der Erzeuger eines SIP-Requests 'INVITE', im nachstehenden Beispiel das HOBPhone, sich zunächst am SIP-Proxy, hier die Telefonanlage Asterisk, authentifizieren soll. Die Sequenz des Dialoges wird in der nachstehenden Konsolenausgabe dargestellt.

**Listing 3.40:** SIPRequest INVITE -SIPRespone 407

```
1  INVITE sip:80@172.22.20.194 SIP/2.0
```

```
2  Via: SIP/2.0/UDP 172.22.80.140:1339;rport;branch=z9hG4bK695CEDB4
3  From: "HOBPhone TEST"<sip:81@172.22.20.194>;tag=118-1940908509-9498
4  To: <sip:80@172.22.80.140>
5  Contact: <sip:81@172.22.80.140:1339;transport=UDP>
6  Call-ID: 14-85-120-96-1354172699@172.22.80.140
7  CSeq: 1 INVITE
8  User-Agent: HOBPhone_v0.1
9  Content-Type: application/sdp
10 Content-Length: 180
11
12 v=0
13 o=HOBPhone_Client 28903 28903 IN IP4 172.22.80.140
14 s=HOBPhone_Session
15 c=IN IP4 172.22.80.140
16 t=0 0
17 m=audio 1342 RTP/AVP 3 0
18 a=rtpmap:3 GSM/8000
19 a=rtpmap:0 PCMU/8000
20
21 SIP/2.0 407 Proxy Authentication Required
22 Via: SIP/2.0/UDP 172.22.80.140:1339;branch=z9hG4bK695CEDB4;received
       =172.22.80.140;rport=1339
23 From: "HOBPhone TEST"<sip:81@172.22.20.194>;tag=118-1940908509-9498
24 To: <sip:80@172.22.80.140>;tag=as023c1de8
25 Call-ID: 14-85-120-96-1354172699@172.22.80.140
26 CSeq: 1 INVITE
27 User-Agent: Asterisk PBX
28 Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY
29 Proxy-Authenticate: Digest algorithm=MD5, realm="asterisk", nonce="062b50e
       7"
30 Content-Length: 0
```

Wie in Kap. 2.1.1.3 'Verbindungsaufbau mit SIP' dargestellt, wird der Empfang der Aufforderung seitens des Aufgeforderten zunächst mit der SIP-Request 'ACK' bestätigt. Ausschlaggebend für das weitere Vorgehen ist hierbei die Zeile mit dem Parameter 'Proxy-Authenticate' der oben dargestellten Konsolenausgabe der Aufforderung. Demnach muss der ursprüngliche SIP-Request 'INVITE' nach der Bestätigung der Empfangs der Aufforderung um die Zeile bzw. dem Parameter 'Proxy-Authorization' mit den erforderlichen Werten für die Authentifizierung anhand der Werte im empfangenen Parameter 'Proxy-Authenticate' erweitert und nochmals als SIP-Request 'INVITE' versendet werden (vgl. Listing 3.2.3.1).

Die Funktion `sendInvite` des `sip_dialog` ist hierbei eine vorbereitende Vorstufe für den eigentlichen Versand mittels des Aufrufes `sendReqest` über den `sip_mngr`, was im nachstehend Quellcode der Funktion `sendInvite` dargestellt wird.

**Listing 3.41:** SIPDialog.java -sendInvite

```
1 public boolean sendInvite(String str_nonce, String str_realm) {
```

```
2
3 try {
4   imp_cSeq++;
5   sip_client.mp_digest.put(Constants.HMP_URI, "sip:" + str_callTo + "@"
6   + sip_client.getSIPHost());
7   sip_client.mp_digest.put(Constants.HMP_NONCE, str_nonce);
8   sip_client.mp_digest.put(Constants.HMP_REALM, str_realm);
9   sip_client.mp_digest.put(Constants.HMP_METHODNAME,
10  Constants.strf_INVITE);
11
12  if (sip_mngr.sendRequest(sip_client, Constants.byf_INVITE, this)) {
13    return true;
14  }
15 } catch (Exception e) {
16  Debug.err(this.getClass().toString(), "Cannot send INVITE! err:");
17  e.printStackTrace();
18  setPhoneStatus(new PhoneStatusEvent(Constants.imf_SERVICE_UNAVAILABLE,
19  Constants.byf_OFFLINE, str_callID, ""));
20
21 }
22 return false;
23 }
```

Hierbei wird ein Container des korrespondierten `sip_client` mit den empfangenen Werten des Parameters 'Proxy-Authenticate' gefüllt, den der `sip_mngr` zur Generierung des erforderlichen MD5-Hash und dem benötigten Parameter 'Proxy-Authorization' heranzieht. Die Konsolenausgabe für den generierten zweiten SIP-Request auf die Aufforderung zur Authentifizierung stellt sich wie folgt dar:

**Listing 3.42:** Konsolenausgabe INVITE-Message

```
1 INVITE sip:80@172.22.20.194 SIP/2.0
2 Via: SIP/2.0/UDP 172.22.80.140:1339;rport;branch=z9hG4bK695CEDB4
3 From: "HOBPhone TEST"<sip:81@172.22.20.194>;tag=118-1940908509-9498
4 To: <sip:80@172.22.80.140>;tag=as023c1de8
5 Contact: <sip:81@172.22.80.140:1339;transport=UDP>
6 Call-ID: 14-85-120-96-1354172699@172.22.80.140
7 CSeq: 2 INVITE
8 User-Agent: HOBPhone_v0.1
9 Proxy-Authorization: Digest username="81", realm="asterisk", nonce="062b50
      e7", uri="sip:80@172.22.20.194", response="ba3167b02c539b1441764e
      0475963a70", opaque="", algorithm="MD5"
10 Content-Type: application/sdp
11 Content-Length: 180
12
13 v=0
14 o=HOBPhone_Client 28903 28903 IN IP4 172.22.80.140
15 s=HOBPhone_Session
16 c=IN IP4 172.22.80.140
17 t=0 0
```

```
m=audio 1342 RTP/AVP 3 0
a=rtpmap:3 GSM/8000
a=rtpmap:0 PCMU/8000
```

#### 3.2.3.2 Verarbeitung von WSP-Signalisierungsdaten

Die Verarbeitung von WSP-Signalisierungsdaten erfolgt auf Basis des in Kap. 3.1.3 dargestellten WSP-Protokoll und wird definiert durch die Klasse `WSPManager.java`, wobei zu beachten ist, dass eine mögliche Verarbeitung von WSP-Signalisierungsdaten ausschließlich im SSL-Modus in Frage kommt. Demnach wird eine Instanz der Klasse `WSPManager.java` nach Aufbau der SSL-Verbindung mit der Channel-Nummer '0' für WSP-Signalisierungsdaten erstellt, der alle Datenpakete die mit dieser Channel-Nummer korrespondieren zur Verarbeitung übergeben werden. Die Klasse `WSPManager.java` implementiert hierbei das Interface `WSPConnectionListener.java` und somit die Methode `receivedData`:

**Listing 3.43:** WSPManager.java -receivedData

```
public void receivedData(byte[] byrp_data, int imp_offset, int imp_len) {
controlReceived(new WSPControlMessage(byrp_data,imp_offset,imp_len));
}
```

Wie aus dem oben dargestelltem Quellcode ersichtlich, werden eingehende Datenpakete als Byte-Array der internen Methode `controlReceived` übergeben. Innerhalb der Methode `controlReceived`, werden die empfangenen Informationen anhand eines internen Status interpretiert. Der interne Status richtet sich nach dem in Kap. 3.1.3 dargestellten Dialogen zwischen dem HOBPhone und dem SDH und wird unterschieden in

- Verbindungsaufbau
- Channel erzeugen
- Channel parametrisieren
- Channel entfernen

Die Klasse `WSPManager.java` setzt in ihrem Konstruktor bei Erzeugung einer Instanz den Status auf 'Verbindungsaufbau'. Der SDH versendet nach Aufbau der SSL-Verbindung anhand der Administration der SIP-Benutzerdaten am WSP die erforderlichen SIP-Benutzerdaten des HOBPhone mittels des WSP-Protokoll für n SIP-Account an n möglichen Asterisk. Nach Aufbau der SSL-Verbindung werden die SIP-Benutzerdaten für jeden Account in nachstehender Form vom SDH generiert und dem HOBPhone versendet:

REGISTER
config-name:test1

channel:1
SIP-Ident:11
INETA-SIP-gateway:172.5.0.1
SIP-shared-secred:geheim
INETA-this:172.5.0.1

Nachstehender Ausschnitt aus dem Quellcode der `switch`-Kontruktion für den Status der Methode `controlReceived` stellt die Verarbeitung von empfangenen Signalisierungsdaten bzw. der oben Dargestellten Control-Request `REGISTER` im Status 'Verbindungsaufbau' dar.

**Listing 3.44:** WSPManager.java -controlReceived

```
case Constants.WSPSTATE_CONNECT:
  if (msg.isCommand()) {
  if (this.bof_WSPDEBUG) {
  Debug.out(this.getClass().toString(),
  "CONTROL-COMMAND Received: " + msg.getCommand());
  }
  if (msg.getCommand().equalsIgnoreCase(Constants.WSP_REGISTER))
 {
  HashMap<Integer, String> hmp_registry = new HashMap<Integer, String>();
    String str_fullName = null;
    String str_SIPIdent = null;
    String str_SIPGW = null;
    String str_SIPshase = null;
    String str_settingsName = null;
    String str_channel = null;
    boolean bo_valid = false;
    str_settingsName
 = msg.getSettingsName();
    if (validValue(str_settingsName)) {
      str_fullName = msg.getFullname();
      if (!(validValue(str_fullName))) {
        str_fullName = "unknown";
      }
      str_SIPIdent = msg.getSIPIdent();
      if (validValue(str_SIPIdent)) {
        str_SIPGW = msg.getTargetINETA();
        if (validValue(str_SIPGW)) {
        str_SIPshase = msg.getSIPSharedSecret();
          if (validValue(str_SIPshase))
 {
          String str_localINETA = msg.getLocalINETA();
          if (validValue(str_localINETA)) {
          wsp_con.setLocalINETA(str_localINETA);
          wsp_con.setSIPGatewayINETA(str_SIPGW);
          str_channel = msg.getChannel();
          if (validValue(str_channel)) {
          bo_valid = true;
```

```
38          } else {
39          Debug.err(this.getClass().toString(),
40          "REGISTER: SIPCHANNEL invalid.");
41          }
42        } else {
43        Debug.err(this.getClass().toString(),
44        "REGISTER: SIPLOCALINETA invalid.");
45        }
46      } else {
47      Debug.err(this.getClass().toString(),
48      "REGISTER: SIPSharedSecret invalid.");
49      }
50    } else {
51    Debug.err(this.getClass().toString(),"REGISTER: SIPGW invalid.");
52    }
53  } else {
54  Debug.err(this.getClass().toString(),"REGISTER: SIPIdent invalid.");
55  }
56 } else {
57  Debug.err(this.getClass().toString(),"REGISTER: SETTINGSNAME invalid.");
58 }
59 }}
```

An dieser Stelle ist zu erwähnen, dass in dieser Version das HOBPhone die in Kap. 3.1 'Architektur des Konzeptes' beschriebene Funktionalität zur Unterstützung von $n$ SIP-Accounts an $n$ möglichen Telefonanlagen zwar unterstützt bzw. dementsprechend implementiert wird, im SSL-Modus bei der Implementierung des SDH jedoch zunächst von einem SIP-Account an einer Telefonanlage Asterisk ausgegangen wird. Bei erfolgreicher Interpretation der übermittelten Informationen wird im Anschluss eine Benutzerhandlung in Form von Eingabe der empfangenen Daten über die Benutzerschnittstelle simuliert:

**Listing 3.45:** WSPManager.java -registryReceived

```
1 if (bo_valid) {
2   hmp_registry.put(Constants.HMP_FULLNAME, str_fullName);
3   hmp_registry.put(Constants.HMP_USERNAME, str_SIPIdent);
4   hmp_registry.put(Constants.HMP_HOST, str_SIPGW);/* SIPGW */
5   hmp_registry.put(Constants.HMP_PWD, str_SIPshase);
6   hmp_registry.put(Constants.HMP_CHANNEL, str_channel);
7   /* ALL REGISTRY DATA PARSED */
8   ui.createClient(str_settingsName);
9   ui.setSIPClient(str_settingsName, hmp_registry, true);
10  if(bo_autoRegister)
11  ui.ua_register(str_settingsName);
12 }// all error done
```

Das mit `ui` bennante Objekt repräsentiert hierbei die Benutzerschnittstelle als Instanz der Klasse `PhoneUserInterface.java`, welche der Klasse `WSPManager.java` mit überge-

ben wird. Im Vergleich dazu wird nachstehend der Vorgang dargestellt, wie er bei der manuellen Eingabe der Benutzerdaten im UDP-Modus verläuft:

**Listing 3.46:** PhoneUserInterface.java -btnSubmitPressed

```
private void btnSubmitPressed() {
  boolean bo_access = false;
  String str_fullName = null;
  String str_user = null;
  String str_host = null;
  str_fullName = txt_fullName.getText();
  if (!(str_fullName != null && str_fullName.length() > 0)) {
    str_fullName = "unknown";
  }
  str_user = txt_user.getText();
  if (str_user != null && str_user.length() > 0) {
    str_host = txt_host.getText();
    if (str_host != null && str_host.length() > 0) {
      String str_pwd = "";
      char[] chr_pw = pwf_auth.getPassword();
      if (chr_pw != null && chr_pw.length > 0) {
        for (int i = 0; i < chr_pw.length; i++) {
          str_pwd = str_pwd + chr_pw[i];
          chr_pw[i] = '\\0';
        }
    hmp_reg_in.put(Constants.HMP_FULLNAME, txt_fullName.getText());
    hmp_reg_in.put(Constants.HMP_USERNAME, this.txt_user.getText());
    hmp_reg_in.put(Constants.HMP_HOST, txt_host.getText());
    hmp_reg_in.put(Constants.HMP_PWD, str_pwd);
        bo_access = true;
      } else {
    JOptionPane.showMessageDialog(null, "Please enter a password.");
    }
    } else {
  JOptionPane.showMessageDialog(null, "Please enter a valid SIP-host.");
    }
  } else {
  JOptionPane.showMessageDialog(null, "Please enter an username.");
  }
  if (bo_access) {
  String str_clientName = this.txt_user.getText() + "@"
  + this.txt_host.getText();
  gui.createClient(str_clientName);
  gui.setSIPClient(str_clientName, hmp_reg_in, false);
  this.setVisible(false);
  }
}
```

Der Vorgang unterteilt sich in Anlegen und Parametrisieren einer Instanz der Klasse SIPClient.java mittels der Methoden `createClient` und `setSIPClient` (vgl. Listing

3.2.2.1 `SIPClient.java -setClient`) sowie der Methode `ua_register` zum Einleiten des Registrierungsdialoges für den SIP-Account mit der entsprechenden Telefonanlage Asterisk, wobei die Registrierung je nach individueller Benutzerkonfiguration separat oder automatisiert nach Empfang bzw. Eingabe der Benutzerdaten erfolgen kann (Z. 8-11 Listing 3.2.3.2).

Nach dem gleichen Prinzip im Bezug auf den Vorgang in Listing 3.2.3.2 erfolgt die Portanforderung und Parametriesierung über den SDH. Hierzu sollen die in Listing 3.1.3 und 3.1.3 dargestellten Dialoge `CREATE` und `SET` zwischen dem HOBPhone und SDH näher betrachtet werden. Der Vorgang bei dem am WSP ein Port für eine bevorstehende RTP-Datenübertragung 'reserviert' wird, um diese Portnummer im SDP-Teil in entsprechenden SIP-Nachrichten aufzuführen, wird im Aktivitätsdiagramm 3.12 durch den Dialog 'CREATE CHANNEL' mit der Antwort 'PORT' dargestellt. Hierbei wird die Mehtode `setRTPConnection` der abstrakten Klasse `RTPConnection.java` gerufen. Während der Aufruf im UDP-Modus als Antwort über die Funktion `receivedRTPLocalConnection` einen lokalen Port bereitstellt, erfolgt im SSL-Modus der in Kap. 3.1.3 dargestellte Dialog zwischen dem HOBPhone und dem SDH. Nachstehender Ausschnitt aus dem Quellcode der Klasse `RTPConnectionWSP.java` veranschaulicht, wie der Dialog beim Aufruf der Mehtode `setRTPConnection` eingeleitet wird.

**Listing 3.47:** RTPConnectionWSP.java -setRTPConnection

```
1 public void setRTPConnection() {
2     wsp_con.getWSPManager().createChannel(imf_RTPChannel);
3     /* by adding listener received RTP data from TCP-Stream can be
          assigned */
4 }
```

Demnach wird die Mehtode `createChannel` der Klasse `WSPManager.java` gerufen, welche im nachstehenden Quellcode dargestellt wird:

**Listing 3.48:** WSPManager.java -createChannel

```
1  public void createChannel(int imp_create_channel) {
2    im_wspState = Constants.WSPSTATE_CREATE;
3    String msg_out = null;
4    WSPControlMessage msg = new WSPControlMessage(Constants.WSP_CREATE,
5    imp_create_channel);
6    msg_out = msg.getMessage();
7    if (msg_out != null) {
8      if (bof_WSPDEBUG)
9      Debug.out(this.getClass().toString(),
10     "jbthvoipc1-l00000FI;-T sending CONTROL: CREATE");
11     wsp_con.send(Constants.WSP_CONTROL_CHANNEL, msg_out.getBytes(),
12     msg_out.length());
13   } else {
14     Debug.err(this.getClass().toString(),
15     "Cannot send CONTROL-Msg:CREATE channel:" + imp_create_channel);
```

```
16   }
17 }
```

Die Mehtode `createChannel` der Klasse `WSPManager.java` erzeugt hierbei eine Instanz der Klasse `WSPControlMessage.java`, welche den Aufbau einer Control-Message nach der in Kap. 3.1.3 dargestellten Notation definiert. Der Control-Message 'CREATE' veranlasst beim SDH einen Port-Bind, worauf als Antwort die Control-Message '200 OK' mit der entsprechenden Port-Nummer an das HOBPhone versendet wird. Ist die am WSP zur verfügung stehende Port-Nummer für eine bevorstehenden RTP-Datenübertragung mit der VoIP-Archtiektur bekannt, so wird am WSP noch die Information über das Ziel der RTP-Datenübertragung benötigt (vgl. Kap. Kap. 3.1.3).

Nachstehender Quellcode veranschaulicht den im Aktivitätsdiagramm 3.12 durch den Dialog 'SET CHANNEL' mit der Antwort '200 OK' dargestellten Ablauf wie er in der Klasse `WSPManager.java` eingeleitet wird.

**Listing 3.49:** WSPManager.java -setChannel

```
1  public void setChannel(int imp_set_channel, String str_INETA_target,
2        int im_port_target) {
3      im_wspState = Constants.WSPSTATE_SET;
4      String msg_out = null;
5      WSPControlMessage msg = new WSPControlMessage(Constants.WSP_SET,
6          imp_set_channel);
7      msg.setTargetINETA(str_INETA_target);
8      msg.setTargetPort(im_port_target);
9      msg_out = msg.getMessage();
10     if (msg_out != null) {
11       if (bof_WSPDEBUG)
12         Debug.out(this.getClass().toString(),
13             "jbthvoipc1-l00000FI;-T sending CONTROL: SET");
14       wsp_con.send(Constants.WSP_CONTROL_CHANNEL, msg_out.getBytes(),
15           msg_out.length());
16     } else {
17       Debug.err(this.getClass().toString(),
18           "Cannot send CONTROL-Msg:SET channel:" + imp_set_channel);
19     }
20   }
```

Ähnlich wie bei dem Aufruf der Methode `setRTPConnection()`, wird im Gegensatz zum UDP-Modus, bei dem der Aufruf der Methode `destinationSetDone()` lediglich zur Fortsetzung des Ablaufes getätigt wird, eine entsprechende Control-Nachricht generiert und dem SDH versendet. Wird die Antwort '200 OK' von SDH als Bestätigung der Parametrisierung des Channels empfangen, so wird der Ablauf durch den Aufruf der Mehtode `destinationSetDone()` fortgesetzt.

### 3.2.4 Verarbeitung von Sprachdaten (RTP)

Bei der Verarbeitung von Sprachdatenpaketen, soll die akustische Ausgabe empfangener Sprachdatenpakete und die Versendung aufgenommener bzw. digitalisierter Sprachsignale des Benutzers erzielt werden.

Für die Verarbeitung von Sprachdatenpaketen, ist im Bezug auf das SIP mit dem SDP zunächst eine situative Steuerung der entsprechenden Audio-Komponenten erforderlich. In diesem Kapitel wird in diesem Zusammenhang zunächst die erforderliche Steuerung der Audio-Komponenten näher betrachtet und im folgenden die dadurch ermöglichte Verarbeitung der Sprachdatenpakete.

#### 3.2.4.1 Steuerung der Audio-Komponenten

Als Audio-Komponenten, sollen die von einer Soundkarte bereitgestellten Funktionalitäten, allgemein als analoges Signal bereitgestellte Schallschwingungen zu digitalisieren und digitale Sprachdaten akustisch wiederzugeben, entsprechend als Aufnahme- und Abspielgerät gesteuert werden. Die für die Verarbeitung der RTP-Daten erforderliche wesentliche Funktionalität in Zusammenhang mit der Steuerung von Audio-Komponenten wird durch die Klassen `RTPManager.java` und `RTPConnection.java` definiert.

Im weiteren Verlauf werden zur Vereinfachung die Instanzen der Klassen `RTPManager.java` und `RTPConnection.java` als `rtp_mngr` und `rtp_con` benannt, wobei die in Kap. 3.2.1 definierte Architektur des HOBPhone, die Grundlage der Implementierung darstellt. Für die Aufnahme bzw. akustische Wiedergabe von Sprachdaten, werden für die aus RTP-Datenpaketen extrahierten bzw. als RTP-Datenpakete bereitgestellten Audio-Daten, je eine Instanz als Aufnahme-bzw. Abspielgerät einer Klasse benötigt, die den Zugriff auf die entsprechend lokal verfügbaren Ressourcen definiert. Hierzu wird das Package `hob.audio` als Application Programming Interface (kurz API), also als Programmierschnittstelle integriert. Das Package `hob.audio` wurde von HOB implementiert und verwaltet selbst wiederum lokale Sound-APIs insofern, dass lokale Sound-APIs erfasst und eingesetzt werden.

Der Einsatz von Sound-APIs richtet sich hierbei nach möglichst systemnahem und effizientem Einsatz der lokal verfügbaren Ressourcen zur Aufnahme bzw. Wiedergabe von Sprachdaten. Nachstehende Sound-APIs werden im Hinblick auf Verfügbarkeit eines Aufnahme- und Abspielgerätes und Einsatzmöglichkeiten von Kompressionsverfahren für Sprachdatenpakete überprüft:

- Java Sound
- Java Media Framework
- Multimedia System Microsoft

- Sun Audio

Die statische Klasse `AudioManager.java` soll mit entsprechenden Methoden, die durch das Package `hob.audio` verfügbaren Funktionalitäten mittels lokaler Sound-APIs, dem `rtp_mngr` bereitstellen. Hierzu sollen anhand der statischen Funktionen `getRecorder` und `getPlayer` der Klasse `AudioManager.java` die Funktionalität und die Voraussetzungen für die Instanzen die jeweils ein Abspielgerät und ein Aufnahmegerät definieren näher betrachtet werden.

**Listing 3.50:** AudioManager.java -getPlayer

```
public static ca_audioplayer getPlayer(c_waveformatex ds_wfx) {
  try {
    if (audiomanager.is_format_supported_for_output(ds_wfx)) {
      return audiomanager.create_player(ds_wfx, REALTIME);
    }
  } catch (Exception e) {
    Debug.err(AudioManager.class.toString(),
      "Cannot create audioplayer. err:" + e.toString());
    }
  return null;
}
```

Die statische Funktion `getPlayer` mit einer Instanz der Klasse `ca_audioplayer.java` als Rückgabewert, erwartet eine Instanz der Klasse `c_waveformatex` als Übergabeparameter. Zur Vereinfachung werden im Folgenden die Instanzen der Klassen `ca_audioplayer .java` als `audio_player` und `c_waveformatex.java` als `ds_wfx` benannt. Der erwartete Übergabeparameter `ds_wfx` ist hierbei eine allgemeine Definition für die Beschaffenheit eines Sprachdatenstroms (Audio-Stream) als Audio-Format, wobei der Zugriff auf die entsprechenden Funktionen aus dem Package `hob.audio` über die Instanz der Klasse `c_audiomanager.java` erfolgt. Um nun einen `audio_player` zu erhalten, wird über die Funktion `is_format_supported_for_output` des audiomanager überprüft, ob eine Audio-Ausgabe von Daten in Beschaffenheit des übergebenen `ds_wfx` als Input möglich ist. Beinhaltet das übergebene Audio-Format die Beschaffenheit eines Datenkompressionsverfahrens, welches anhand lokal verfügbarer Sound-APIs nicht angewendet kann, so gibt die Funktion `null` als Rückgabewert zurück. Analog dazu, wird bei der Funktion `getRecorder` überprüft, ob die übergebene `ds_wfx` als digitales Ausgabeformat für vom Mikrofon aufgezeichnete Sprachsignale angewendet werden kann. Mit `is_format_ supported_for_input` ist also in diesem Zusammenhang die Möglichkeit der Aufnahme als Input von Sprachsignalen gemeint, bei bestimmter erwarteter Beschaffenheit der digitalisierten Sprachsignale als Ausgabe bzw. Output.

**Listing 3.51:** AudioManager.java -getRecorder

```
public static ca_audiorecorder getRecorder(c_waveformatex ds_wfx) {
  try {
    if (audiomanager.is_format_supported_for_input(ds_wfx))
```

```
4  {
5        return audiomanager.create_recorder(ds_wfx, REALTIME);
6      }
7    } catch (Exception e) {
8      Debug.err(AudioManager.class.toString(),
9        "Cannot create recorder! Err:" + e.getMessage());
10   }
11   return null;
12
13 }
```

Die Erstellung eines `audio_player` in Verbindung mit einem `audio_recorder` erfolgt im Zusammenhang mit der sich aus dem SIP und SDP ergebenden jeweiligen Situation. Ausgehend von dem in Kap. 2.1.2 für das SDP beschriebene Vorgehen, werden bei einem eingehenden oder ausgehenden Anruf, über das SDP die für die bevorstehende Sprachdatenübertragung zu verwendenden Parameter ausgehandelt. Anhand des nachstehenden Ausschnittes eines SDP-Teils einer SIP-Request für einen ausgehenden Anruf, soll der Vorgang zur Steuerung der Audio-Komponenten näher betrachtet werden.

**Listing 3.52:** SDP-Content

```
1 v=0
2 o=HOBPhone_Client 28903 28903 IN IP4 172.22.80.140
3 s=HOBPhone_Session
4 c=IN IP4 172.22.80.140
5 t=0 0
6 m=audio 1697 RTP/AVP 3 0
7 a=rtpmap:3 GSM/8000
8 a=rtpmap:0 PCMU/8000
```

Zur Beschreibung der Beschaffenheit der zu übertragenden Sprachdatenpakete im Bezug zur Sprachdatenkompression, werden im weiteren Verlauf die in 2.1.2 dargestellten Datenkompressionsverfahren als Codec mit der entsprechenden Notation für das SDP zu 'Payload types' herangezogen. Der weiter oben dargestellte SDP-Teil einer ausgehenden SIP-Request 'INVITE' beinhaltet hierbei für dem Parameter mit der Kennzeichnung 'm', der nach SDP allgemein für die Beschreibung der eingesetzten Medien steht, die Werte '3' und '0' als Schlüsselnummer für die Codecs 'GSM610' und 'ULAW'.

Bei einem ausgehenden Anruf, dessen SIP-Request 'INVITE' von einem `sip_client` über einen `sip_dialog` initiiert wird (vgl. Kap. 2.1.1.3 'Verbindungsaufbau mit SIP'), muss somit vor der Generierung der SIP-Request die Information gegeben sein, welche Codecs anhand der lokalen Ressourcen eingesetzt werden können, damit diese bei der SIP-Request mit den entsprechenden Schlüsselnummern als Payload type im SDP-Teil angegeben werden können. Die lokal verfügbaren Codecs werden beim Start des HOBPhone mittels der statischen Funktion `setAudioSupport()` der Klasse `AudioManager.java` ermittelt und bei allen ausgehenden Anrufen in Form eines Arrays für Payload-

Types als default-Wert verwendet. Zusammen mit den Angaben über die lokale IP-Adresse und für die bevorstehende Sprachdatenübertragung korrespondierte lokale Portnummer wird der weiter oben dargestellte SDP-Teil in der Funktion `setSDPContent` des `sip_dialog` generiert. Die Funktion `setSDPContent` erwartet hierbei eine Instanz der Klasse `SDPContent.java` als Übergabeparameter. Wird der Funktion `null` übergeben, so wird ein SDP-Teil mit default-Werten generiert, was bei einem ausgehenden Anruf der Fall wäre. Wird die Funktion mit einem Übergabeparameter als Instanz der Klasse `SDPContent.java` gerufen, so handelt es sich um einen aus einer SIP-Request 'INVITE' oder SIP-Response mit SDP-Teil empfangenen SDP-Teil, der mit den lokalen Bedingungen verglichen wird.

Unabhängig davon, ob es sich um einen ausgehenden oder eingehenden Anruf handelt, ist vor einer Sprachdatenübertragung immer mit einem Empfang von einem SDP-Teil auszugehen, bei einem ausgehenden Anruf in der SIP-Response '200 OK' der Gegenstelle nach Akzeptanz des Anrufes oder bei einem eingehendem Anruf anhand der eingehenden SIP-Request 'INVITE'. Beim Vergleich von lokalen Bedingungen mit einem empfangenen SDP-Teil, wird anhand der Information welcher Codec für die bevorstehende Sprachdatenübertragung verwendet werden soll in der Funktion `setSDPContent` einer korrespondierten Instanz der Klasse `SIPDialog.java` überprüft, ob dieser entsprechend den lokalen Bedingungen auch verfügbar ist. Kann der vorgeschlagene Codec verwendet werden, wird in der Funktion `setSession` je ein entsprechender `audio_player` und `audio_recorder` erzeugt. Während die Überprüfung der Verfügbarkeit in der Funktion `setSDPContent` der Klasse `SIPDialog.java` erfolgt, werden je ein entsprechender `audio_player` und `audio_recorder` bei gegebenem Audio-Support innerhalb der Funktion `setSession` der Klasse `RTPManager.java` erstellt, die auch im weiteren Verlauf die Verarbeitung der Sprachdatenpakete übernimmt. Im Aktivitätsdiagramm aus Abb. 3.12, wird der Vorgang zur Steuerung der Audio-Komponenten bei ausgehenden oder eingehenden Anrufen, der mittels eines neuen `sip_dialog` in Gang gesetzt wird, im einzelnen dargestellt.

Unabhängig von dem für die bevorstehende Sprachdatenübertragung zu verwendenden Codec, ist für jede Sprachdatenübertragung eine lokale Portnummer erforderlich, über welche die Übertragung erfolgen soll und entsprechend im SDP eingetragen bzw. aufgeführt werden muss. Wie aus dem in Abb. 3.12 dargestelltem Aktivitätsdiagramm ersichtlich, wird die lokal zu verwendende Portnummer über einen Dialog in Form von Port-Anfrage und Erhalt einer Portnummer mittels der `rtp_con` ermittelt. Die Ermittlung der zu verwendenden Portnummer mittels der Klasse `RTPConnection.java` ist abhängig von der Art der Datenübertragung bzw. dem Modus des HOBPhone, ob es im Betrieb für Datenübertragung mittels UDP oder über eine SSL-Verbindung eingesetzt wird.

Hierbei ist zu beachten, dass es sich beim Aufruf der Funktion `connectionReceived` des `sip_dialog` der eine Portnummer und IP-Adresse übergeben wird, lediglich um die Infor-

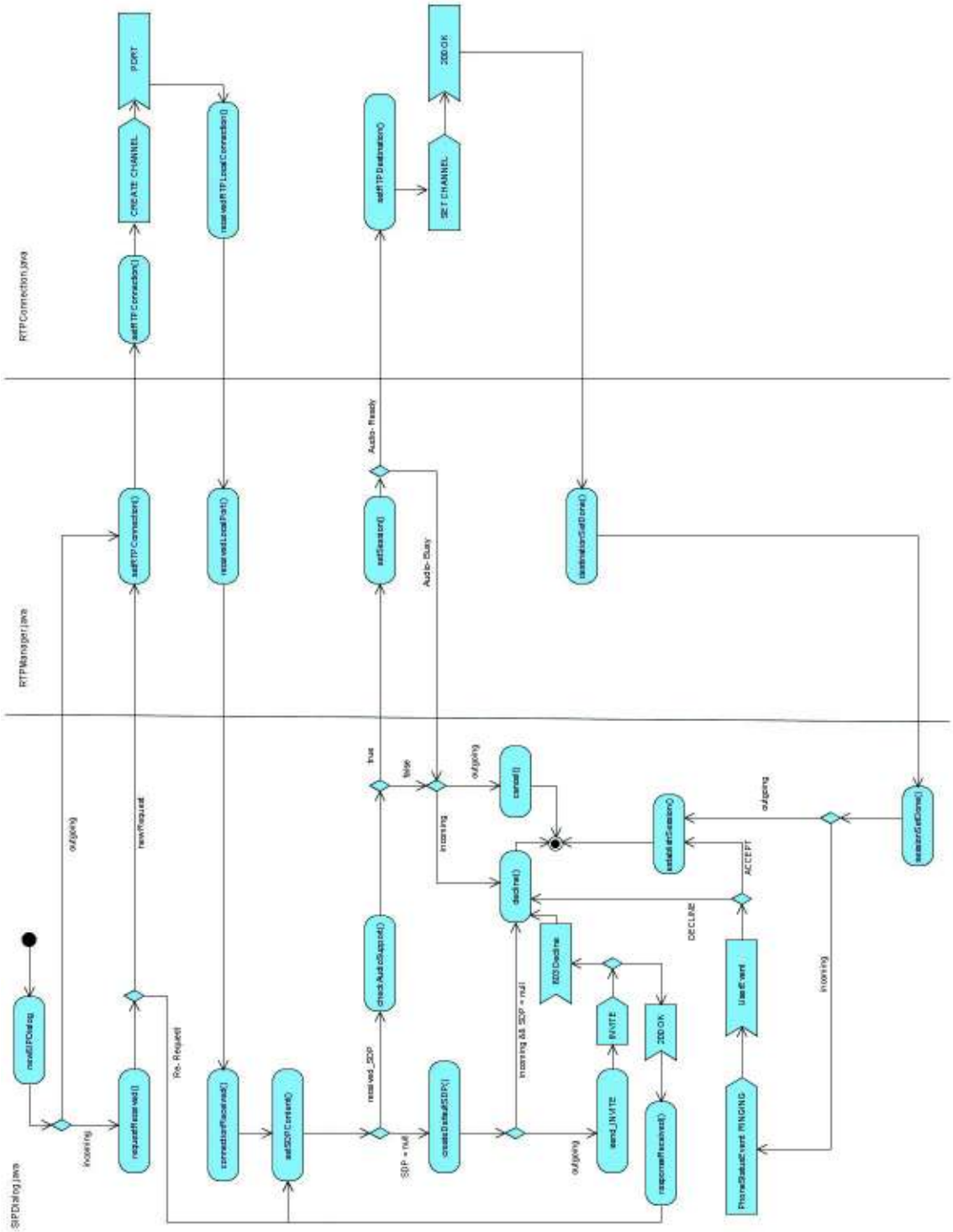


**Abbildung 3.12:** Aktivitätsdiagramm 'newSIPDialog()'

mation handelt, welche Portnummer und IP-Adresse im SDP-Teil als lokale Portnummer und IP-Adresse angegeben wird bzw. werden soll und nicht um die tatsächliche Portnummer und IP-Adresse des HOBPhone, über die lokal Daten erwartet bzw. Daten versendet werden würden. Durch die abstrakte Definition der Klasse `RTPConnection.java` spielt dies an dieser Stelle keine Rolle, da die tatsächliche Datenübertragung unabhängig erfolgt, wobei der Dialog der zur Anforderung einer Portnummer der abstrakten Klasse `RTPConnection.java` in seinen Einzelheiten in Kap. 3.1.3 'WSP-Protokoll' dargestellt wird.

#### 3.2.4.2 Verarbeitung der Sprachdatenpakete

Sind die Parameter für eine Sprachdatenübertragung gesetzt, innerhalb der Funktion `setSession` erzeugte Audio-Komponenten als `audio_recorder` und `audio_player` verfügbar und die Funktion `establishSession` des jeweiligen `sip_dialog` gerufen, so kann die Übertragung der Sprachdatenpakete beginnen. Bei der Übertragung der Sprachdatenpakete, sind vom `audio_recorder` generierte Sprachdatenpakete mit einem RTP-Header zu versehen und zu versenden und empfangene RTP-Datenpakete als Sprachdatenpakete dem `audio_player` zur Wiedergabe bereitzustellen. Der Versand vom `audio_recorder` aufgenommener Sprachdaten erfolgt durch einen Thread als Instanz der Klasse `RTPSender.java`. Entsprechend den Ausführungen für RTP in Kap. 2.1.3 'Real-time Transport Protocol (RTP)', werden den durch den `audio_recorder` bereitgestellten Sprachdatensequenzen entsprechende RTP-Header hinzugefügt, was im nachstehenden Ausschnitt aus dem Quellcode des `RTPSender.java` dargestellt wird.

**Listing 3.53:** RTPSender.java

```
do {
  im_nBytesRead = ca_recorder.read(byr_recBuff, HEADER_SIZE,
      im_buffLen_recorder, im_lenFromInput);

  if (im_nBytesRead > 0) {
    byr_recBuff[0] = (byte) ((0x80 >>> 0) & 0xFF);
    byr_recBuff[1] = (byte) ((im_pt_recorder >>> 0) & 0xFF);
    im_mySeqNo++;
    /* seqNo 2-3 */
    byr_recBuff[2] = (byte) ((im_mySeqNo >>> 8) & 0xFF);
    byr_recBuff[3] = (byte) ((im_mySeqNo >>> 0) & 0xFF);
    /* Timestamp 4-7 */
    byr_recBuff[4] = (byte) ((im_timestamp >>> 24) & 0xFF);
    byr_recBuff[5] = (byte) ((im_timestamp >>> 16) & 0xFF);
    byr_recBuff[6] = (byte) ((im_timestamp >>> 8) & 0xFF);
    byr_recBuff[7] = (byte) ((im_timestamp >>> 0) & 0xFF);
    /* ssrc 8-11 */
    byr_recBuff[8] = (byte) ((im_mySSRC >>> 24) & 0xFF);
    byr_recBuff[9] = (byte) ((im_mySSRC >>> 16) & 0xFF);
```

```
20     byr_recBuff[10] = (byte) ((im_mySSRC >>> 8) & 0xFF);
21     byr_recBuff[11] = (byte) ((im_mySSRC >>> 0) & 0xFF);
22     im_timestamp += im_nBytesRead;
23     rtp_con.sendRTP(byr_recBuff, 0, im_nBytesRead + HEADER_SIZE);
24   }
25 } while (bo_session);
```

Während der Versand über die Funktion `sendRTP` der `rtp_con` erfolgt, ruft wiederum die `rtp_con` bei empfangenen RTP-Datenpaketen die Funktion `playPacket` des `rtp_mngr`, dessen Quellcode nachstehend dargestellt wird.

**Listing 3.54:** RTPManager.java -playPacket

```
1 public synchronized boolean playPacket(byte[] byr_receiveBuff, int offset,
2       int imp_len) {
3   if (ca_player != null \&\& ca_player.isStarted()) {
4     // TODO: interpret the changing fields of the header:
5
6     int im_newSeqNr = 0;
7 if ((((short) (unsigned_int(byr_receiveBuff[2 + offset]) + 256 * unsigned_
      int(byr_receiveBuff[0 + offset])) >>> 14) \& 3) != im_rtp_version) {
8     Debug.err(this.getClass().toString(),
9         "!! Received a non RTP packet !!");
10 }
11
12   im_pt_in = byr_receiveBuff[1 + offset] \& 0x7F;
13
14   // Marker = (s \& RTP_MARKER_BIT) != 0;
15   im_newSeqNr = unsigned_int(byr_receiveBuff[3 + offset]) + 256
16       * unsigned_int(byr_receiveBuff[2 + offset]);
17
18   im_timeStamp_in = unsigned_int(byr_receiveBuff[7 + offset]) + 256
19       * unsigned_int(byr_receiveBuff[6 + offset]) + 65536
20       * unsigned_int(byr_receiveBuff[5 + offset]) + 16777216
21       * unsigned_int(byr_receiveBuff[4 + offset]);
22
23   im_ssrc_in = unsigned_int(byr_receiveBuff[11 + offset]) + 256
24       * unsigned_int(byr_receiveBuff[10 + offset]) + 65536
25       * unsigned_int(byr_receiveBuff[9 + offset]) + 16777216
26       * unsigned_int(byr_receiveBuff[8 + offset]);
27     if (im_newSeqNr < im_lastSeqNr) {
28     Debug.err(getClass().toString(), "Received too late: im_lastSeqNr: "
29       + im_lastSeqNr + " received:" + im_newSeqNr);
30     }
31     im_lastSeqNr = im_newSeqNr;
32
33   if (ca_player.write(byr_receiveBuff, HEADER_SIZE + offset, imp_len
34       -HEADER_SIZE) > 0) {
35       return true;
36   } else {
```

```
37      Debug.err(this.getClass().toString(),
38          "ca_player.write FAILED: im_readlen=0");
39    }
40    }
41    return false;
42  }
```

### 3.2.5 Grafische Benutzeroberfläche

Für die Implementierung einer grafischen Benutzeroberfläche wird die abstrakte Klasse `PhoneUserInterface.java` als Basis definiert. Hierbei werden die wesentlichen Eigenschaften einer Benutzerschnittstelle, welche der in Kap. 3.1 definierten Architektur des HOBPhone im Bezug auf die VoIP-Konzeption für SSL-geschützte Internettelefonie entspricht, definiert. Durch die abstrakte Definition der Eigenschaften einer Benutzerschnittstelle wird zum einen die Implementierung von Benutzeroberflächen allgemein ermöglicht, wodurch z.B. durch eine entsprechende Benutzerschnittstelle das HOBPhone als Konsolenanwendung verwendet werden könnte. Des weiteren werden für die Implementierung einer grafischen Benutzeroberfläche, grafische Objekte für die wesentliche Funktionalität abstrakt bereitgestellt. Diese können bei einer Implementierung einer grafischen Benutzeroberfläche in Form und Farbe definiert bzw. geändert werden, wodurch dem Benutzer z.B. Farbschemen oder Themenbereiche im Bezug zum Erscheinungsbild zur Verfügung gestellt werden könnten.

Einstiegspunkt der Anwendung als Java-Applet ist die Klasse MainApplet.java. Entsprechend der in Kap. 3.2.1 definierten Architektur des HOBPhone, soll das HOBPhone im Bezug auf die grundsätzliche Funktionsweise bei der Datenübertragungsart im UDP- oder SSL-Modus gestartet werden können. Die sich aus dem Modus ergebende differenzierte Funktionsweise im Bezug auf die Datenübertragung wurde in Kap. 3.2.2 'Übertragung von Signalisierungs- und Sprachdaten' im Einzelnen erläutert. Entscheidend ist, dass entsprechend der Definition der Architektur des HOBPhone in Kap. 3.2.1, die Komponenten Bereitstellung und Verarbeitung der Signalisierungs- und Sprachdaten sowie die Benutzerschnittstelle unabhängig voneinander konzipiert werden können. Diesbezüglich soll im weiteren Verlauf vom Einstiegspunkt in der Klasse `MainApplet.java` ausgehend, die erforderliche Interaktion zwischen den Komponenten zur Verarbeitung und Bereitstellung der Signalisierungs- und Sprachdaten zunächst anhand dem nachstehende Ausschnitt aus dem Quellcode der Klasse MainApplet.java näher betrachtet werden.

**Listing 3.55:** MainApplet.java

```
1 str_boSSL = this.getParameter("mode");
2 if (str_boSSL != null \&\& str_boSSL.length() > 0) {
3   if (str_boSSL.trim().equalsIgnoreCase("WSP")) {
```

```
    im_mode = this.mode_WSP;
  } else {
    if (str_boSSL.trim().equalsIgnoreCase("UDP")) {
      im_mode = this.mode_UDP;
    } else {
      im_mode = this.mode_INVALID;
    }
  }
} else {
  im_mode = this.mode_INVALID;
}
switch (im_mode) {
  case mode_WSP:
    userData = new String[3];
    userData[0] = this.getParameter(HLS.PARAM_USERNAME);
    userData[1] = this.getParameter(HLS.PARAM_PASSWORD);
    if (userData[0] != null \&\& userData[1] != null) {
      connectWSP();
    } else {
      setLoginViewSSL(true);
    }
    break;
  case mode_UDP:
    PhoneManager.getPhoneManager().createPhoneUDP();
    setLoginViewSSL(false);
    break;
  default:
    Debug.err(this.getClass().toString(), "INVALID MODE.");
    System.exit(0);
  break;
}
```

Wie aus dem oben aufgeführten Ausschnitt des Quellcodes der Methode `init()` ersichtlich, wird über die HTML-Seite über die das HOBPhone geladen wird zunächst ein Parameter `mode` erwartet. Soll das Applet im SSL-Modus gestartet werden, so wird das HOBPhone über den in Kap. 3.1 Abb. 3.2 dargestellten Anmelde-Dialog des WSP mit unter anderem dem Parameter `mode=WSP` geladen, wobei das HOBPhone im UDP-Modus mit dem übergebenen Parameter `mode=UDP` geladen werden kann. Entsprechend dem Modus wird über die Funktion `createPhoneUDP` bzw. `createPhoneWSP` mittels der internen Methode `initGUI` der Klasse `PhoneManager.java` eine vordefinierte Benutzerschnittstelle erzeugt, was durch den nachstehenden Quellcode verdeutlicht werden soll:

**Listing 3.56:** PhoneManager.java

```
private void initGUI(boolean bo_SSL) {
  PhoneUserInterface ui = null;
  switch (im_ui_mode) {
  case Constants.byf_MAIN_GUI:
    ui = new MainGUI(this, bo_SSL);
```

```
6       break;
7     case Constants.byf_BASIC_GUI:
8       ui = new BasicGUI(this, bo_SSL);
9       break;
10    case Constants.byf_CONSOLE_UI:
11      ui = null;
12      break;
13    case Constants.byf_TEST_UI:
14      ui = null;
15      break;
16    }
17    if (ui != null) {
18      this.mf = ui;
19      phone_mngr.addPhoneStatusListener(ui);
20    } else {
21      Debug.err(getClass().toString(), "CANNOT START UI: invalid mode:"
22          + im_ui_mode);
23    }
24
25  }
```

Im Aktivitätsdiagramm aus Abb. 3.13 soll der oben beschriebene Ablauf veranschaulicht werden.

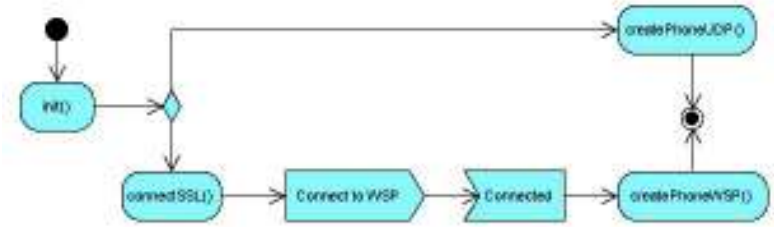


**Abbildung 3.13:** Einstiegspunkt in MainApplet.java im Aktivitätsdiagramm

Der Konstruktor für die Klassen `MainGUI.java` und `BasicGUI.java` ergibt sich aus dem Konstruktor der gemeinsamen abstrakten Oberklasse `PhoneUserInterface.java`:

**Listing 3.57:** PhoneUserInterface.java

```
1   public PhoneUserInterface(PhoneManager phone_mngrp, boolean bop_SSL) {
2     bo_SSL = bop_SSL;
3     phone_mngr = phone_mngrp;
4   }
```

Anhand des Übergabeparameters `bop_SSL` ist ersichtlich, dass ist für eine Instanz einer Benutzerschnittstelle die Unterscheidung des Modus in SSL oder UDP erforderlich ist.

Wird die Anwendung im SSL-Modus gestartet, so erhält das Applet nach Aufbau der SSL-Verbindung alle Informationen über Asterisk-Accounts die am WSP konfiguriert sind über einen Dialog zwischen dem WSP bzw. dem entsprechenden SDH und dem HOBPhone. Im UDP-Modus hingegen, erfolgt die Datenübertragung unmittelbar mit einer Komponente in der VoiP-Architektur, bei der das HOBPhone lokal eingesetzt wird und Asterisk-Accounts vom Benutzer manuell definiert werden.

Durch den übergebenen Parameter `bop_SSL` werden somit lediglich die Funktionalitäten für eine manuelle Eingabe von Account-Daten ausgeblendet, wobei verbleibende Funktionalitäten anhand der unabhängigen Konzeptionierung der Komponenten nicht betroffen sind, d.h. bei Erhalt von Account-Daten im SSL-Modus vom WSP ,wird intern eine manuelle Registrierung sozusagen nachgestellt, indem die entsprechende Methode für die SIP-Registrierung einer Benutzerschnittstelle von der Klasse `WSPManager.java` anstatt einer Handlung des Benutzers gerufen wird (vgl. Listing 3.2.3.2, Listing 3.2.3.2). Die Methode für eine SIP-Registrierung über die Benutzerschnittstelle ist in der abstrakten Klasse `PhoneUserInterface.java` definiert, welche im nachstehenden Quellcode dargestellt wird.

**Listing 3.58:** PhoneUserInterface.java -ua_register

```
1 public boolean ua_register(String str_settingName) {
2     return phone_mngr.ua_register(str_settingName);
3 }
```

Einer Instanz der abstrakten Definition einer Benutzerschnittstelle `PhoneUserInterface .java`, wird über den im weiter oben dargestellten Konstruktor der Benutzerschnittstelle eine Instanz der Klasse `PhoneManager.java` übergeben, die im weiteren Verlauf zur Vereinfachung als `phone_mngr` benannt wird. Anhand des `phone_mngr`, ist die Benutzerschnittstelle in der Lage, gewünschte Funktionen über Handlungen des Benutzers auszulösen, wodurch die Informationsrichtung vom Benutzer zur Applikation implementiert wird. Hierzu sind die Möglichen Handlungen die eine Interaktion der Anwendung erfordern abstrakt in der Benutzerschnittstelle definiert und werden über den `phone_ mngr` angesteuert, was z.B. auch aus der oben dargestellten Funktion `ua_register` der abstrakten Klasse `PhoneUserInterface.java` ersichtlich ist.

Die Informationsrichtung von der Applikation zur Benutzerschnittstelle wird anhand intern ausgelöster Ereignisse implementiert, die durch die Klasse `PhoneStatusEvent. java` definiert werden. Ereignisse, die von der Anwendung ausgelöst werden können, sind Anwendungsstatus-Meldungen wie z.B. 'Eingehender Anruf' in Verbindung mit SIP-Status-Meldungen die den Anwendungsstatus betreffen, wie z.B. die Beendigung eines gerade aktiven eingehenden Anrufes. Die Zusammensetzung der Definition eines Phonestatus-Event soll anhand des nachstehenden Quellcodes veranschaulicht werden.

**Listing 3.59:** PhoneStatusEvent.java

```
public PhoneStatusEvent(int imp_newSipKey, int imp_newPhoneKey, String
    strp_callID,String str_callInfop) {
        this.im_sipKey = imp_newSipKey;
        this.im_phoneKey = imp_newPhoneKey;
        this.str_callInfo = str_callInfop;
        this.str_callID = strp_callID;
}
```

Für eine Benachrichtigung der Benutzerschnittstelle über einen Phonestatus-Event, wird eine entsprechende Instanz der Klasse `PhoneStatusEvent` erstellt,

**Listing 3.60:** PhoneUserInterface.java -setPhoneStatus

```
setPhoneStatus(new PhoneStatusEvent(Constants.imf_TRYING,
Constants.byf_INCOMING_CALL, str_callID, msg_in.header.str_fromFullName));
```

die dem `phone_mngr` über die Methode `notifyPhoneStatus` übergeben wird.

**Listing 3.61:** PhoneUserInterface.java -isActiveClient

```
if (phone_mngr.isActiveClient(this)) {
    phone_mngr.notifyPhoneStatus(pse);
}
```

Der `phone_mngr` ruft dann die Methode `newPhoneStatusEvent` für alle Instanzen die als PhoneStatusListener einem entsprechenden Container hinzugefügt worden sind, wobei von einer Benutzerschnittstelle als `PhoneStatusListener` ausgegangen wird. Durch nachstehenden Quellcode der Methode `newPhoneStatusEvent`, welche als abstrakte Methode in der Klasse `PhoneUserInterface.java` definiert ist, soll die Verarbeitung einer Instanz der Klasse `PhoneStatusEvent` veranschaulicht werden.

**Listing 3.62:** PhoneUserInterface.java -newPhoneStatusEvent

```
public void newPhoneStatusEvent(PhoneStatusEvent e) {

        num.setText(Constants.getPhoneStatusName(e.im_phoneKey));
        num.setText(Constants.getKeyName(e.im_sipKey));

        switch (e.im_phoneKey) {
        case 11:
            //SESSION
            thr_beeper.bo_ringing = false;
            mi_Reg.setEnabled(false);
            mi_unReg.setEnabled(true);
            cut.setText("HANGUP");
            cut.setEnabled(true);
            call.setEnabled(false);
          str_currentCallID = e.str_callID;
            break;
```

```
17        }
18 }
```

In Abb. 3.14 wird eine Benutzeroberfläche dargestellt, die hauptsächlich für die vereinfachte Bedienung der wesentlichen Funktionalitäten während der Implementierungsphase erstellt ist, jedoch auch als kompakte Variante nutzbar wäre.

**Abbildung 3.14:** Grafische Benutzeroberfläche 'Standard'

In Abb. 3.15 ist die Implementierung der grafischen Oberfläche abgebildet, wie sie in dieser Version des HOBPhone eingesetzt wird, wobei ein Entwurf für spätere Versionen in Kap. 4 'Fazit und Ausblick' dargestellt wird.

**Abbildung 3.15:** Grafische Benutzeroberfläche 'Basic'

## 3.3 Implementierung eines SDH

Die Implementierung des Server Data Hook zur Verarbeitung von Datenpaketen basiert auf der in Kap. 2.3 'HOB WebSecureProxy (WSP)' beschriebenen Funktionalität und Architektur des WSP und wird in den folgenden Kapiteln dargestellt.

### 3.3.1 Architektur des SDH

Für die Implementierung eines SDH für das VoIP-Konzept für SSL-geschützte Internettelefonie, wird die Architektur des zu implementierenden SDH, aufbauend auf der in Kap. 3.1 dargestellten Architektur des Konzeptes, weiter definiert.

#### 3.3.1.1 Aufbau des SDH

Die Implementierung des SDH erfolgt in der Programmiersprache C++ in der Datei `xltclib9.cpp` und soll durch den WSP als DLL geladen werden bzw. dem WSP als `xltclib9.dll` zur Verfügung stehen. Hierzu wird die für den Zugriff auf den SDH als DLL die Funktion `m_hlclib01` definiert und mittels `extern "C"` exportiert:

**Listing 3.63:** xltclib9.cpp

```
extern "C" __declspec( dllexport ) void m_hlclib01( struct dsd_hl_clib_1 *
    adsp_hl_clib_1 ) {
.
.
.
}
```

Die erforderliche Konfiguration des WSP zum Laden der DLL wird in der Konfigurationsdatei des WSP anhand nachstehendem Eintrag vorgenommen:

**Listing 3.64:** WSPConf.xml -library file name

```
<library-file-name>F:\Alan\Project\WSP\AKBI61\HLSEC\xltclib9-d\xltclib9.
    dll</library-file-name>
```

Wie in Kap. 2.3 dargestellt, wird für jede SSL-Verbindung eine Instanz des SDH erzeugt, die beim Empfang von mit dem HOBPhone korrespondierten Daten über die oben dargestellte Funktion gerufen wird. Hierbei wird dem SDH bei jedem Aufruf eine Struktur seitens des WSP übergeben, die unter anderem zur Kommunikation zwischen dem SDH und dem WSP dient und die für den Aufruf maßgeblichen Daten bereitstellt. Die bereitgestellten Daten empfangener Datenpakete, werden anhand der für die Kommunikation des SDH mit dem WSP bestimmten Parametern der übergebenen Struktur interpretiert. Nachstehender Quellcode stellt die Struktur `dsd_hl_clib_1` dar, welche in der Header Datei `hob-xsclib01.h` definiert ist:

**Listing 3.65:** hob-xsclib01.h

```
struct dsd_hl_clib_1 {    /* HOBLink Copy Library 1 */
   int              inc_func;   /* called function    */
   int              inc_return;   /* return code    */
   char *        achc_work_area;  /* addr work-area   */
   int              inc_len_work_area;  /* length work-area */

   struct         dsd_gather_i_1 *adsc_gather_i_1_in; /* input data */
   struct         dsd_gather_i_1 *adsc_gather_i_1_out;  /* output data  */

   BOOL       (* amc_aux) ( void *, int, void *, int );  // Helper routine
        pointer
   void *        ac_ext;    /* attached buffer pointer  */
   void *        ac_conf;   /* data from configuration  */
   int              imc_flags_1;    /* flags of configuration */
   int              imc_signal;   /* signals occured  */
   void *       vpc_userfld;    /* User Field Subroutine  */
   BOOL       boc_callagain;  /* call again this direct.  */
   BOOL       boc_callrevdir; /* call on reverse direct.  */
   BOOL       boc_no_conn_s;  /* do not conn. to server   */
   BOOL       boc_eof_client; /* End-of-File Client */
   BOOL       boc_eof_server; /* End-of-File Server */
```

```
};
```

Erfolgt der Zugriff auf den SDH über die weiter oben dargestellte Funktion `m_hlclib01` mit Übergabe der Struktur `dsd_hl_clib_1`, so wird innerhalb der Funktion `m_hlclib01` zunächst über den Aufruf `dsd_hl_clib_1->inc_func` die Schlüsselnummer unterschieden, welche die für diesen Aufruf gewünschte Funktion darstellt:

**Listing 3.66:** xltclib9.cpp -switch (adsp_hl_clib_1->inc_func)

```
   switch (adsp_hl_clib_1->inc_func) {
       case DEF_IFUNC_START:

          bol1 = adsp_hl_clib_1->amc_aux( adsp_hl_clib_1->vpc_userfld,
                                          DEF_AUX_MEMGET,
                                          &adsp_hl_clib_1->ac_ext,
                                          sizeof(struct dsd_clib1_session
                                              _9_1) );
       if (bol1 == FALSE) {

         adsp_hl_clib_1->inc_return = DEF_IRET_ERRAU;
         return;
       }


       memset( adsp_hl_clib_1->ac_ext, 0, sizeof(struct dsd_clib1_session
           _9_1) );
       return;

       case DEF_IFUNC_CLOSE:
           bol1 = adsp_hl_clib_1->amc_aux( adsp_hl_clib_1->vpc_userfld,
                                          DEF_AUX_MEMFREE,
                                          &adsp_hl_clib_1->ac_ext,
                                          sizeof(struct dsd_clib1_session
                                              _9_1) );
       if (bol1 == FALSE) {
         adsp_hl_clib_1->inc_return = DEF_IRET_ERRAU;
       }
       return;

      case DEF_IFUNC_REFLECT:
.
.
.
}
```

Wie aus dem dargestellten Ausschnitt des Quellcodes der Funktion `m_hlclib01` ersichtlich, kann der SDH über die Funktionen START, CLOSE und REFLECT aufgerufen werden. Während die Funktion REFLECT für die Verarbeitung von Datenpaketen maßgeblich ist und in Kap. 3.3.2 'Verarbeitung von Datenpaketen im SDH' behandelt

wird, soll im Bezug auf die Architektur und den Aufbau des SDH die Funktion START näher betrachtet werden, die als erstes vom WSP nach dem Aufbau der SSL-Verbindung am SDH aufgerufen wird.

Bei der in der Header-Datei definierten Struktur handelt es sich um eine allgemeine Definition von Informationen, die dem SDH vom WSP bei jedem Aufruf übergeben wird, wonach die Struktur Basis für alle implementierten SDH ist. Während Informationen vom WSP an den SDH über die erwähnte Struktur übergeben werden, erfolgt die Kommunikation des SDH mit dem WSP vom SDH ausgehend, über die Funktion `amc_aux`, der ein Wert als Kommando an den WSP übergeben werden kann. Abgesehen von dieser allgemeinen Definition in der Header-Datei, wird im SDH selbst eine Struktur definiert, die einen Zugriff auf Informationen über alle Aufrufe des SDH seitens des WSP ermöglichen soll. Nachstehender Quellcode stellt die Struktur `dsd_clib1_session_9_1`, welche im SDH selbst definiert wird dar.

**Listing 3.67:** xltclib9.cpp -struct dsd_clib1_session_9_1

```
1  struct dsd_clib1_session_9_1 {    /* structure session    */
2     int        imc_count_call;     /* count all calls    */
3     int        imc_count_inv_call;   /* count inv. server calls  */
4     int        inc_pos_to_server;    /* position in datastream   */
5     BOOL       boc_first_packet;   /* first packet send  */
6     BOOL       boc_end_connection;      /* end connection now  */
7     BOOL       boc_sipr1_1_active;     /* SIP request 1 active   */
8     BOOL       boc_udpr1_1_active;   /* UDP request 1 active   */
9     struct dsd_hl_aux_ch_ident dsc_aux_ch_ident;
10    struct dsd_sdh_ident_set_1 dsc_g_idset1;
11    struct dsd_sdh_sip_requ_1 dsc_sipr1_1;  /* SIP request     */
12    struct dsd_sdh_udp_requ_1 dsc_udpr1_1;  /* UDP request     */
13    struct sockaddr_in dsc_soai_udpr1_bind;  /* UDP address info */
14    struct sockaddr_in dsc_soai_udpr1_send;  /* UDP address info */
15    int        iml_state_2;      /* part two of state  */
16    int        iml_sp;       /* send pointer     */
17    int        iml_rc;       /* return code    */
18    char       chl_channel;      /* channel received   */
19    ied_client_input_state_def iel_cis_def;  /* state of client input  */
20
21 };
```

Hierzu wird der im SDH selbst definierten Struktur aus dem durch den WSP bei jedem Aufruf neu bereitgestelltem Speicherplatz ein Bereich reserviert. Dies erfolgt bei Aufruf der Funktion START des SDH. Nachstehender Ausschnitt aus dem weiter oben dargestelltem Quellcode der Funktion `m_hlclib01`, veranschaulicht wie mittels `memset` dem Zeiger `ac_ext` die Struktur `dsd_clib1_seesion_9_1` zugeordnet wird, wobei dem SDH über die Struktur `dsd_hl_clib_1` mittels `ac_ext` ein Speicherbereich zur Verfügung gestellt wird (vgl. Listing 3.3.1.1 `case DEF_IFUNC_START:`). Die Struktur `dsd_hl_clib_1`,

über die der Zugriff auf `ac_ext` erfolg, wird bei Ausführung der Funktion `m_hlclib01` übergeben.

Bevor der Zeiger `ac_ext` auf einen Speicherbereich verweisen kann, wird hierbei dem WSP mittels der Schlüsselnummer `DEF_AUX_MEMGET` ein Signal über den Aufruf der Funktion `amc_aux` übergeben, wodurch ein entsprechender Speicherplatz freigegeben wird, bzw. der Zeiger `ac_ext` auf eine gültige Speicheradresse zeigt.

Durch die Reservierung eines Speicherbereiches für den SDH am WSP kann eine sog. 'State-Machine' realisiert werden. Anhand Informationen im reservierten Speicherbereich, auf die bei jedem Aufruf zugegriffen werden kann, wird zum Beispiel eine Speicherung eines Schlüsselwertes ermöglicht, der einen Status darstellt und bei einem nächsten Aufruf für die Interpretation der übergebenen Daten maßgeblich sein kann.

#### 3.3.1.2 Gather-Strukturen

Ausgehend von der Zielsetzung in Kap. 3.1 die Verarbeitung von Daten möglichst effizient zu gewährleisten, ist eine nähere Betrachtung der Vorgehensweise bei der Speicherverwaltung erforderlich. Hierzu soll durch sog. 'Gather-Strukturen' die Belegung von Speicherbereichen bzw. Speicheradressen optimiert werden. Eine 'Gather-Struktur' besteht hierbei aus drei Komponenten, welche als 'Zeiger' fungieren: CURRENT, NEXT und END. Anhand dieser drei Zeiger, können Sequenzen von Informationen korrespondiert bzw. verwendet werden, ohne dass die Informationen durch neue Belegung von einem Speicherplatz kopiert werden müssen. Der nachstehende Quellcode stellt die Definition von Gather-Strukturen in der Header-Datei dar.

**Listing 3.68:** hob-xsclib01.h -struct dsd_gather_i_1

```
1 struct dsd_gather_i_1 {                        /* gather input data       */
2    struct dsd_gather_i_1 *adsc_next;  /* next in chain
3     */
4    char *     achc_ginp_cur;                   /* current position        */
5    char *     achc_ginp_end;                  /* end of input data       */
6 };
```

Während die Zeiger CURRENT und END auf zwei Speicheradressen zeigen woraus sich eine Sequenz von Informationen interpretieren lässt, kann durch den Zeiger NEXT auf eine Speicheradresse verwiesen werden, die wiederum eine weitere Gather-Struktur repräsentiert. Somit können 'Ketten' von Gather-Strukturen erstellt werden, wodurch insbesondere fragmentierte Daten bzw. Informationen effizient durch 'Gather-Ketten' wieder als eine Informationseinheit dargestellt werden können. Nachstehende Abbildung soll die Funktionsweise von Gather-Strukturen verdeutlichen, welche für die Ausführungen der Verarbeitung von Datenpaketen im folgenden Kap. 3.3.2 in Ihrer Beschaffenheit maßgeblich sein werden.

### 3.3.2 Verarbeitung von Datenpaketen im SDH

Bei der Verarbeitung von Datenpaketen im SDH ausgehend von der Funktionsweise des WSP ist zunächst grundsätzlich die Informationsrichtung der zu erarbeitenden Datenpakete zu unterscheiden, wobei es sich unabhängig davon um Control-, gemäß dem WSP-Protokoll aus Kap. 3.1.3 und SIP- oder RTP-Datenpakete handeln kann. Hierbei kommen die Informationsrichtungen der Datenpakete vom HOBPhone ausgehend über den SDH in die VoIP-Architektur und umgekehrt in Frage. Wie aus Abb. 3.16 ersichtlich, erfolgt bei beiden Informationsrichtungen eine Datenübertragung über eine SSL-Verbindung und mittels UDP.

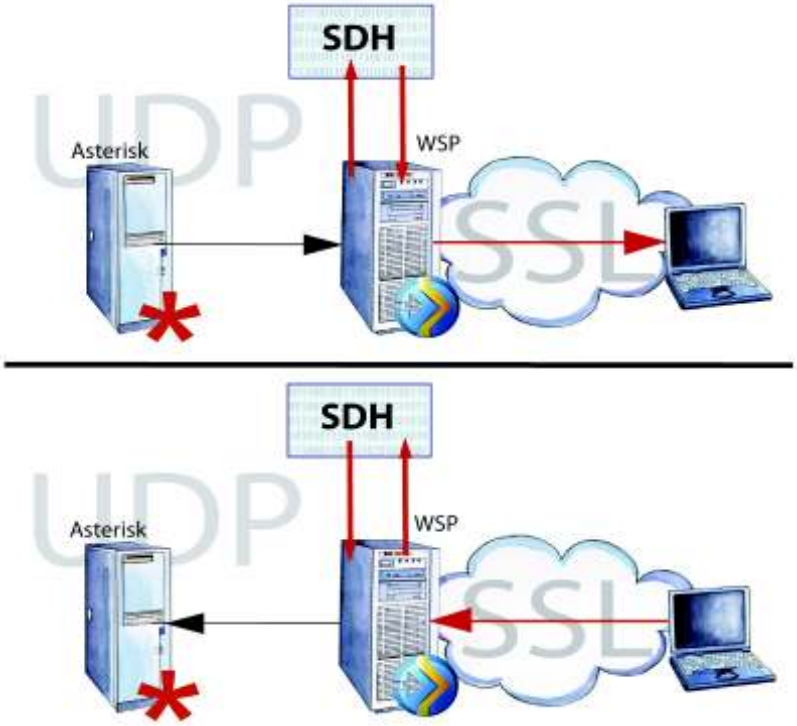


**Abbildung 3.16:** Informationsrichtung von Datenpaketen

Die jeweilige Verarbeitung erfolgt in der Methode `m_reflect`, welche im weiteren Verlauf im Einzelnen dargestellt wird und den Kern der Implementierung des SDH darstellt. Wie in Kap. 3.1.3 dargestellt, versendet der SDH nach Aufbau der SSL-Verbindung und erfolgter Registrierung am WSP, die am WSP administrierten Informationen für das HOBPhone für Asterisk-Accounts. In diesem Zusammenhang soll zunächst näher auf die Informationsrichtung von der VoIP-Architektur zum HOBPhone eingegangen werden.Handelt es sich um Daten, die von der VoIP-Architektur mittels UDP empfangen wurden, so ist seitens des WSP bei Übergabe der Empfangenen Daten der Wert für die

Variable `imc_signal` aus der allgemein definierten Struktur mit einer für entsprechend der empfangenen Daten definierten Schlüsselnummer für SIP oder RTP belegt. Dabei kann es auch vorkommen, dass bei einem Aufruf des SDH, sowohl Daten die vom HOBPhone als auch Daten die mittels UDP von der VoIP-Architektur empfangen wurden bereitgestellt werden.

Bevor jedoch Daten aus der VoIP-Architektur empfangen werden können, findet der erwähnte Dialog zwischen dem HOBPhone und dem SDH nach Aufbau der SSL-Verbindung und Anmeldung am WSP statt. Damit die am WSP administrierten Information für das HOBPhone für Asterisk-Accounts vom SDH dem HOBPhone versendet werden können, werden diese vorerst vom WSP durch die mit

`DEF_AUX_GET_IDENT_SETTINGS`

korresponiderte Schlüsselnummer als Kommando über die Funktion `amc_aux` angefordert und in eine Struktur `dsc_g_idset1` zwischengespeichert:

**Listing 3.69:** xltclib9.cpp -DEF_AUX_GET_IDENT_SETTINGS

```
1
2 memset( &adsl_cls91_1->dsc_g_idset1, 0, sizeof(struct dsd_sdh_ident_set_1)
      );
3    bol1 = adsp_hl_clib_1->amc_aux( adsp_hl_clib_1->vpc_userfld,
4                                    DEF_AUX_GET_IDENT_SETTINGS,  /* return
                                         settings of this user */
5                                    &adsl_cls91_1->dsc_g_idset1,
6                                    sizeof(struct dsd_sdh_ident_set_1) );
```

Liegen die benötigten Informationen dem SDH vor, so wird eine Gather-Kette erstellt, die auf die jeweiligen Speicheradressen verweist, um nachstehenden Control-Request 'REGISTER' zu erzeugen, wie sie bereits in Kap. 3.1.3 dargestellt bzw. definiert wurde:

REGISTER
config-name:test1
channel:1
SIP-Ident:11
INETA-SIP-gateway:172.5.0.1
SIP-shared-secred:geheim
INETA-this:172.5.0.1

Zur Erstellung einer entsprechenden Gather-Kette, die über die jeweiligen Speicheradressen die dargestellte Control-Request repräsentiert, wird die Funktion `m_extend_gather` implementiert. Die Funktion `m_extend_gather` soll den im Ablauf identisch wiederkehrenden Vorgang der Erweiterung einer Gather-Struktur definieren und wird im nachstehenden Quellcode dargestellt:

**Listing 3.70:** xltclib9.cpp -m_extend_gather

```
1  struct dsd_gather_i_1 *m_extend_gather( struct dsd_hl_clib_1 *adsp_clib_1,
2                                          struct dsd_gather_i_1 *adsp_gather
                                               _1,
3                                          char *achp_start_data,
4                                          char *achp_end_data, void **avo_wa_
                                               tmp )
5    {
6    struct dsd_gather_i_1 *adsl_gather_send_1 = NULL;
7    // valid parameters ?
8    if (adsp_gather_1 && adsp_clib_1)
9    {
10     if((adsp_gather_1->achc_ginp_end -adsp_gather_1->achc_ginp_cur)>0){
11 adsl_gather_send_1 = (struct dsd_gather_i_1*)m_wa_alloc((void**)avo_wa_tmp
       , sizeof(struct dsd_gather_i_1));
12     adsp_gather_1->adsc_next = adsl_gather_send_1;
13   }else{
14         adsl_gather_send_1 = adsp_gather_1;
15       }
16     adsl_gather_send_1->adsc_next = NULL;
17   adsl_gather_send_1->achc_ginp_cur = (char *)achp_start_data;
18   adsl_gather_send_1->achc_ginp_end = (char *)achp_end_data;
19   return adsl_gather_send_1;
20   }
21 }
```

Als Parameter werden hierzu unter anderem die zu erweiternde Gather-Struktur und die Zeiger auf die Speicheradressen, die die Sequenz an Informationen für die neue anzuhängende Gather-Struktur darstellt, übergeben. Die Überprüfung der Summe aus der Subtraktion der Zeiger für die Datensequenz der übergebenen zu erweiternden Gather-Struktur liefert, ob es sich bei der übergebenen Gather-Struktur um eine handelt, die bereits eine Datensequenz repräsentiert oder ob eine Gather-Struktur übergeben wurde, die das erste Glied einer Gather-Kette darstellt und noch nicht auf eine Datensequenz zeigt. Im Bezug auf den oben dargestellten Quellcode, stellt `adsl_gather_send1` den Rückgabewert dar. Unabhängig davon, ob `adsl_gather_send1` das erste oder letzte Glied einer Kette ist, werden für `CUR` und `END` immer die übergebenen werde für die entsprechenden Speicheradressen zugeordnet (Z.17-21 Listing 3.3.2).

Aus `adsl_gather_send1` wird für den Fall, dass eine Gather-Struktur die bereits auf eine Datensequenz zeigt übergeben wurde, eine neue Gather-Struktur erzeugt und als letzes Glied, der übergebenen Kette `adsp_gather_1`, angehängt (vgl. Z.10-13 Listing 3.3.2). Falls eine Gather-Struktur übergeben wurde, die noch nicht auf eine Datensequenz zeigt, stellt `adsl_gather_send1` das erste Glied einer Kette bzw. eine einzige Gather-Struktur dar. In diesem Fall wird keine neue Gather-Struktur erzeugt, sondern die übergebene Gather-Struktur als erstes Glied einer möglichen Kette verwendet (vgl. Z.13-15 Listing 3.3.2).

Der nachstehend abgebildete Ausschnitt aus dem Quellcode stellt die Generierung der weiter oben Dargestellten Control-Request 'REGISTER' entsprechend der Konstruktion von Gather-Ketten mithilfe der Funktion `m_extend_gather` dar.

**Listing 3.71:** xltclib9.cpp -REGISTER

```
/*REGISTER*/
 adsl_gai1_out_first = m_extend_gather( adsp_hl_clib_1 ,
          (struct dsd_gather_i_1 *)achl_work_1,
          (char *) chrs_sip_register,
          (char *) chrs_sip_register +
          sizeof(chrs_sip_register),&avo_wa_tmp);
    adsl_gai1_out_last = m_extend_gather( adsp_hl_clib_1 ,adsl_gai1_out_
        first,
                                    (char *) chrs_sip_configname,
                                    (char *) chrs_sip_configname +
                          sizeof(chrs_sip_configname),&avo_wa_tmp);

/*CONFIGNAME*/
     adsl_gai1_out_last = m_extend_gather( adsp_hl_clib_1 ,adsl_gai1_out_
         last,
                                    (char *) chrs_line,
                                    (char *) chrs_line+
                          sizeof(chrs_line),&avo_wa_tmp);

   adsl_gai1_out_last = m_extend_gather( adsp_hl_clib_1 ,adsl_gai1_out_
       last,
                                    (char *) chrs_channel,
                                    (char *) chrs_channel+
                          sizeof(chrs_channel),&avo_wa_tmp);

 /*CHANNEL*/
  adsl_gai1_out_last = m_extend_gather( adsp_hl_clib_1 ,adsl_gai1_out_last
      ,
                                     (char *) (char *)chrs_CHANNEL_SIP,
                                     (char *) chrs_CHANNEL_SIP+
                          sizeof(chrs_CHANNEL_SIP),&avo_wa_tmp);

      adsl_gai1_out_last = m_extend_gather( adsp_hl_clib_1 ,adsl_gai1_out_
          last,
                                     (char *) chrs_line,
                                     (char *) chrs_line+
                          sizeof(chrs_line),&avo_wa_tmp);

     adsl_gai1_out_last = m_extend_gather( adsp_hl_clib_1 ,adsl_gai1_out_
         last,
                                     (char *) chrs_sip_ident,
                                     (char *) chrs_sip_ident+
                          sizeof(chrs_sip_ident),&avo_wa_tmp);
.
.
```

```
.
/* send sip-registry data   */
 m_send_channel( adsp_hl_clib_1 ,adsl_gai1_out_first,
            D_CHANNEL_CONTROL,&avo_wa_tmp);
```

Wie aus dem Quellcode ersichtlich, werden zwei GathSer-Strukturen verwendet, um die erwünschte Gather-Kette mittels der Funktion `m_extend_gather` zu erhalten. Die als `adsl_gai_out_first` benannte Gather-Struktur wird der Funktion `m_extend_gather` als eine Gather-Struktur übergeben, die noch nicht auf eine Datensequenz zeigt und stellt somit das erste Glied der zu erzeugenden Kette dar. Wie in Kap. 3.1 definiert, ist für alle Datenpakete zwischen dem HOBPhone und dem SDH ein HOB-Header erforderlich, der mittels einer Channel-Nummer die transportierten Informationen zu einem Standard bzw. Protokoll zuordnet. Hierzu wird die Funktion `m_send_channel` implementiert, die durch nachstehenden Quellcode dargestellt wird.

**Listing 3.72:** xltclib9.cpp -m_send_channel

```
void m_send_channel ( struct dsd_hl_clib_1 *adsp_clib_1, struct dsd_gather
    _i_1 *adsp_gather_1, int iml_channel, void **avo_wa_tmp){

char        chl_more;
char*     achl1;
struct dsd_gather_i_1 *adsl_gai1_out;  /* last in chain of output data*/
int iml1 = 0;
adsl_gai1_out = (struct dsd_gather_i_1 *)m_wa_alloc(avo_wa_tmp,
                                 sizeof(struct dsd_gather_i_1));
memset(adsl_gai1_out, 0, sizeof(struct dsd_gather_i_1));
printf( "xltclib9 l %05d SEND TO CHANNEL = %d.\n", __LINE__,iml_channel );
adsl_gai1_out->adsc_next = adsp_gather_1;
 while(adsp_gather_1){
  iml1 += (adsp_gather_1->achc_ginp_end -adsp_gather_1->achc_ginp_cur);
  adsp_gather_1 = adsp_gather_1->adsc_next;
 }
  m_set_channel_gather(adsl_gai1_out, iml1, iml_channel, avo_wa_tmp );
 // queue gather
 if (adsp_clib_1->adsc_gather_i_1_out)
   adsp_clib_1->adsc_gather_i_1_out->adsc_next = adsl_gai1_out;
 else
   adsp_clib_1->adsc_gather_i_1_out = adsl_gai1_out;
}
.
.
.
}
```

Der Funktion `m_send_channel` wird die weiter oben erwähnte erste Gather-Struktur `adsl_gai_out_first` (Z.2 Listing 3.3.2) als Gather-Kette zur Darstellung der Control-Request-'REGISTER' die nun als erstes Glied eine Gather-Struktur hat, welche den

benötigten HOB-Header darstellt, übergeben (Z.40-43 Listing 3.3.2).

Da der HOB-Header aus einer Angabe der Länge der durch die Channel-Nummer korrespondierten Daten in NHASN beinhaltet, wird in der Funktion `m_send_channel` eine mit einem HOB-Header zu versehende Gahter-Struktur bzw. das erste Glied einer Gather-Kette erwartet, aus der dann die Länge der Datensequenz extrahiert wird (Z.13-16 Listing 3.3.2). Zur Ermittlung der Länge wird der oben dargestellten `while`-Schleife das erste Glied einer möglichen Gahter-Kette übergeben. Nach jedem Durchgang zeigt `adsp_gather_1` auf das nächste Glied der Gather-Struktur, dessen Datenlänge soeben ermittelt wurde. Zeigt der `NEXT`-Pointer dieser Gather-Struktur auf `null`, so ist keine weitere Gather-Struktur in der Kette vorhanden und die soeben bearbeitete Gather-Struktur stellte die letzte der Kette dar, bzw. war eine einzige Gather-Struktur.

Zeigt der `NEXT`-Pointer der `adsp_gather_1` zugewiesen wird auf `null`, so ist die Bedingung der `while`-Schleife `while(asdp_gather_1)` nicht mehr erfüllt, so dass die Schleife verlassen wird und `iml1` die Länge der durch die Gather-Kette dargestellten Daten beinhaltet. Die Funktion `m_set_channel_gather` erzeugt hierbei eine Gather-Struktur die den HOB-Header darstellt und verkettet diese mit der weiter oben dargestellten Gather-Kette, so dass im Gegensatz zur Funktion `m_extend_gather` der übergebenen Gather-Struktur eine neue als HOB-Header vorangestellt wird.

Im Bezug auf die Informationsrichtung von der VoIP-Architektur zum HOBPhone, erfolgt die Verarbeitung von Datenpaketen die mittels UDP von einer VoIP-Komponente empfangen wurden nach dem gleichen Prinzip, wie die Verarbeitung eines durch den SDH generierten Datenpaketes aus dem oben aufgeführten Beispiel. Wurde mittels UDP ein Datenpaket empfangen, so sind entsprechende bei Aufruf des SDH entsprechende Flags für SIP oder RTP gesetzt. Nachstehend wird die entsprechende Codezeile für die Abfrage des Flags für mittels UDP empfangene SIP-Datenpakete der Methode `m_reflect` dargestellt:

**Listing 3.73:** xltclib9.cpp -SIP received

```
1 if (adsp_hl_clib_1->imc_signal & HL_AUX_SIGNAL_IO_1) {
                              /* SIP something received  */
2 .
3 .
4 .
5 m_redirect_udp ( adsp_hl_clib_1, adsl_recb_1_w1, D_CHANNEL_SIP,&avo_wa_tmp
     );
6 }
```

Zur Weiterleitung von mittels UDP empfangenen Datenpaketen wird die Funktion `m_redirect_udp` implementiert. Zu beachten ist hierbei, dass bei der Implementierung des SDH in dieser Version, von explizit einem Asterisk-Account und einer Asterisk als

Komponente in der VoIP-Architektur ausgegangen wird. Demzufolge wird keine Differenzierung bzw. dynamische Vergabe von Channel-Nummern für verschiedene Asterisk-Accounts vorgenommen, sondern eine allgemeine Channel-Nummer für SIP-Datenpakete definiert.

Ähnlich wie beim Vorgehen zur Versendung von Control-Datenpaketen an das HOBPhone, werden bei der Weiterleitung entsprechende Gather-Strukturen erstellt, die auf zu versendende Speicheradressen Zeigen und somit die Informationen darstellen, die dem HOBPhone weitergeleitet bzw. versendet werden sollen. Der Unterschied zur Versendung von Datenpaketen, die als Control-Datenpakete vom SDH generiert wurden besteht lediglich darin, dass die erstellten Gather-Strukturen auf die Speicheradressen zeigen, die vom WSP verwendet wurden um die mittels UDP empfangenen Informationen dem SDH bereitzustellen.

Der nachstehende Ausschnitt aus dem Quellcode der Funktion `m_redirect_udp` zeigt, wie anhand der vom WSP bereitgestellt Daten `adsl_recb_1_w1` eine Gather-Struktur als `adsl_gai1_out_first` erzeugt wird, der im späteren Verlauf ein entsprechender HOB-Header vorangestellt wird.

**Listing 3.74:** xltclib9.cpp -m_redirect_udp

```
1 adsl_gai1_out_first = m_extend_gather(adsp_clib_1 ,adsl_gai1_chnl_first,
2                           adsl_recb_1_w1->achc_data,
3                           adsl_recb_1_w1->achc_data +
4                           adsl_recb_1_w1->imc_len_data,
5                           avo_wa_tmp);
```

Bei der Verarbeitung von mittels UDP-Empfangenen Daten ist zu berücksichtigen, dass es sich bei der die empfangenen Daten darstellenden Struktur `adsl_recb_1_w1` ebenfalls um eine Daten-Kette handelt und diese somit mehrere empfangene Datenpakete für die übergebene Channel-Nummer enthalten kann. Sind mehrere Datenpakete bei einem Aufruf für eine Channel-Nummer zu verarbeiten, so muss für jedes empfangene Datenpaket ein HOB-Header erzeugt und der Gather-Struktur, welche die empfangenen Daten darstellt, wiederum als Gather-Struktur vorangestellt werden. Im nachstehenden Ausschnitt aus dem Quellcode der Funktion `m_redirect_udp` wird der erwähnte Vorgang dargestellt.

**Listing 3.75:** xltclib9.cpp -loop over all buffers received

```
1 while (adsl_recb_1_w1) {                  /* loop over all buffers
      received */
2
3   adsl_recb_1_w1 = adsl_recb_1_w1->adsc_next;  /* get next in chain */
4   if(adsl_recb_1_w1){
5
6         adsl_gai1_chnl = (struct dsd_gather_i_1 *)m_wa_alloc(avo_wa_tmp,
              sizeof(struct dsd_gather_i_1));
```

```
7       memset(adsl_gai1_chnl, 0, sizeof(struct dsd_gather_i_1));
8           m_set_channel_gather(adsl_gai1_chnl,adsl_recb_1_w1->imc_len_data,
9                       iml_channel,avo_wa_tmp);
10      if(adsl_gai1_out_last){
11        adsl_gai1_out_last->adsc_next = adsl_gai1_chnl;
12      }else{
13      adsl_gai1_out_first->adsc_next = adsl_gai1_chnl;
14      }
15
16      adsl_gai1_out_last = m_extend_gather( adsp_clib_1 ,adsl_gai1_chnl,
17        adsl_recb_1_w1->achc_data,
18      adsl_recb_1_w1->achc_data + adsl_recb_1_w1->imc_len_data,avo_wa_tmp);
19
20    }
21    }
```

Wie aus dem oben dargestelltem Quellcode ersichtlich, wird jedes Datenpaket, auf welches in der Struktur `adsl_recb_1_w1` verwiesen wird, der weiter oben beschriebenen Funktion `m_set_channel_gather` übergeben. Die Verarbeitung von mittels RTP-empfangenen Datenpaketen erfolgt mit Unterscheidung der Channel-Nummer identisch. Ein Empfang von Datenpaketen für SIP kann bereits nach dem ersten Aufruf des SDH nach Versendung der Control-Request 'REGISTER' an das HOBPhone erfolgen.

Anhand der am WSP administrierten Account-Daten wird hierzu dem jeweiligen Asterisk-Account eine Channel-Nummer zugeordnet. Die Weiterleitung von mittels UDP empfangenen Datenpaketen für SIP vom WSP an den SDH, erfolgt durch den in Kap. 3.1 beschriebenen Ablauf der in Abb. 3.7 'Aufruf eines SDH' dargestellt wird, wobei die für den möglichen Empfang von RTP-Datenpaketen vorausgesetzten Dialoge in Kap. 3.1.3 'WSP-Protokoll' beschrieben wurden. Unter anderem auch im Bezug auf die für den Empfang von RTP-Datenpaketen vorausgesetzten Dialoge zwischen dem HOBPhone und dem SDH, ist eine nähere Betrachtung der Informationsrichtung vom HOBPhone über den SDH zur VoIP-Architektur erforderlich.

Bei Datenpaketen, welche vom HOBPhone über die SSL-Verbindungen empfangen wurden, erfolgt zunächst die Interpretation des erwarteten HOB-Header. Hierzu werden die empfangenen Daten ausgelesen und entsprechend der in Kap. 3.1 definierten Notation für den HOB-Header interpretiert. Die empfangenen Daten werden durch die Gather-Struktur `adsl_gai1_inp_1` dargestellt und können aus einer Kette von Datensequenzen bestehen, was in Abb. 3.17 verdeutlicht wird.

Demzufolge wird in Abb. 3.17 ein Datenpaket abgebildet, wie es bei einem Aufruf des SDH zwar komplett aber durch mehrere Gather-Strukturen als Datensequenzen dargestellt wird. Der Aufbau dieser Struktur erfolgt hierbei im WSP und wird mehreren Faktoren beeinflusst, wobei an dieser Stelle zu erwähnen ist, dass für die Implementierung des SDH die in Kap. 2.3 dargestellte Funktionalität des WSP als Basis dient und eine weitergehende Betrachtung der Implementierung des WSP nicht Gegenstand

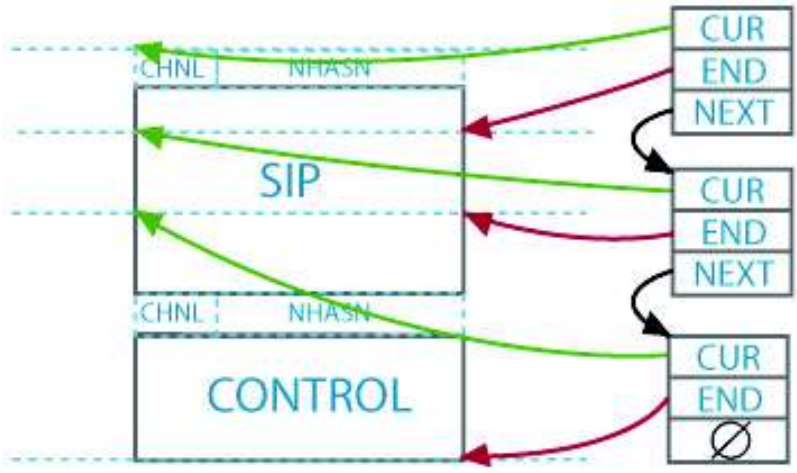


**Abbildung 3.17:** Darstellung einer Datensequenz mit Gather-Strukturen

des Konzeptes im Hinblick auf die in Kap. 3.1 definierte Architektur ist. Das Auslesen der Daten mittels der übergebenen Gather-Struktur `adsl_gai1_inp_1` erfolgt innerhalb einer 'While'-Schleife mit der Bedingung `while(adsl_gai1_inp_1)`.

Innerhalb der Schleife wird nach jeder Verarbeitung einer Datensequenz die durch eine Gather-Struktur `adsl_gai1_inp_1` dargestellt wird, diese durch den eigenen `NEXT`-Pointer referenziert. Zeigt der `NEXT`-Pointer der bearbeiteten Gather-Struktur auf eine weitere Gather-Struktur, ist die weiter oben aufgeführten Bedingung der Schleife erfüllt und es erfolgt die Verarbeitung der nächsten Gather-Struktur. Die der Verarbeitung erfolgt mittels einem Read-Pointer und einer Schlüsselnummer für den aktuellen Stand der Bearbeitung, die bei jedem Durchlauf der Schleife innerhalb einer `switch` Konstruktion unterschieden wird. Der Bearbeitungsstand als Status wird hierbei unterschieden in

- Interpretation des HOB-Header mit Channel und NHASN
- Interpretation der mit der Channel-Nummer korrespondierten Daten
- Abschluss der Verarbeitung

Die Konstruktion des `switch` mit dem Bearbeitungsstatus innerhalb der Schleife stellt eine sog. 'State-Machine' dar, die nach jedem Bearbeitungsschritt verlassen und nach erneuter Überprüfung der while-Bedingung mit neuem Bearbeitungsstatus wieder durchlaufen wird. Ist der Status zum Abschluss der Verarbeitung für die aktuelle Gather-Struktur erreicht, wird `adsl_gai1_inp_1` mit dem NEXT-Pointer referenziert und bei Existenz einer weiteren Gather-Struktur aufgrund der Gültigkeit der while-Bedingung der switch wieder durchlaufen wird usw.

Vor einer Interpretation der mit der Channel-Nummer korrespondierten Daten wird die erwartete Länge der Daten als Informationseinheit, die durch NHASN dargestellt wird, mit der tatsächlichen übergebenen Länge an Daten verglichen (Bearbeitungsstatus Interpretation des HOB-Header mit Channel und NHASN). Wenn die Größe zu verarbeitende Datensequenz welche durch die Gather-Struktur bzw. Kette `adsl_gai1_inp_1` dargestellt wird kleiner ist als die Länge die für dieses Datenpaket im NHASN angegeben wurde, handelt es sich um ein Datenfragment.

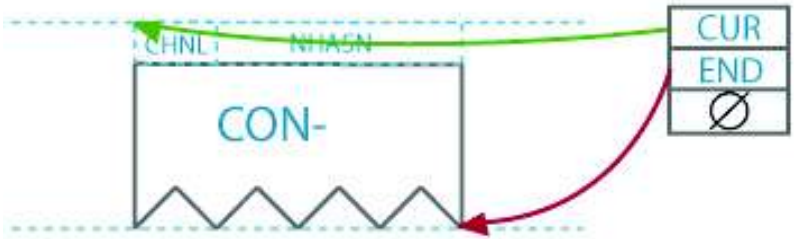


**Abbildung 3.18:** Datenfragmentierung im SSL-Modus

Die übergebene Gather-Struktur bei einem Aufruf des SDH, kann unabhängig davon ob es sich um einer Gather-Kette handelt oder nur um eine Gather-Struktur, ein Fragment einer Informationseinheit bzw. eines Datenpaketes darstellen oder aber auch mehrere Datenpakete, wobei z.B. das letzte Glied der Gather-Kette ein Datenfragment darstellen kann (vgl. Abb. 3.18).

Hierzu ist zu beachten, dass der `CURRENT`-Pointer der Gather-Struktur `adsl_gai1_inp_1`, welche die an den SDH übergebenen Daten darstellt, bei jedem Aufruf des SDH vom WSP so übergeben wird, wie er beim Verlassen des SDH eingestellt wurde. Verschiebt man nun vor dem Verlassen des SDH den `CURRENT`-Pointer auf die Speicheradresse, welche den Anfang eines neuen fragmentierten Datenpaketes darstellt, so erhält man beim nächsten Aufruf des SDH eine Gather-Struktur die exakt mit dem HOB-Header beginnt, wodurch auch die Übergabe einer Gather-Struktur an den SDH die nicht mit einem HOB-Header beginnt ausgeschlossen werden kann (vgl. Abb. 3.19).

Wird das erwartete Datenpaket immer noch nicht vollständig durch die übergebene Gather-Struktur bzw. Gather-Kette dargestellt, so wird der Aufruf des SDH schlicht beendet, so dass der SDH vom WSP wieder aufgerufen wird, wenn neue Daten eingetroffen sind. Die übergebene Gather-Kette vom WSP wird demzufolge, bis die durch NHASN ermittelte Länge und somit ein vollständiges Datenpaket erreicht ist, kontinuierlich verlängert.

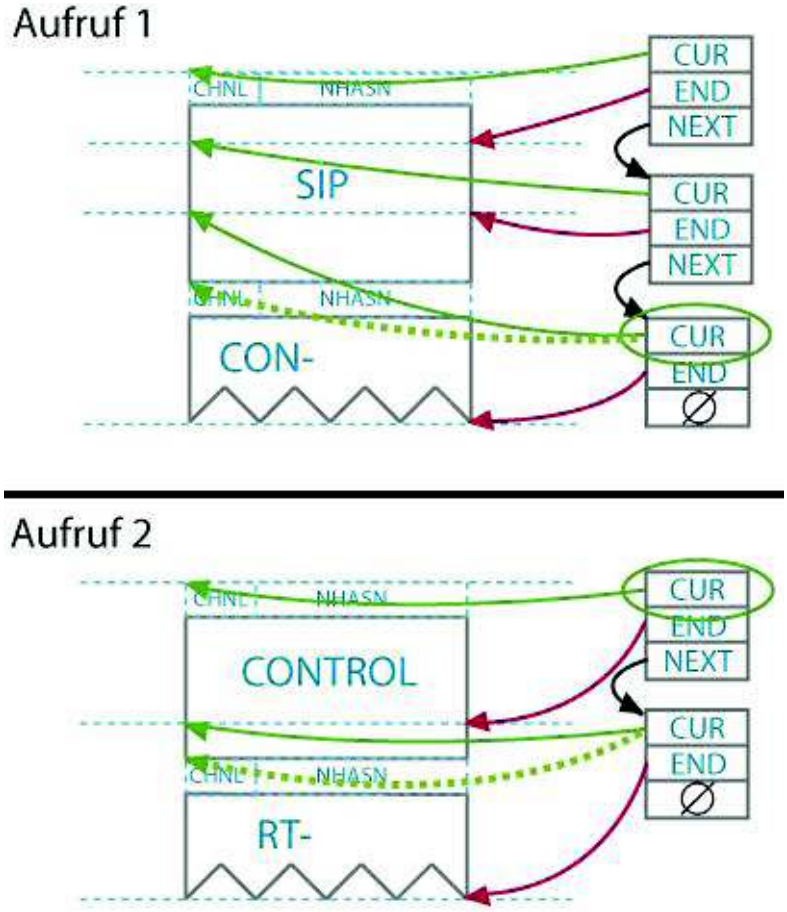


**Abbildung 3.19:** Verarbeitung fragmentierter Daten im SDH

# Kapitel 4

# Fazit und Ausblick

Das VoIP- Konzept für SSL- geschützte Internettelefonie wird mit der Implementierung der für diese Arbeit maßgeblichen Version im Kern umgesetzt. Die in Kap. 3.1 'Architektur des Konzeptes' festgelegte Definition für die Funktionalität der Komponenten ist in diesem Zusammenhang auf die Umsetzung des Konzeptes in seiner Grundfunktionalität ausgerichtet. In der weiteren Entwicklung der Komponenten HOBPhone und einem dem VoIP- Konzept korrespondierenden SDH am WSP, soll der Funktionsumfang aus technischer Sicht und in Zusammenhang dazu entsprechend auf Anwendungsebene erweitert werden.

Die Verarbeitung von RTP- Datenpaketen soll um die Themenbereiche Datenkompression und Bandbreitenmanagement ausgeweitet werden. Ausgehend von der grundsätzlichen Festlegung, die Vorraussetzungen zum Einsatz des HOBPhone auf ein minimum zu reduzieren, ist die Implementierung eines Datenkompressionsverfahrens auf Basis eines frei verfügbaren Codec angedacht (z.B. GSM160). Durch den Einsatz von eigens implementierten Datenkompressionsverfahren soll eine Unabhängigkeit von lokal verfügbaren Codecs gewährleistet werden, die mittels vorhandener Sound API's (vgl. 3.2.4.1'Steuerung der Audio- Komponenten') verwendet werden können. In Zusammenhang mit dem Einsatz von Codec´s ist auch die Implementierung entsprechender Aufnahme- und Wiedergabemechanismen auf Basis des in Kap. 2.1.3 beschriebenen Real- Time Transport Protocol (RTP) entsprechend dem Codec, der eingesetzt werden soll, erforderlich.

Wie bereits in Kap. 3.3.2 'Verarbeitung von Datenpaketen im SDH' erwähnt wurde, steht ferner die Erweiterung des SDH zur Unterstützung der in Kap. 3.1 'Architektur des Konzeptes' dargestellten VoIP- Architektur mit SIP- Clients, welche an $n$ Asterisk Telefonanlagen jeweils $n$ SIP- Accounts haben können.

Ausgehend von der in Kap. 3.2.5 'Grafische Benutzeroberfläche' dargestellten Architektur zur Implementierung von möglichen Benutzeroberflächen, ist eine Implementierung von verschiedenen Benutzeroberflächen möglich, die der Benutzer individuell auswählen kann. Hierbei sollen Benutzeroberflächen als 'Farbschemen' oder 'Themenbereiche' (z.B.

'Aqua', 'Futuristic') zur Verfügung stehen. Abb. 4.1 veranschaulicht einen Entwurf für eine Benutzeroberfläche mit dem Thema 'Futuristic'.

**Abbildung 4.1:** Grafische Benutzeroberfläche 'Futuristic'

Die Erweiterung der Benutzeroberflächen steht hierbei auch in Zusammenhang mit einer Erweiterung der grundsätzlichen Funktionalität des HOBPhone. Demnach sollen dem Benutzer Funktionen wie z.B. Konferenzen, Insant- Messaging oder die Übertragung eines Benutzerfotos bei Anrufen zur Verfügung stehen, wobei auch auf Basis der in dieser Version eingesetzten Protokolle und in Kap. 3.1 definierten Architektur die Erweiterung des HOBPhone auf Echtzeitbildübertragung und somit auf ein Video- Telefon möglich wäre.

Das VoIP- Konzept für SSL- geschützte Internettelefonie erweist sich in der Praxis als Konzept, welches Problemstellungen im VoIP- Umfeld nicht nur aus Sicherheitsaspekten eine Lösung bietet. Durch Berücksichtigung von in der Praxis eingesetzter Architekturen und der tatsächlichen Nutzung von einheitlichen Protokollen, kann im Bezug auf Kompatibilität, Sicherheit und Flexibilität das VoIP- Konzept für SSL- geschützte Internettelefonie wirtschaftlich sinnvoll und effizient eingesetzt werden.

# Literaturverzeichnis


[1] Alan B. Johnston. *SIP Understanding the Session Initiation Protocol*. Artech House telecommunications library, 2003.

[2] V. Jacobson M. Handley. *RFC 2327 SDP: Session Description Protocol*. The Internet Society, http://www.ietf.org/rfc/rfc2327.txt, 1998.

[3] Jim Van Meggelen. *Asterisk The Future of Telephony*. O´Reilly, 2005.

[4] Colin Perkins. *RTP Audio and Video for the Internet*. Addison- Wesley, 2003.

[5] Dr. Kai-Oliver Detken Prof.Dr. Evren Eren. *VoIP security : Konzepte und Lösungen für sichere VoIP-Kommunikation*. Hansen, 2007.

[6] R. Rivest. *RFC 1321 - The MD5 Message-Digest Algorithm*. MIT Laboratory for Computer Science and RSA Data Security,Inc., http://www.ietf.org/rfc/rfc1321.txt, 1992.

[7] J. Rosenberg. *RFC 3261 SIP Session Initiation Protocol*. The Internet Society, http://www.ietf.org/rfc/rfc3261.txt, 2002.

[8] Stephen Thomas. *SSL and TLS Essentials*. Wiley, 2000.

[9] Herbert Voß. *Kryptografie mit JAVA*. Franzis, 2006.

# Anhang A

# Screenshots

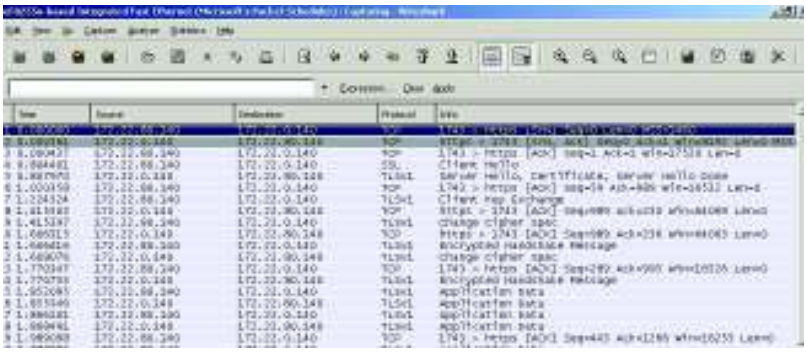

**Abbildung A.1:** Wireshark Capture: SSL-Verbindungsaufbau

**Abbildung A.2:** Entwicklungsumgebung eclipse, Package Explorer