Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Volker Gropp

Voice over IP Qualitätstests auf Basis der Android Plattform

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Volker Gropp

Voice over IP Qualitätstests auf Basis der Android Plattform

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Gunter Klemke
Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am 19.12.2008

Volker Gropp

Thema der Bachelorarbeit

Voice over IP Qualitätstests auf Basis der Android Plattform

Stichworte

Android, Dalvik, VoIP, SIP, RTP, Qualitätstests, QoS, QoE, MOS, JNI, Analyse, Diagnose, QoA, QoD, Interarrival Time, Jitter, Paketverlust

Kurzzusammenfassung

Die hier beschriebene Arbeit befasst sich mit Voice over IP-Qualitätstests unter dem Android-Framework der Open Handset Alliance. Die Voice over IP-Qualität wird mit Hilfe der Technologie von VoIPFuture untersucht. Zur Durchführung der Qualitätstests wird ein Voice over IP-Softphone für das Android-Framework exemplarisch entwickelt und die VoIPFuture Library zur Analyse und Diagnose von Voice over IP auf Android portiert.

Volker Gropp

Title of the paper

Testing the Quality of Voice over IP on Android

Keywords

Android, Dalvik, VoIP, SIP, RTP, Qualitytests, QoS, QoE, MOS, JNI, Analysis, Diagnosis, QoA, QoD, Interarrival Time, Jitter, Packet Loss

Abstract

This thesis is about measuring the quality of Voice over IP using the technology by VoIPFuture on Android developed by the Open Handset Alliance. To analyze and diagnose the Voice over IP quality, it is necessary to develop a Voice over IP Softphone for Android and port the VoIPFuture Library to this new platform.

Danksagung

Hiermit möchte ich mich bei Herrn Prof. Dr. Gunter Klemke für die freundliche Betreuung bedanken. Ebenfalls danken möchte ich Herrn Prof. Dr. Olaf Zukunft, der sich bereit erklärt hat, das Zweitgutachten zu übernehmen.

Ferner möchte ich meinen Kollegen von VoIPFuture für das interessante Thema danken und dafür, dass sie mich offen aufgenommen, mir eine angenehme Arbeitsatmosphäre und interessante Diskussionen geboten haben.

Weiter möchte ich mich bei meiner Freundin Berit Salchow bedanken, die sich unermüdlich Zeit genommen und mir bei der Korrektur geholfen hat. Außerdem danken möchte ich Jens Mayer und meiner Schwester Andrea Gropp, die in den letzten Tagen wertvolle Hinweise und Korrekturen gegeben haben.

Insbesondere danke ich herzlich meinen Eltern, die mich während meiner ganzen Studienzeit unterstützt und alle Entscheidungen mitgetragen haben.

Volker Gropp, Dezember 2008

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

Voice over IP (VoIP) wird in immer mehr Umgebungen verwendet. Es ist nach Plain Old Telephone System[1] (POTS) und Public Switched Telephone Network[2] (PSTN) die Weiterentwicklung der Telefonie. Im Gegensatz zu den bisherigen herkömmlichen, leitungsvermittelten Telefonnetzen, handelt es sich bei VoIP-Netzen um paketvermittelte Netzwerke, wie zum Beispiel das Internet. Auch die Endgeräte sind bei VoIP intelligenter und können in die vorhandenen IT-Systeme integriert werden. Es ist zum Beispiel möglich aus den zentral verwalteten Kontakt-Adressen einen Anruf aufzubauen. Auf Mobiltelefonen ist VoIP dagegen noch nicht sehr verbreitet. Dies ändert sich jedoch zur Zeit durch Universal Mobile Telecommunications System (UMTS)-Flatrates zunehmend. Außerdem ist es durch VoIP möglich, dass der Anwender immer unter der gleichen Rufnummer erreichbar ist.

Da die Anwender allerdings die sehr zuverlässigen Lösungen der klassischen Telefonie gewöhnt sind, besteht die Notwendigkeit die Qualität bei VoIP zu untersuchen und zu überwachen. Insbesondere gilt dieses für neue und unbekannte Umgebungen, sowie für neue Plattformen wie der Android-Plattform.

Das von Google entwickelte Android bietet eine neue vielversprechende offene Plattform. Nach dem iPhone verspricht Android die nächste Revolution im Mobiltelefon-Markt. Viele neue und interessante Konzepte auf mobilen Geräten werden durch Google in Android realisiert. Neben Google stehen viele namhafte Unternehmen in der Open Handset Alliance[3] (OHA) hinter Android, was die kommenden Möglichkeiten dieser Plattform belegt. Vor allem aber die von Google und der OHA versprochene Offenheit bezüglich der Erweiterbarkeit und des Quellcodes, können Grundlegende Änderungen bei den Mobiltelefon-Plattformen nach sich ziehen.

1.1 Aufgabenstellung und Zielsetzung

Ziel dieser Arbeit ist es, die Qualität von VoIP-Verbindungen unter Android zu messen. Deshalb gilt es zu klären, welche Einschränkungen diese neue Plattform bezüglich VoIP und der Entwicklung dafür enthält. Es muss untersucht werden, ob Android bereits eine Unterstützung für VoIP mitbringt oder zumindest generell dafür geeignet ist. Für die VoIP-Qualitätstests besteht die Notwendigkeit einen Software-VoIP-Client für Android zu entwickeln, in dem die Qualitätsanalyse stattfinden kann. Eine prototypische Realisierung reicht

[1] Das analoge Telefonnetz, das elektromechanisch vermittelt wurde, nur reine Telefonie erlaubte, und dabei auch keine weiteren Dienstmerkmale unterstützte.

[2] Das auch heute noch verwendete digital vermittelte Telefonnetz mit erweiterten Dienstmerkmalen.

[3] Zusammenschluss von über 30 Firmen unter der Federführung von Google. Von der Open Handset Alliance wird das Android-Framework entwickelt.

hierfür, so dass einfache VoIP-Verbindungen hergestellt werden können. Die VoIP-Qualitätsanalyse und die daraus abgeleitete Diagnose soll dabei mit der Technologie von VoIPFuture durchgeführt werden. Dafür wird die VoIPFuture Library auf die Android-Plattform portiert sowie in die zu entwickelnde VoIP-Anwendung integriert. Am Ende dieser Arbeit soll es möglich sein, anhand der Analyse-Daten der VoIPFuture Library mit Hilfe von weiteren Werkzeugen und Lösungen von VoIPFuture Diagnosen über die Qualität von VoIP unter Android zu erstellen.

Es ist vor Beginn der Arbeit unklar, welche Probleme bezüglich VoIP und der Durchführung von VoIP-Qualitätstests mit der Technologie von VoIPFuture unter Android zu erwarten sind. Daher muss die Realisierbarkeit geprüft und für auftretetende Schwierigkeiten Lösungen erarbeitet werden. Außerdem ist die Portierbarkeit der Library auf eine Android-ARM-Architektur zu prüfen, da die VoIPFuture Library vor dieser Arbeit ausschließlich auf der x86-Architektur getestet und verwendet wurde.

Die vorliegende Arbeit beschäftigt sich ausschließlich mit der Realisierung von VoIP-Qualitätstests auf der Android-Plattform. Andere Mobiltelefon-Systeme werden nicht näher betrachtet oder untersucht. Es wird jedoch ein Vergleich der Android-Plattform zu Linux-VoIP-Clients gezogen. Darüber hinaus werden nur VoIP-Qualitätstests mit Hilfe der Technologie von VoIPFuture durchgeführt. Diese Technologie wird nicht durch Tests mit anderen Möglichkeiten verglichen oder überprüft.

1.2 Inhaltlicher Aufbau der Arbeit

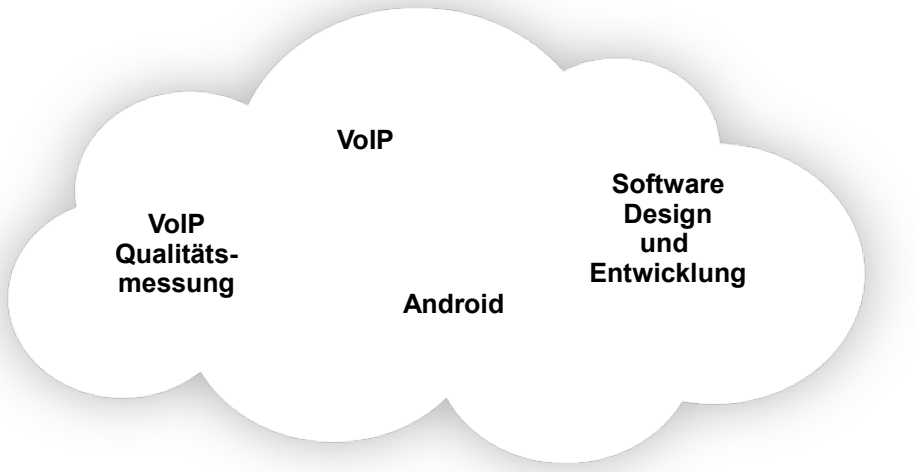


Abbildung 1.1: Umfang dieser Arbeit

Obwohl diese Arbeit sich vor allem mit der Messung der VoIP-Qualität durch die Analyse und Diagnose von VoIP-Verbindungen unter Android beschäftigt, berührt die Umsetzung weitere umfangreiche Themen, unter anderem Android und Software-Design und -Entwicklung, wie in Abbildung 1.1 angedeutet.

Im Rahmen dieser Arbeit werden in Kapitel 2 zuerst die Grundlagen von VoIP, der Qualitätsmessung von VoIP und der neuen Plattform Android erarbeitet. Im weiteren Verlauf werden in Kapitel 3 sowohl die Anforderungen an ein Android-VoIP-Client definiert als auch eine genaue Analyse des Android-Frameworks durchgeführt. Auf Grundlage der Analyse werden in Kapitel 4 die Android-VoIP-Anwendungen mit Hilfe von Methoden des Softwaredesigns entworfen und dieser Entwurf vorgestellt. Das Kapitel 5 „Realisierung der Android-VoIP-Anwendung“ beschreibt den daraus entwickelten Prototypen und einzelne Aspekte der Realisierung. In Kapitel 6 werden die durchgeführten VoIP-Qualitätstests beschrieben und die Ergebnisse vorgestellt. Den Abschluß bildet das Fazit gibt es einen Rückblick auf die Ergebnisse dieser Arbeit und ein Ausblick auf mögliche weitere Entwicklungen.

1.3 Veränderungen während dieser Arbeit

Diese Arbeit war bezüglich der Android-Plattform ständigen Veränderungen unterworfen. Sie wurde ein halbes Jahr nach der Vorstellung von Android im Juni 2008 begonnen. Zu dieser Zeit war bereits ein von der initialen Version aus Dezember 2007 weiterentwickeltes Software Development Kit[4] (SDK) in Version m5-rc15 verfügbar, welches über mehrere Wochen unverändert blieb. Im Laufe der Arbeit wurde Ende August 2008 das SDK 0.9_r1 veröffentlicht und am 23. September schließlich die Version 1.0r1. Viele Fehler des SDK in Version m5-rc15 wurden in dieser aktuellen Version behoben und die nun vorliegende Version entspricht laut Google dem Stand der Software auf dem ersten Android-Handy HTC G1.

Die im Laufe dieser Arbeit entstandenen Anwendungen wurden für das SDK m5-rc15 entworfen und entwickelt. Sie wurden nur so weit an die Version 1.0r1 angepasst, dass sie unter der neuen Version ausgeführt und benutzt werden können. Neue Funktionen und Erweiterungen des SDK 1.0r1 wurden nicht berücksichtigt oder verwendet.

[4] Ein Software Development Kit ist die Zusammenstellung einer Umgebung aus verschiedenen Programmen und Werkzeugen für die Entwicklung von Software. Es kann wie im Fall von Android zum Beispiel einen Emulator, Bibliotheken und Erweiterungen für die Entwicklungsumgebung beinhalten.

2 Grundlagen

In diesem Kapitel werden die Grundlagen für VoIP und die Qualitätsmessung von VoIP erklärt. Dazu gehören die VoIP-Protokolle, der Aufbau von VoIP-Netzen und die Vorstellung von Methoden zur Kontrolle und Umsetzung von VoIP-Qualität. Außerdem wird ein Einblick in die Technologie, Werkzeuge und Produkte von VoIPFuture gegeben. Es folgt eine Einführung in die neue Mobiltelefon-Plattform Android gegeben und diese, sowie vergleichbare Plattformen, auf vorhandene Lösungen von VoIP und der Qualitätsmessung untersucht.

2.1 Netze im Wandel

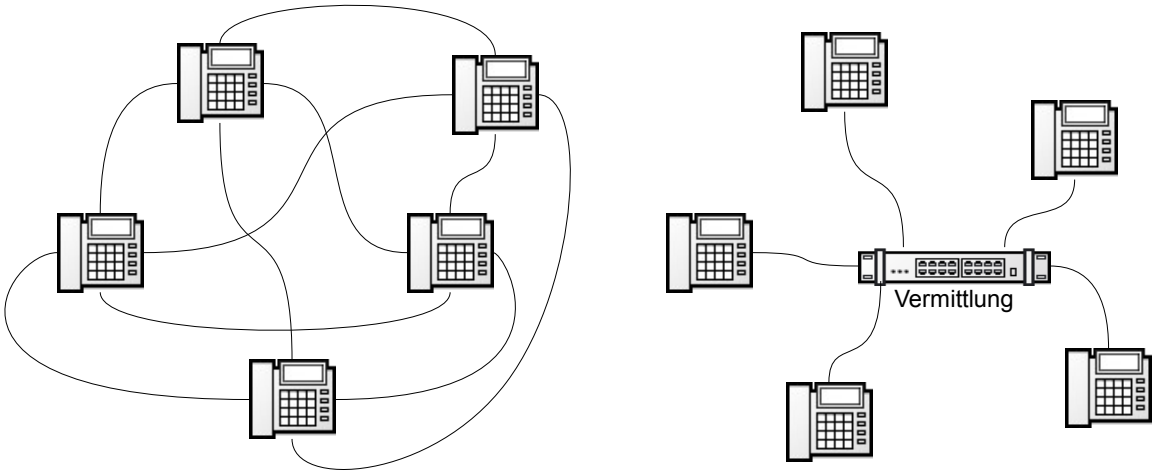
Schon 1976 wurde das Network Voice Protocol entwickelt. Ein Verfahren zur Übermittlung von Sprache über das ARPANET, dem Vorläufer des heutigen Internet. 1996 entwickelte die Firma VocalTec Telefonie über das Internet Protocol (IP), welche anfangs nach dem CB-Funk Prinzip funktionierte. Das bedeutet es konnte jeweils nur ein Teilnehmer sprechen. Diese Technik wurde kontinuierlich verbessert und führte 1998 und 1999 zu den noch heute verbreiteten VoIP Protokollen H.323 und SIP.

Gleichzeitig stieg der Datentransport über paketorientierte Netze, wie das Internet, extrem an und der Umfang der leitungsorientiert vermittelten Gespräche über das Telefonnetz veränderte sich kaum. Dadurch gewannen Datennetze immer mehr an Bedeutung und mittlerweile werden die Telekommunikationsnetze nicht mehr für Sprachverkehr, sondern für den paketorientierten Datenverkehr ausgelegt. (vgl. Siegmund, 2002)

2.1.1 Klassische Telefonnetze - leitungsorientierte Vermittlung

Die ersten Telefonnetze waren *Mesh Netzwerke* (vgl. Abbildung 2.1), unvermittelte Netze in denen jeder Teilnehmer mit jedem anderen direkt verbunden war. Diese stießen aber sehr schnell an Grenzen, da die Leitungsanzahl quadratisch mit der Teilnehmeranzahl wächst. Daher wurden elektromechanische Vermittlungen aufgebaut, mit denen bei einem Telefongespräch die Leitungen physikalisch zusammengeschaltet wurden. Dieses Prinzip war noch bis Mitte der 1980er Jahre aktuell und wird heute als POTS bezeichnet.

Danach setzten sich digital vermittelte Netze durch, die aber weiterhin leitungsorientiert waren und als PSTN bezeichnet werden. Dieses Netz war auch die Basis für die Verbreitung von Integrated Services Digital Network (ISDN), welches als *ein Netz für alles* gedacht war. Es stellte sich aber heraus, dass die 64kbit/s pro Kanal auf Dauer nicht ausreichend sind. (vgl. Siegmund, 2002; Wallingford, 2005)



(a) Ein unvermitteltes Mesh Telefonnetz

(b) Ein vermitteltes Telefonnetz

Abbildung 2.1: Verschiedene Telefonnetze

2.1.2 Next Generation Networks - paketorientierte Vermittlung

Die aktuellen Netze wie ISDN, die Mobilfunknetze Global System for Mobile Communications (GSM) und UMTS, sowie die privaten IP-Netzwerke, sind bereits digitale Netze, in denen Bitströme übermittelt werden. Um die Ressourcen dieser Netze besser nutzen zu können, ist es das Ziel, dass die verschiedenen Netze zu einem einzelnen Netz zu konvergieren und dabei mehrere Single-Service-Netze zu einem Multiservice-Netz zusammenzufassen. In diesem Multiservice Netz liegt der Schwerpunkt klar auf dem Datenservice.

Die Umstellung vollzieht sich allerdings nur langsam, da ein Multiservice Netz zwar viele Vorteile bietet, aber auch für den Betreiber komplexer und damit komplizierter ist. Daher wird es auch weiterhin den klassischen PSTN-Anschluss geben. Neue Anbieter ohne bestehende Netze werden aber vor allem auf Next Generation Network[1] (NGN) setzen. Für den Anwender verändert sich dabei nicht viel, da Media Gateways unbemerkt für den Austausch zwischen den verschiedenen Netzen sorgen. Die für die Signalisierung zuständigen Media Gateway Controller verwalten dabei eine Vielzahl dieser Media Gateways.

Durch die Netzkonvergenz entstehen niedrigere Kosten beim Netzmanagment aufgrund einheitlicher Technik, Wiederverwendung vorhandener Infrastrukturen und deren bessere Auslastung sowie bessere Skalierbarkeit des Netzes. Außerdem wird auch die Mobilität des Nutzers ermöglicht, der sich an jedem Anschluss des Netzes als Teilnehmer unter seinen Kontaktdaten registrieren kann.

In den klassischen, leitungsvermittelten Telefonnetzen gibt es sehr komplexe Netztechnik, die Endgeräte hingegen sind technisch relativ einfach gehalten. Diese Eigenschaft kehrt sich bei NGN um. Dort ist die Netztechnik relativ einfach, weswegen die Endgeräte zusätzliche Aufgaben übernehmen, wie zum Beispiel Qualitätsmessung und Steuerung. So werden in leitungsvermittelten Netzen bei Überlast neue Verbindungen

[1] Ein Telekomunikations-Netzwerk, das Daten-, Video- und Sprachkommunikation in einem Netz vereint. Realisiert wird es über das Internet Protocol.

abgelehnt, um die bestehenden Verbindungen nicht zu stören. Ein IP-Netzwerk reagiert anders. Dort treten bei Überlast Verzögerungen von Paketen auf und im extremen Fall werden einzelne Pakete verworfen. In einem NGN müssen daher zusätzliche Maßnahmen durchgeführt werden, um für Echtzeitdienste wie VoIP eine zufriedenstellende Qualität zu erreichen. Einige dieser Maßnahmen können unter Quality of Service[2] (QoS) zusammengefasst werden (vgl. Kapitel 2.2.3.1).

Ein NGN ist im Gegensatz zu den PSTN paketvermittelt. Daher ist es für VoIP notwendig eine Leitungsvermittlung durch Protokolle in der Sitzungsschicht nach dem OSI-Modell[3] aufzubauen. Hierfür werden durch Steuerungsprotokolle wie zum Beispiel das Session Initiaton Protocol (SIP) (vgl. Kapitel 2.2.1.1) Verbindugen aufgebaut und damit Leitungen emuliert. Für den Anwender besteht diese Verbindung scheinbar ununterbrochen. Aus technischer Sicht aber ist die Verbindung virtuell und belegt nur Ressourcen, wenn diese durch Daten verwendet werden.

Wie in den aktuellen PSTN ist auch bei VoIP die Signalisierung, und damit die Verbindungssteuerung strikt vom Nutzdatentransport getrennt. Es ist dadurch auch möglich, ein IP-Netzwerk zwischen mehreren klassischen PSTN als Backbone zu verwenden. Dabei ist es für den Anwender nicht ersichtlich, dass es sich auch in diesem Fall um VoIP handelt. Er telefoniert wie gewohnt mit seinem PSTN-Telefon.

In einem klassischen Telefonnetz gibt es hohe Anforderungen an die Verfügbarkeit und Sicherheit. So erreicht ein solches Netz weit über 99 Prozent Verfügbarkeit. Es ist eine große Herausforderung, diese Verfügbarkeit auch in einem IP-Netzwerk und damit in einem NGN zu erreichen. Zudem werden neue Konzepte für eine Absicherung des Netzes benötigt. (vgl. Siegmund, 2002; Badach, 2007; Trick und Weber, 2007; Wallingford, 2005)

2.2 Voice over IP

Lange wurde der Durchbruch von VoIP vorrausgesagt, der aber nicht eintraf. Die Umstellung vollzieht sich langsam und für den Endanwender oft kaum merklich. Bei vielen Anwendern hat VoIP einen schlechten Ruf, vor allem durch Implementierungsfehler in den ersten Versionen der Geräte. Zudem war die Benutzung von VoIP-Software für den Computer für die meisten Anwender zu umständlich oder kompliziert. Mittlerweile gibt es ausgereifte VoIP-Telefone und VoIP ist in WLAN- sowie DSL-Router integriert und wird von Service-Providern als einfach anzuschließendes Gerät vertrieben.

Unternehmen können mit VoIP, durch die Vereinfachung und Konvergenz von Netzen, häufig Kosten eingespart und die Komplexität der Netze vereinfachen (vgl. Kapitel 2.1.2). So ist es auch möglich, zwei Standorte eines Unternehmens durch VoIP miteinander zu verbinden, wodurch Telefonkosten eingespart werden können. (vgl. Badach, 2007)

Im Folgenden werden die für VoIP verwendeten Protokolle, unterschiedliche Ansätze zur Sicherung und Überwachung der VoIP-Qualität. Außerdem wird die Technologie und Anwendungen von VoIPFuture vorgestellt.

[2] Beschreibt die Qualität eines Dienstes in einem paketorientierten Netz aufgrund von festgelegten Parametern.

[3] Ein Netzwerk-Modell, das die Kommunikation vertikal in einzelne Schichten aufteilt, die aufeinander aufbauen.

2.2.1 Verwendete Protokolle bei VoIP

Grundlegend gliedert sich ein VoIP-Telefonat in den Verbindungsaufbau, die daran anschließende Nachrichtenübertragung und den Verbindungsabbau am Ende. Für die Verbindungssteuerung bzw. Signalisierung, also den Verbindungsaufbau, Verbindungsabbau sowie die weitere Beeinflussung der Verbindung während des Telefonats, werden dabei Protokolle wie SIP und H.323 verwendet. Für die Nachrichtenübertragung der Audiodaten wird vor allem Real-time Transport Protocol (RTP) verwendet.

2.2.1.1 SIP

Bei der Signalisierung findet zur Zeit SIP die häufigste Verwendung. Dabei ist es mit SIP nicht nur möglich VoIP zu steuern, sondern jegliche Medien-Übertragung über ein IP-Netzwerk. Für die Verwendung mit VoIP erfüllt SIP den selben Zweck wie die Signalisierung in einem PSTN. Erstmals wurde SIP 1999 von der IETF in Form des RFC 2543 standardisiert. Später kamen die in RFC 3261 bis 3265 beschriebenen Erweiterungen hinzu. SIP wird auch innerhalb von IP Multimedia System (IMS) im Mobilfunkstandard UMTS nach 3GPP Release 5 für die Signalisierung verwendet.

SIP bietet im Vergleich zu H.323 (vgl. Kapitel 2.2.1.3) eine einfache Architektur. Da SIP Handshake-, Wiederholungs- und Timeout-Verfahren implementiert, kann es neben dem verbindungsorientiertem Transmission Control Protocol (TCP) auch über verbindungslose Netzwerkprotokolle wie das User Datagram Protocol (UDP) laufen.

Neben dem Auf- und Abbau von Verbindungen ist es mit SIP auch möglich, bereits bestehende Verbindungen zu beeinflussen. So kann zum Beispiel eine Codec-Änderung angekündigt oder auch ein Stream pausiert werden. Außerdem kann ein Anwender sich mit SIP bei einem Registrar-Server anmelden (vgl. Kapitel 2.2.2.2). Es ist mit dem SIP aber auch möglich, direkte Punkt-zu-Punkt-Verbindungen aufzubauen und eine Gegenstelle direkt anzuwählen. (vgl. Siegmund, 2002; Badach, 2007; Trick und Weber, 2007)

Aufbau einer SIP-Nachricht

SIP ist in seinem Aufbau dem Hypertext Transfer Protocol (HTTP) ähnlicher als anderen klassischen Signalisierungsprotokollen bei ISDN oder GSM. Deutliche Spuren von HTTP finden sich in den Statuscodes bei Antworten, wie zum Beispiel der Fehler „404 not found“. SIP basiert auf UTF8, wodurch die Nachrichten nicht dekodiert werden müssen.

Eine Nachricht besteht aus einer Statuszeile, einem Header und einem optionalem Body mit Session Description Protocol (SDP)-Informationen. In der Statuszeile ist die SIP-Version, bei Antworten der Statuscode und bei Anfragen das Kommando enthalten. Der Header der SIP-Nachricht besteht aus Paaren eines Schlüsselworts mit einem Wert, wobei diese durch einen Doppelpunkt getrennt werden (z. B. `Max-Forwards: 70`). Schlüsselwörter dürfen mehrfach auftreten, so können zum Beispiel mehrere `Contact`-Einträge vorhanden sein. Dabei ist es auch zulässig diese Einträge zu einem zusammenzufassen und die verschiedenen Wert durch ein Komma zu trennen. Die optionalen SDP-Einträge sind vom Header durch eine Leerzeile getrennt,

```
INVITE sip:volker@192.168.1.159 SIP/2.0
Header  Via: SIP/2.0/UDP 192.168.1.150:5060; branch=1
From: sip:volker@192.168.1.159;tag=29ae1a33
Max-Forwards: 70
To: sip:volker@192.168.1.157
Call-ID: 47df467de7a@voipcall1
Cseq: 1 INVITE
Contact: sip:volker@192.168.1.159
Contact: sip:volker@192.168.1.150
Content-Length: 202
Supported: 100rel
Content-Type: application/sdp

v=0
o=Anonymous 1234567890 1234567890 IN IP4 192.168.1.150
s=Call to Volker
c=IN IP4 192.168.1.150
t=0 0
m=audio 6006 RTP/AVP 8 3 0
a=rtpmap:8 PCMA/8000
a=rtpmap:3 GSM/8000
a=rtpmap:0 PCMU/8000
```

Listing 2.1: Beispiel einer SIP-Anfrage mit SDP

wobei ein Zeilenumbruch bei SIP grundsätzlich aus den Zeichen mit den Dezimal-Werten 13 und 10 besteht.

Eine SIP-Adresse ist ähnlich einer E-Mail-Adresse aufgebaut. Sie besteht aus dem voran gestellten Schlüsselwort `sip:` und einem Benutzernamen, der durch ein „@“ von einer IP Adresse oder einer Domain (z. B. `sip:volker@gropp.org` oder `sip:volker@192.168.0.1`) getrennt wird. (vgl. Siegmund, 2002; Trick und Weber, 2007)

SIP-Kommunikation

SIP unterscheidet grundsätzlich zwischen Anfragen (Requests) sowie Antworten (Responses). Anfragen leiten Transaktionen ein, die entweder durch eine Antwort von der Gegenstelle erwidert werden oder mindestens durch ein `OK` oder ein `ACK` bestätigt werden. Dabei wird eine Transaktion durch einen eindeutigen Code (ID) gekennzeichnet, der in der Antwort verwendet werden muss.

Neben Transaktionen gibt es in der SIP-Kommunikation so genannte Dialoge. Diese bestehen aus einer oder mehreren Transaktionen, also Anfragen mit dazu passenden Antworten. Nur die Anfragen `INVITE`, `SUBSCRIBE` und `REFER` bauen einen Dialog auf. Ein Beispiel eines SIP-Dialogs ist ein Anruf, der erst angenommen und später beendet wird. Die erste Transaktion ist die Einladung `INVITE`, die im positiven Fall mit einem `OK` und einem darauf folgenden `ACK` beantwortet wird. Damit ist die Transaktion beendet. Der Dialog aber bleibt bis zum Ende des Anrufs und damit bis zur letzten Transaktion (`BYE` mit einem `ACK`) bestehen.

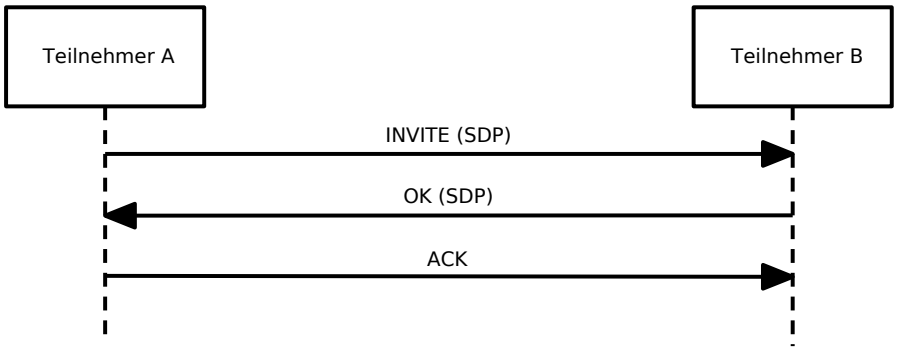


Abbildung 2.2: 3-Way-Handshake beim Aufbau einer SIP-Verbindung

So wie die Transaktionen, werden auch Dialoge durch einen eindeutigen Code gekennzeichnet, der in allen zum Dialog gehörenden Nachrichten verwendet werden muss.

SIP benützt für den Aufbau einer Verbindung einen „3-Way-Handshake“ (vgl. Abbildung 2.2). Dadurch lassen sich auch Codecs mittels SDP und dem „offer/answer“ Modell aushandeln. Dabei bietet der erste Teilnehmer einen oder mehrere Codecs zur Verwendung an, von denen der zweite Teilnehmer einen bestimmten Codec auswählt. (vgl. Trick und Weber, 2007)

Session Description Protocol

SDP wurde nicht speziell für SIP entwickelt, sondern in RFC4566 von der IETF spezifiziert. Für die SIP-Kommunikation liefert SDP die Medieninformationen mit den unterstützten Codecs, sowie Kontaktinformationen wie die IP-Adresse und Portnummer. (vgl. Trick und Weber, 2007)

Wie SIP basiert SDP auf einem UTF8-Zeichensatz, besteht aus Paaren von Schlüsselwörtern mit jeweils einem Wert, getrennt durch ein „=“ (vgl. Listing 2.1). Die Schlüsselwörter sind mit einem einzelnen Kleinbuchstaben codiert.

2.2.1.2 RTP

RTP wird für die meisten Multimedia-Anwendungen zum Streaming der Daten verwendet. Darunter zählen SIP und auch Alternativen wie H.323. RTP verwendet UDP und ist sehr einfach gehalten. Es bietet keine Kontrolle durch einen „Handshake“ und sorgt damit für einen verbindungslosen und ungesicherten Ende-zu-Ende-Transport von Echtzeitnutzdaten. Damit ist RTP, wie auch UDP, der Schicht 4 im OSI-Modell zuzuordnen.

Für den Transport der Echtzeitnutzdaten, zum Beispiel Audiostreams, bietet RTP eine Nummerierung der Pakete sowie einen Zeitstempel im Paketheader. Dadurch können die Daten in richtiger Reihenfolge und in korrekter zeitlicher Abfolge wiedergegeben werden. Allerdings besitzt RTP keine Merkmale bezüglich der Dienstgüte der Übertragung (vgl. Kapitel 2.2.3.1). (vgl. Trick und Weber, 2007)

Header

Der RTP-Header ist binär codiert und genau 12 Byte groß. Daran können bis zu 16 optionale Contributing Source (CSRC)-Identifikationen angefügt werden. Die CSRC-Felder sind bei einer Punkt-zu-Punkt-Kommunikation nicht enthalten. Sie werden hinzugefügt, wenn mehrere Quellen gemeinsam verwendet werden, zum Beispiel bei einer Telefonkonferenz. In diesem Fall werden die einzelnen Quellen über die CSRC-Felder identifiziert.

In jedem RTP-Header sind Informationen zum Ziel, zum Datentyp (z. B. der verwendete Codec), die Sequenznummer und die Synchronization Source (SSRC) enthalten. Mit dem zu Beginn der RTP-Übertragung zufällig gewählten und 32 Bit langen SSRC-Wert wird der Sender identifiziert.

RTCP

Zusammen mit RTP wird häufig das eigenständige Protokoll Real-time Transport Control Protocol (RTCP) verwendet, das der Kontrolle von RTP-Streams dient. Es ist jedoch nicht nicht unabhängig und wird nur zusammen mit RTP verwendet. RTCP baut wie RTP auf UDP auf und nutzt den nächst höheren ungradzahligen Port über dem verwendeten RTP-Port. Dabei ist RTCP optional und muss während einer RTP-Übertragung nicht verwendet, betrachtet oder ausgewertet werden.

Per RTCP ist es möglich, QoS-Informationen mit der Gegenstelle auszutauschen (vgl. Kapitel 2.2.3.1). Dabei wird periodisch ein „Sender Report“ verschickt, der der Gegenstelle mitteilt, welche Pakete versendet wurden. Diese erwidert den „Sender Report“ mit einem „Reciever Report“ und informiert über die empfangenen Pakete. Allerdings erfolgt aufgrund der Echtzeitnutzdaten kein wiederholtes Versenden von verlorengegangenen Paketen. RTCP dient nur der Überwachung sowie der Statistik und kann standardkonform im profilspezifischem Teil um eigene Messdaten erweitert werden.

Außerdem wird RTCP auch im geringen Umfang zur Sitzungssteuerung verwendet. So meldet sich ein neuer Client, zum Beispiel in einer SIP-Konferenz, mit einer speziellen RTCP-Nachricht mit seinem Namen an oder ab. (vgl. Trick und Weber, 2007)

2.2.1.3 Weitere alternative Protokolle

Neben SIP gibt es noch weitere Protokolle zur Signalisierung in VoIP-Verbindungen. Die meisten verwenden dabei ebenfalls RTP zur Übertragung der Echtzeitnutzdaten.

H.323

H.323 war lange Zeit das verbreitetste VoIP-Protokoll, wurde mittlerweile aber durch SIP verdrängt. Dabei ist H.323 mehr als nur ein einzelnes Protokoll. Vielmehr ist es die Beschreibung einer Architektur und eine Protokollsammlung. Es ist, im Gegensatz zu modularen SIP-Netzen, monolithisch aufgebaut und insgesamt viel mehr dem klassischen Telefonnetz nachempfunden. Dadurch ist es schwerer zu erweitern und in vielen

Belangen komplexer. Zudem ist es vollständig binär codiert. (vgl. Trick und Weber, 2007; Siegmund, 2002; Badach, 2007)

Skype

Mit Skype gibt es noch eine weitere vollständig properitäre VoIP-Lösung. Sie entstand aus der Peer-to-Peer Filesharing-Architektur KaZaA und basiert daher auch auf dem Peer-to-Peer-Ansatz. Eine Integration in ein öffentliches NGN ist durch die Architektur sowie die properitären Protokolle nicht möglich. Damit ist Skype ausschließlich auf das Internet beschränkt. (vgl. Trick und Weber, 2007)

2.2.2 VoIP-Netzelemente

In einem SIP-VoIP-Netz werden verschiedene Netzelemente unterschieden und diese im Folgenden vorgestellt. Dabei kann zwischen essentiellen Elementen, wie zum Beispiel User-Agents und Gateways, und optionalen, nicht in allen VoIP-Netzen eingesetzten Geräten, wie zum Beispiel einem Session-Border-Controller unterschieden werden.

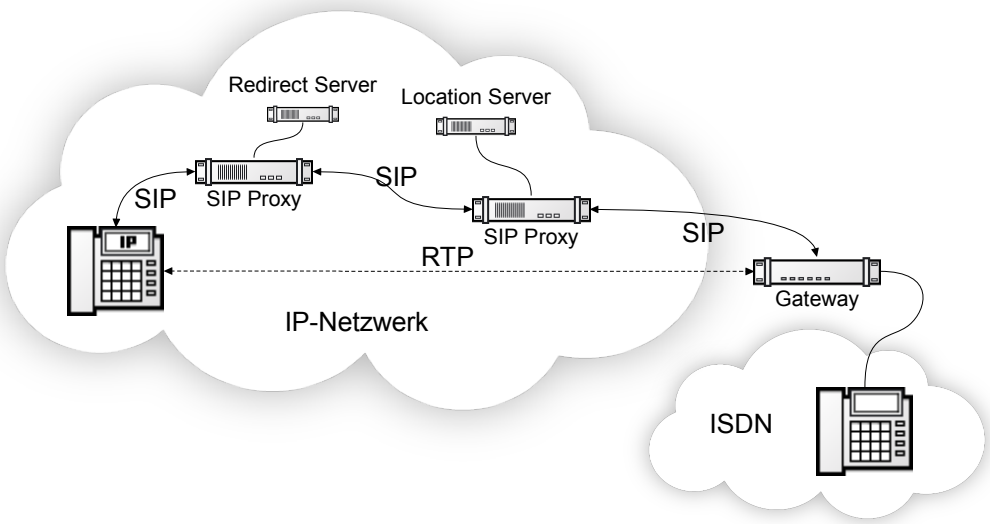


Abbildung 2.3: VoIP-Netzelemente in einem SIP-Netzwerk (vgl. Siegmund, 2002)

2.2.2.1 User-Agent

Als User-Agent bezeichnet man die Endgeräte in der VoIP-Kommunikation. Endgeräte können PC-Softphones, also Software die auf einem PC läuft, aber auch dedizierte IP-Telefone sein(vgl. Abbildung 2.4). Letztere gleichen äußerlich und von der Bedienung stark analogen oder ISDN-Telefonen. Die Funktion aber

entspricht derer eines Softphones. User-Agents können Anrufe über die direkte Adressierung aufbauen oder sich bei einem Registrar-Server registrieren und darüber die Kommunikation abwickeln. (vgl. Trick und Weber, 2007)

(a) Softphone Twinkle (b) SIP-Telefon Cisco 7960G

Abbildung 2.4: Verschiedene User-Agents

2.2.2.2 Registrar-Server

An einem Registrar-Server können sich User-Agents unabhängig von ihrer IP-Adresse registrieren und sind damit für andere Teilnehmer erreichbar. Der User-Agent meldet sich mit der SIP-Anfrage `REQUEST` beim Registrar-Server an. Danach ist er unter der Adresse `sip:<user>@<registrar-domain>` erreichbar. Es können sich dabei mehrere User-Agents auch gleichzeitig für den selben Account anmelden. (vgl. Trick und Weber, 2007)

2.2.2.3 SIP-Proxy

Ein SIP-Proxy verteilt SIP-Nachrichten. Dabei gibt es „stateless“ und „statefull“ SIP-Proxy. Ein „stateless“-Proxy ist ein passives SIP-Netzelement. Er versendet keine eigenen SIP-Nachrichten, sondern leitet ausschließlich empfangene Nachrichten weiter. Ein „statefull“ SIP-Proxy agiert als ein aktives Netzelement auf der SIP-Transaktionsebene. Der „statefull“-Proxy kann eigenständig auf Anfragen antworten und diese wiederholen. So beantwortet er zum Beispiel ein `INVITE` bereits mit einem `TRYING`, bevor er die Anfrage weiterleitet.

Die Informationen zum Routing kann ein SIP-Proxy unter anderem von einem Location-Server erhalten (vgl. Kapitel 2.2.2.5). In einfachen SIP-Netzwerken werden Registrar-Server und SIP-Proxy häufig zu einem einzelnen Server zusammengefasst. (vgl. Trick und Weber, 2007)

2.2.2.4 Gateway

Ein vollständig abgeschlossenes VoIP-Netzwerk bietet den Anwendern nur wenige Möglichkeiten mit anderen zu kommunizieren. Daher ist es wichtig, das VoIP-Netz mit anderen Netzwerken zu verbinden und die Daten zu konvertieren. Gateways verbinden zum Beispiel ein VoIP-Netzwerk mit einem PSTN, in dem ISDN-Geräte eingesetzt werden. Für diesen Zweck gibt es Media- und Signalisierungs-Gateways, sowie für jedes Netz *einen* steuernden Media Gateway Controller (MGC). Gateways arbeiten in der Regel bidirektional, sehen also eine Umsetzung der Signale und Media-Daten in beide Richtungen vor.

Es besteht die Möglichkeit, mehrere Gateways in einem Netzwerk zu betreiben. Diese werden dann von einem einzelnen MGC gesteuert, der die Verbindungen verwaltet. In der Literatur wird der MGC häufig auch Call-Agent, Call-Server oder Softswitch genannt. Es ist zudem möglich Media- und Signalisierungs-Gateways zu einem Gerät zu kombinieren, welches dann häufig als Residential-Gateway beszeichnet wird. (vgl. Trick und Weber, 2007; Siegmund, 2002)

2.2.2.5 Weitere SIP-Netzelemente

Neben den bereits beschriebenen Elementen, die in den meisten SIP Netzwerken vorhanden sind, gibt es noch optionale Elemente, die vor allem in größeren VoIP-Netzen zum Einsatz kommen. Diese sollen in den nächsten Absätzen kurz vorgestellt werden.

Redirect-Server

Ein Redirect-Server kann einem User-Agent geänderte oder neue Adressen mitteilen. Er stellt ein logisches, also nicht zwingend separates Netzelement dar und kann unter anderem mit einem User-Agent gekoppelt werden. Außerdem kann er, zum Beispiel bei beschriebener Kopplung mit einem User-Agent, auch zur dezentralen Rufumleitung genutzt werden und damit einen SIP-Proxy entlasten. SIP-Anfragen beantwortet er mit dem Grundtyp 3XX (Redirection). (vgl. Trick und Weber, 2007)

Location-Server

Im Gegensatz zu den bisherigen Netzelementen verarbeitet ein Location-Server keine SIP-Nachrichten, sondern ist direkt an einen SIP-Proxy oder Redirect-Server angeschlossen. Er arbeitet als eine Datenbank im SIP-Netzwerk und liefert die Kontaktinformationen der verschiedenen User-Agents aus. Die Aufgabe ist vergleichbar mit einem Domain-Name-Server im Internet. Die Informationen erhält der Location-Server von einem Registrar-Server und ist in einfachen Netzwerken häufig in den Registrar-Server, bzw. einer Kombination von SIP-Proxy und Registrar-Server, integriert.

Ein Beispiel für die Arbeitsweise eines Location-Servers ist die Auflösung einer Adresse wie `sip:<user>@<sipdomain>` zu einer direkt adressierbaren wie `sip:<user>@<IP>`. (vgl. Trick und Weber, 2007)

Presence-Server

Ein Presence-Server kann den Zustand der Erreichbarkeit von User-Agents zentral speichern und verwalten. So sendet ein User-Agent seinen Zustand (z. B. „abwesend“ oder „erreichbar“) an den Presence-Server, der wiederum den Zustand selbständig an alle Abonnenten veröffentlichen kann. Damit ist eine sogenannte „buddy list“ möglich, vergleichbar mit denen in Instant-Messaging-Systemen. (vgl. Trick und Weber, 2007)

Session-Border-Controller

Ein Session-Border-Controller stellt alle Funktionen am Übergang von zwei VoIP-Netzwerken bereit. Diese können auch problemlos unterschiedliche Protokolle benutzen, zum Beispiel SIP und H.323. Dabei kann der Session-Border-Controller Sicherheitsfunktionen bereitstellen, wie zum Beispiel Schutz vor Denial of Service (DoS), aber auch die Verschlüsselung der Signalisierung. Des Weiteren kann der Session-Border-Controller QoS-Funktionen anbieten, wie zum Beispiel ein „traffic shaping“ oder die Erfüllung eines Verkehrskontrakts. Auch die Unterstützung für das gesetzliche Abhören von Telefongesprächen ist in einem Session-Border-Controller möglich. (vgl. Trick und Weber, 2007)

2.2.3 VoIP-Qualität

In einem NGN müssen die verschiedenen Services unterstützt werden, welche sehr unterschiedliche Anforderungen an Bandbreite, Reaktionzeit und Verbindungsorientierung haben. Die bisherigen, herkömmlichen Netze waren immer auf einen bestimmten Service, zum Beispiel Daten *oder* Sprache, ausgelegt und konnten daraufhin leicht optimiert werden. Zudem gibt es in diesen Netzen jahrzehntelange Erfahrung bezüglich möglicher Probleme und deren Lösung.

In klassischen, leitungsvermittelten PSTN wird die Qualität der Verbindungen automatisch sichergestellt, da nur Geräte mit Zulassung verwendet werden, die getestet sind und sich so immer korrekt verhalten. Außerdem sind die Anforderungen an eine Verbindung zum Beispiel bezüglich der Bandbreite und Verzögerung klar definiert. Ist die Bandbreite nicht mehr vorhanden, wird der Verbindungsaufbau abgelehnt, da alle verfügbaren Kanäle vergeben sind. Die Kanäle werden die komplette Verbindung über reserviert. Dadurch kann keine Überlast während einer laufenden Verbindung auftreten. (vgl. Siegmund, 2002)

2.2.3.1 Quality of Service

In den NGN, vornehmlich für paketvermittelte Kommunikation ausgelegten IP-Netzen, treffen die oben beschriebenen Eigenschaften nicht zu. Dort gibt es für die Endgeräte keine Informationen über die vorhandene oder eine zugesicherte Bandbreite. Darüber hinaus können die teilnehmenden Endgeräte nicht, bzw. nur unzulänglich durch einen Netzbetreiber reglementiert werden. Andere Medien wie der klassische Datenverkehr, können zudem die VoIP-Verbindungen stören. Daher werden besondere Anforderungen an die IP-Netze

gestellt, die vor allem die Übermittlungszeit (Delay), den Jitter[4] sowie das Verhindern von Paketverlusten betreffen.

QoS ist kein einzelnes definiertes Verfahren, sondern ein Überbegriff für verschiedene Verfahren zur Dimensionierung des Netzwerks, der Reservierung von Bandbreite und Priorisierung sowie Routing von einzelnen Paketen in IP-Netzen. (vgl. Wallingford, 2005)

Im Folgenden werden die einzelnen Verfahren wie die „Überdimensionierung des Netzwerks“ und „Traffic Engineering“, spezielle Erweiterungen wie „IntServ“, „DiffServ“ und „MPLS“ sowie herstellerspezifische Verfahren genauer erläutert.

Überdimensionierung des Netzwerks

Die einfachste Möglichkeit QoS in einem Netzwerk zu unterstützten besteht in der Überdimensionierung der Netzwerk-Ressourcen. Das bedeutet, dass die Netzwerk-Ressourcen so ausgelegt werden, dass nur maximal 50 Prozent der verfügbaren Bandbreite genutzt werden. Dadurch werden gleichbleibende minimale Verzögerungen sichergestellt.

Diese Lösung ist technisch relativ einfach und auch effektiv, aber vor allem in größeren Netzwerken sehr teuer. Zudem muss nicht nur der VoIP-Verkehr beachtet werden, sondern auch Dienste, die nach dem Best-Effort-Prinzip arbeiten in die Planungen einbezogen werden. Die Überdimensionierung bietet ausschließlich eine Lösung für Netzwerke, in denen man auf alle Netzwerk-Segmente Zugriff hat. Sie kann damit nicht in gemischten Netzen verwendet werden, wie zum Beispiel dem Internet. (vgl. Trick und Weber, 2007)

Traffic Engineering

Vereinzelte Engpässe in einem Netzwerk, und damit Paketverluste und Verzögerungen, können mit Hilfe von Traffic Engineering durch eine möglichst gleichmäßige Netzauslastung vermieden werden. Das kann durch „constraint-based“-Routing erreicht werden. Hierbei wird nicht der kürzeste Pfad mit möglichst wenigen Hops[5] verwendet, sondern der Pfad mit der geringsten Auslastung, bzw. mit noch genügend Kapazität. (vgl. Trick und Weber, 2007)

IntServ

Bei dem IntServ-Verfahren werden einzelne Pakete nicht abhängig von anderen Paketen priorisiert, stattdessen erhalten sie eine absolute Dienstgüte. IntServ unterstützt damit absolutes QoS und emuliert das Verhalten von PSTN- bzw. ISDN-Netzwerken. Dabei werden bei IntServ drei verschiedene Dienstklassen unterstützt:

guaranteed service Diese Klasse ist für Echtzeitanwendungen wie VoIP und IPTV gedacht. Sie bietet eine zugesicherte Übertragungsrate, maximale Verzögerung und es dürfen keine Pakete verloren gehen.

[4] Die Schwankung der Übermittlungszeiten von IP-Paketen.

[5] Mit Hops werden in Netzwerken die Wege von einem Knoten zum anderen bezeichnet.

controlled load service Ohne Last im Netzwerk verhält sich diese Klasse wie „best effort“. Sie ist gedacht für Anwendungen ohne besondere Anforderung an Verzögerung und Verlustrate.

best effort Diese Klasse ist identisch mit dem Standard in IP-Netzwerken. Alle Pakete werden so schnell wie möglich weitergeleitet. Allerdings besteht keine Garantie gegen Verlust von Paketen.

Um die erforderlichen Ressourcen im Netzwerk zu reservieren, wird Resource Reservation Protocol (RSVP) benutzt, das Ende-zu-Ende den Bedarf und das Angebot signalisiert. Diese Reservierung muss allerdings immer wieder erneuert werden, da das IP-Netzwerk verbindungslos arbeitet. Zudem müssen alle Endgeräte und alle beteiligten Router RSVP unterstützten. Zudem ist eine zentrale Instanz im Netzwerk erforderlich, die Reservierungen auf Zulässigkeit prüft.

Durch die Signalisierung für RSVP kann es aber zu großer Last im Netzwerk kommen. In großen IP-Netzen ist es nur an den Rändern (Edge Router) und nicht im Kern (Core Router) handhabbar, da im Core sonst gleichzeitig eine zu große Zahl von Verbindungen gleichzeitig bearbeitet werden müssten.

In einem IPv4-Netzwerk müssen für IntServ mehrere Felder im IP-Paketheader ausgewertet werden. IPv6 bietet hier Abhilfe mit einem speziellen „flow label“ im Paketheader. (vgl. Trick und Weber, 2007)

DiffServ

Im genauen Gegensatz zu IntServ unterstützt DiffServ ausschließlich relatives QoS und IP-Pakete werden nur relativ zu anderen Paketen priorisiert. Aber es gibt ebenfalls verschiedene Dienstklassen, die die Priorität der Pakete bestimmen. Die insgesamt 32 Dienstklassen können zu drei Hauptklassen zusammengefasst werden:

expedited forwarding Die IP Pakete werden beschleunigt und vorrangig versendet.

assured forwarding Die Pakete werden sicher versendet und auch bei hoher Last nicht verworfen.

best effort Alle Pakete werden gleichberechtigt behandelt.

Durch die relative Priorisierung erhält man für die Dienstgüte keine absolute, sondern nur eine relative Garantie. Man spricht daher auch von Class of Service (CoS) anstatt von QoS.

Die Klassifizierung der Pakete erfolgt bei DiffServ vollständig verbindungslos anhand des IPv4-Type of Service (ToS)-Feldes, bzw. bei IPv6 durch das „traffic class“-Feld. Durch die fehlende Signalisierung ist DiffServ relativ einfach umzusetzen. (vgl. Trick und Weber, 2007; Siegmund, 2002)

Multiprotocol Label Switching

Generell ist in einem Netzwerk Switching schneller als Routing. Deswegen ersetzt Multiprotocol Label Switching (MPLS) das Routing in Schicht drei des OSI-Modells durch Switching in der zweiten Schicht und senkt damit die Last im Netzwerk.

Beim Eintritt eines IP-Paketes wird diesem ein Label zugewiesen, wobei zusammengehörige Pakete das gleiche Label erhalten. Dadurch müssen nicht mehr die vollständigen IP-Header für das Routing ausgewertet werden, sondern lediglich die vergebenen Labels. Alle Pakete mit dem gleichen Label werden auf dem

gleichen Weg durch das Netz geführt. Dadurch sind Vertauschungen und sehr stark differenzierende Paketlaufzeiten unwahrscheinlicher.

Allerdings ist nur durch zusätzliche Protokolle wie Resource Reservation Protocol-Traffic Engineering (RSVP-TE) und Label Distribution Protocol (LDP), mit deren Hilfe beim Aufbau des Pfades Ressourcen reserviert werden, QoS möglich. (vgl. Trick und Weber, 2007)

Herstellerspezifische Verfahren

Neben den hier vorgestellten Konzepten für QoS haben noch verschiedene Hersteller Verfahren entwickelt, um Pakete priorisieren zu können. So ist es auf Geräten von Cisco möglich, RTP-Pakete strikt zu priorisieren. Dabei werden alle UDP-Pakete, die auf einen ungeraden Port im Bereich 16384-32768 gesendet werden bevorzugt behandelt. Ein anderes Beispiel ist „dynamic access“ von 3com. Dabei werden Pakete anhand der Portnummer und des IPv4-ToS-Feldes priorisiert. (vgl. Detken, 2002)

Fazit

Die hier vorgestellten Verfahren sind vor allem in einer Kombination wirkungsvoll, da sie sich zum Teil sehr gut ergänzen. Durch die Bearbeitung der Pakete wird allerdings eine zusätzliche Verzögerung erzeugt, welches sich wiederum negativ auf die Qualität auswirken kann. Trotzdem ist es möglich, durch QoS viele Probleme zumindest zu mildern, allerdings können die Probleme nur im eigenen, selbst kontrollierten Netzwerk beseitigt werden. (vgl. Detken, 2002; Badach, 2007; Siegmund, 2002)

2.2.3.2 Quality of Experience

Wie schon in Kapitel 2.2.3.1 festgestellt, wird die Qualität von VoIP vor allem von der Laufzeit, den Laufzeitunterschieden sowie Paketverlusten beeinflusst. Diese QoS-Parameter sind relativ leicht zu messen, sagen aber nur wenig über die eigentliche Sprachqualität des Telefongesprächs aus. Für die Beschreibung der Quality of Experience (QoE) werden die im Folgenden vorgestellten Verfahren Mean Opinion Score (MOS), Perceptual Evaluation of Speech Quality (PESQ) und R-Factor verwendet.

Mean Opinion Score

Die Sprachqualität lässt sich durch folgende Aspekte beschreiben:

- die Verständlichkeit der Sprache
- die Akzeptanz der Lautstärke
- die Akzeptanz der Laufzeitunterschiede und Echos.

Um die subjektiv empfundene Sprachqualität objektiv bewerten zu können, werden die Auswirkungen dieser Aspekte durch den MOS eingeteilt und bewertet. Diese MOS-Werte (vgl. Tabelle 2.1) wurden durch einen standardisierten Testaufbau mit ausgewählten Versuchspersonen ermittelt. Dabei entspricht eine analoge Sprachübertragung einem MOS-Wert von 3,5 bis 4,0, über ISDN sind Werte von 4,5 zu erzielen. VoIP bietet je nach verwendeter Sprachcodierung eine Qualität zwischen 3,8 und 4,4. Damit kann VoIP fast die Qualität einer ISDN-Übertragung erreichen. (vgl. Badach, 2007)

MOS-Wert	**Bedeutung**
5 = excellent	keinerlei Anstrengung zum Verständnis der Sprache notwendig; totale Entspannung möglich
4 = good	keine Anstrengung notwendig, Aufmerksamkeit nötig
3 = fair	leichte, moderate Anstrengung nötig
2 = poor	merkbare, deutliche Anstrengung nötig
1 = bad	trotz Anstrengung keine Verständigung

Tabelle 2.1: MOS-Skala (vgl. Badach, 2007)

Paketverluste haben einen direkten und negativen Einfluss auf die Qualität einer Verbindung und damit auch auf den MOS-Wert. So ist trotz eines guten MOS-Wertes von 4,4 bei G.711, in einem Netzwerk mit 4 Prozent Paketverlust ein MOS-Wert von 3 nicht mehr zu erreichen. (vgl. Wallingford, 2005)

Perceptual Evaluation of Speech Quality

Durch die Abhängigkeit von Versuchspersonen ist es allerdings nicht möglich, automatisiert MOS-Werte zu ermitteln. Daher wurde PESQ von der ITU Telecommunication Standardization Sector[6] (ITU-T) entworfen und als Standard P.862 verabschiedet. Mit PESQ steht ein objektives und automatisches Analyseverfahren zur Bewertung der Sprachqualität in VoIP-Umgebungen zur Verfügung. Dafür wird das Ausgangssignal mit dem Empfangssignal verglichen und anhand von mathematischen Modellen bewertet. Dabei werden nur die Unterschiede der beiden Signale untersucht und zusammengefasst. Der erhaltene Wert ist direkt übertragbar in einen MOS-Wert. Das Maximum liegt jedoch bei PESQ bei 4,5, da statistisch gesehen auch bei MOS im Durchschnitt maximal eine Bewertung von 4,5 erreicht werden kann. (vgl. Wallingford, 2005)

R-Factor

Neben MOS und PESQ kann außerdem der R-Factor für eine QoE-Analyse berechnet werden. Dieser ist ebenfalls direkt in einen MOS-Wert umrechenbar um ihn vergleichbar einordnen zu können. Ermittelt wird der R-Factor aus dem aufgetretenen Jitter, der Latenz, dem Packetverlust. Er ist in der ITU-T Recommendation G.107 beschrieben. Ein *typischer* R-Factor-Wert liegt zwischen 50 und 90, theoretisch befindet er sich aber in der Spanne von 0 bis 94. Mit der Hilfe des R-Factors ist es möglich, QoE aus einem Laborumfeld in ein produktiv genutztes Netzwerk zu überführen.

[6] Die ITU-T erarbeitet internationale technische Standards und Normen.

2.2.3.3 Quality of Analysis und Quality of Diagnosis

Durch VoIPFuture, einem in Hamburg ansässigen Unternehmen für VoIP-Qualitätsmessung, wurden die Begriffe Quality of Analysis (QoA) und Quality of Diagnosis (QoD) definiert. Dabei handelt es sich um eine genaue Analyse des VoIP-Netzwerkverkehrs und der Feststellung der konkreten Ursachen bei aufgetretenen Fehlern. Es kann sich zum Beispiel um „Network Buffering“ oder „Sender Synchronization“ handeln. Diese Analyse ist wesentlich genauer und in höherer Auflösung, als es beim R-Factor der Fall ist. Aus der Analyse wird eine Diagnose erzeugt, die präzise Informationen über konkrete Ursachen der gefundenen Probleme liefert.

Bei QoA werden nicht nur Jitter und Paketverluste untersucht, sondern jedes einzelne RTP-Paket. Es wird jedoch nur der RTP-Header analysiert, wodurch verschlüsselte RTP-Streams (SRTP) untersucht werden können. Durch die Untersuchung aller RTP-Pakete können nicht nur Testverbindungen analysiert werden, sondern auch reale Verbindungen in Echtzeit. Zudem können Fehler in der Implementierung des RTP-Stacks in VoIP-Geräten gefunden werden.

Die Lösung von VoIPFuture kann in zwei Schichten aufgeteilt werden, der Analyseschicht und der Diagnoseschicht. In der Analyseschicht ist es möglich, Indikatoren zu bestimmen. Dabei kann es sich zum Beispiel um „Consecutive Packet Loss“ handeln, mehrere aufeinander folgende verlorengegange Pakete. Oder auch um „Jitter Critical Level“, das einen Hinweis darauf gibt, ob der Jitter Buffer nicht mehr in der Lage ist, den Jitter auszugleichen.

In der Diagnoseschicht werden aus der Analyse Indikatoren über konkrete Ursachen erzeugt. Dabei kann es sich um „Network Buffering"handeln, eine Überlast des Netzwerks. Ein weiterer Indikator kann „Sender Timing“ sein, welches darauf hindeutet, dass der Sender nicht die zum Codec passende Sample Rate verwendet. (vgl. White Paper, 2008)

2.2.3.4 Parameter zur Analyse und Diagnose

Zur Analyse und damit auch zur Diagnose werden nur die Isochronität, die Vollständigkeit sowie die Chronologie geprüft. Daraus abgeleitete Parameter werden nicht verwendet, wie zum Beispiel Jitter, die aus der Interarrival Time ermittelt werden.

Isochronität (Interarrival Time)

Die Audiodaten werden *kontinuierlich* beim Sender abgetastet, in RTP-Pakete verpackt und versendet. Die Pakete müssen auch kontinuierlich und mit dem immer gleichen und ïdealen“ Abstand, das heißt isochron, beim Empfänger eintreffen. Der ïdeale“ Abstand wird aus der Sample-Rate des Codecs spezifiziert. Im Gegensatz zum Jitter, der nur die Abweichung vom Sollzustand des Abstands beschreibt, bietet sich mit QoA auch die Möglichkeit festzustellen, ob der erzielte Wert *über* oder *unter* dem Sollzustand liegt. Nur dadurch lässt sich in der Diagnose zuverlässig auf Fehlerquellen schließen. Der Wert, die Interarrival Time, wird aus der Differenz der Empfangszeit zweier aufeinander folgender Pakete berechnet (vgl. Abbildung 2.5). Da sich der Jitter aus der Interarrival Time ableiten lässt, wird dieser nicht gesondert untersucht.

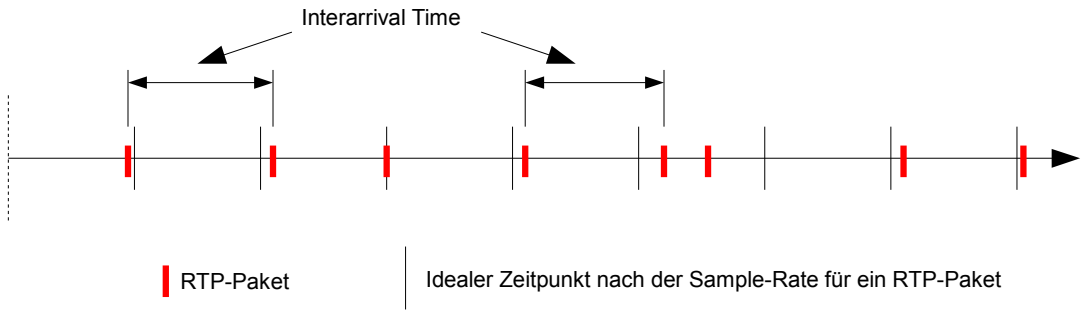


Abbildung 2.5: Interarrival Time von RTP-Paketen auf einer Zeitachse

Zur Ermittlung des Abstands kann nicht der Zeitstempel aus den RTP-Paketen verwendet werden, da dieser vom Sender bei der Erzeugung der Pakete in aller Regel korrekt gesetzt wird und sich damit keine Netzwerkprobleme aufdecken lassen. Nur der Zeitpunkt des Eintreffens des Pakets ist aussagekräftig für die Analyse, da nur dieser alle Probleme, von der Erzeugung bis zur Übertragung, enthält. (vgl. White Paper, 2008)

Vollständigkeit (Packet Loss)

Absolute oder prozentuale Angaben über den Packetverlust liefern keine Information darüber, ob vereinzelt Pakete verloren gingen, oder ob der Verlust direkt hintereinander geschehen ist. Im ersten Fall kann der Paketverlust häufig durch den verwendeten Codec kompensiert werden. Im zweiten Fall kann ein Codec den Verlust nicht mehr ausgleichen und es kommt zu hörbaren Qualitätseinbußen.

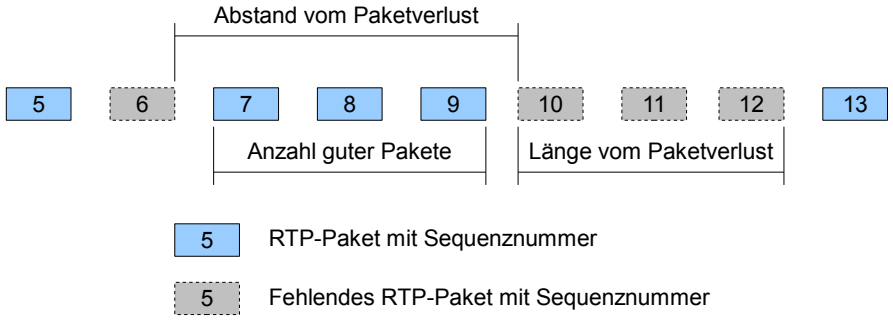


Abbildung 2.6: Vollständigkeit der RTP-Pakete

Bei der Analyse wird nicht nur die Anzahl der verlorenen Pakete hintereinander registriert, sondern auch die Anzahl der guten, also empfangenen Pakete, zwischen zwei oder mehr verlorenen Paketen (vgl. Abbildung 2.6). Dadurch lässt sich nicht nur die Anzahl, sondern auch der Abstand der Paketverluste untersuchen. Dabei gilt, dass Paketverluste im großen Abstand für den Anwender als nicht so störend empfunden werden wie viele kleinere, aber ständig auftretende, Paketverluste. Durch die Analyse von beiden, den verlorenen

als auch den empfangenen Paketen, bietet sich die Möglichkeit zur späteren Diagnose. Technisch basiert die Erkennung der Verluste auf der RTP-Sequenznummer im Header der Pakete. (vgl. White Paper, 2008)

Chronologie (Packet Order)

Die Chronologie basiert wie die Vollständigkeit technisch auf der Sequenznummer der RTP-Pakete. Ideal ist hier ein streng monoton steigender Wert. Steigt der Wert nicht streng monoton, kann die Ursache eine Vertauschung zweier Pakete sein. Zum Beispiel springt zunächst die Sequenznummer von 5 auf 7 vor und nach dem Paket mit der Sequenznummer 9 wird das Paket mit der Sequenznummer 6 empfangen. Hier springt die Sequenznummer zurück, um dann sofort wieder vor zu springen (vgl. Abbildung 2.7).

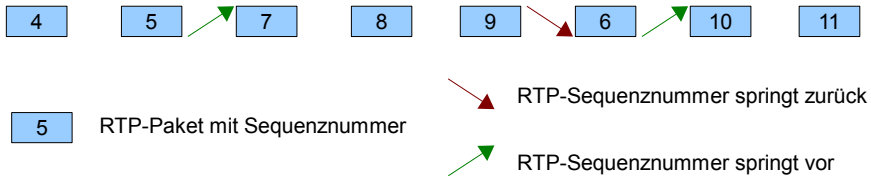


Abbildung 2.7: Chronologie der RTP-Pakete

In der Praxis kann es aber, zum Beispiel durch fehlerhafte Implementierungen, auch vorkommen, dass die Sequenznummer zurückspringt *ohne* dass eine Vertauschung von Paketen vorliegt. In beiden Fällen kann die fehlende Chronologie zur Erkennung einer Störung dienen. (vgl. White Paper, 2008)

2.2.4 Kritischer Vergleich von QoS/QoE und QoA/QoD

Es ist möglich, dass durch QoA und QoD gefundene Fehler vom Anwender nicht bemerkt werden. Die Diagnose läuft ausschließlich auf der technischen Ebene ab und es werden keine Analysen der übertragenen Audiodaten erstellt. Daher kann eine schlecht bewertete Verbindung trotzdem von einem Menschen als gut oder zumindest zufriedenstellend empfunden werden.

Im Gegensatz dazu sind Messungen für MOS oder PESQ nicht während des normalen Betriebs möglich, da für MOS Testpersonen die Audiodaten in einem definierten Umfeld bewerten müssen. Bei PESQ ist es erforderlich, neben den empfangen Daten auch das Original für einen Vergleich zur Verfügung zu haben. Daher lässt sich PESQ nur zum zeitweiligen Test eines Netzwerks oder von Hardware verwenden. Der R-Factor dagegen kann zwar in einem regulären Betrieb für Messungen herangezogen werden, allerdings werden damit nur wenige und auch nur unpräzise Aussagen möglich. Ein weiterer Nachteil von MOS ist auch die Abhängigkeit vom verwendeten Codec. So kann ein stark komprimierender Codec wie zum Beispiel G.723.1 ACELP selbst in einem optimalen Umfeld und mit einer optimalen Übertragung nur einen MOS-Wert von 3,65 erreichen, obwohl der Stream keinen Fehler enthält.

Mit QoA und QoD kann dagegen kontinuierlich ein VoIP-Netzwerk während des normalen und laufenden Betriebs überwachen. Dadurch ist es möglich, schon frühzeitig kleine Fehler zu erkennen und zu beheben, bevor sie von Anwendern bemerkt oder als störend empfunden werden. Das ist auch möglich, da durch QoD nicht nur eine Bewertung erfolgt, sondern auch genaue Hinweise auf die Ursache der Fehler gegeben werden. Damit kann man neben Netzwerkfehlern auch Implementierungsfehler in Geräten erkennen.

2.2.5 Lösungen und Anwendungen von VoIPFuture

VoIPFuture ermöglicht Kunden ihren VoIP-Verkehr 24 Stunden, 7 Tage in der Woche in Echtzeit zu überwachen und liefert dank QoA und QoD neben Fehlern auch deren mögliche Ursachen. Dafür bietet VoIPFuture mehrere, auf viele Kundenszenarios zugeschnittene Lösungen an. Darüber hinaus verwendet VoIPFuture für interne Analysen zusätzliche Werkzeuge. Im Folgenden werden die Produkte und die internen Werkzeuge von VoIPFuture vorgestellt.

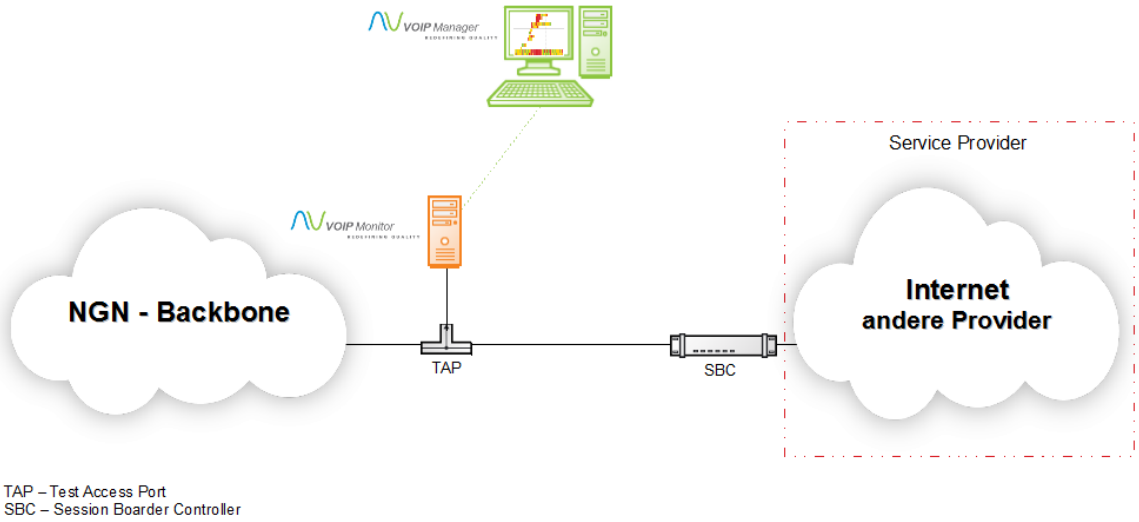


Abbildung 2.8: Szenario der VoIPFuture-Lösungen bei einem Carrier (Quelle: VoIPFuture)

2.2.5.1 Library

Die VoIPFuture Library analysiert mit Hilfe von QoA (vgl. Kapitel 2.2.3.3) die RTP-Pakete. Dabei werden von der Library regelmäßig Analysen angefordert und diese ununterbrochen mit allen neu eingetroffen RTP-Paketen versorgt. Die jeweils pro Abfrage für jeden RTP-Stream gelieferten Analysen werden als „Vektor“ bezeichnet.

Es wurde bei der Library vor allem auf Performanz und Portabilität geachtet. Aus diesem Grund ist sie vollständig in American National Standards Institute[7] (ANSI)-C geschrieben und enthält nur Abhängigkeiten zur Standard Library C (libc) sowie libmath. Dadurch kann die Library nicht nur von dem Monitor (vlg. Kapitel 2.2.5.2) verwendet werden, sondern auch in die Firmware von Telefonen und anderen VoIP-Geräten wie zum Beispiel einem Session Border Controller implementiert werden.

2.2.5.2 Monitor

Der VoIPFuture Monitor verwendet die VoIPFuture Library zur Analyse von RTP-Paketen und speichert die aus dieser Analyse erhaltenen Verktoren in *VoIPFuture Analytic and Diagnostic file format (VAD)*-Dateien. Zur Zeit sind problemlos mehrere tausend gleichzeitige Streams pro Monitor auf einer x86-basierten Standard-Server-Hardware möglich. Dabei kann der Monitor die angefallenen Daten komprimieren und mehrere fehlerfreie Vektoren zusammenfassen.

Der Monitor wird in den typischen Anwendungsszenarios mit Hilfe von einem Test Access Port[8] (TAP) oder dem „Monitor Port“ eines Carrier Grade Routers in das Netzwerk eingeschleift (vgl. Abbildung 2.8). Dabei ist es wichtig, dass der „Monitor Port“ des Routers die Laufzeiten, sowie die Informationen der Pakete, nicht verändert. Aufgrund dessen wird die Verwendung eines TAP von VoIPFuture empfohlen.

2.2.5.3 Manager

Der VoIPFuture Manager ist eine browserbasierte Java Enterprise Edition (J2EE)-Anwendung, die dem Anwender komfortablen Zugriff auf die von der Library und dem Monitor erzeugten Vektoren ermöglicht. Dabei bietet der Manager die Möglichkeit, ausgehend von einer Tageszusammenfassung, die den Überblick über tausende Streams zur Verfügung stellt, bis auf einzelne Streams und Vektoren zuzugreifen. Dabei werden detaillierte Auswertungen geboten, sowie anpassbare Reports für bestimmte Analysen.

2.2.5.4 Weitere Werkzeuge für die Analyse und Diagnose

Neben den bisher vorgestellten offiziellen Lösungen und Produkten, verwendet VoIPFuture die im Folgenden beschriebenen Werkzeuge für die interne Analyse und Diagnose von VoIP-Verbindungen.

[7] Das US-amerikanische Gegenstück zum Deutsche Institut für Normung e.V. (DIN). Eine Stelle zur Normierung von Verfahrensweisen und Spezifikationen.

[8] Ein Gerät, das einen zusätzlichen Netzwerkport in eine Netzwerkverbindung schleift, ähnlich einem T-Stück in einer BNC-Verkabelung

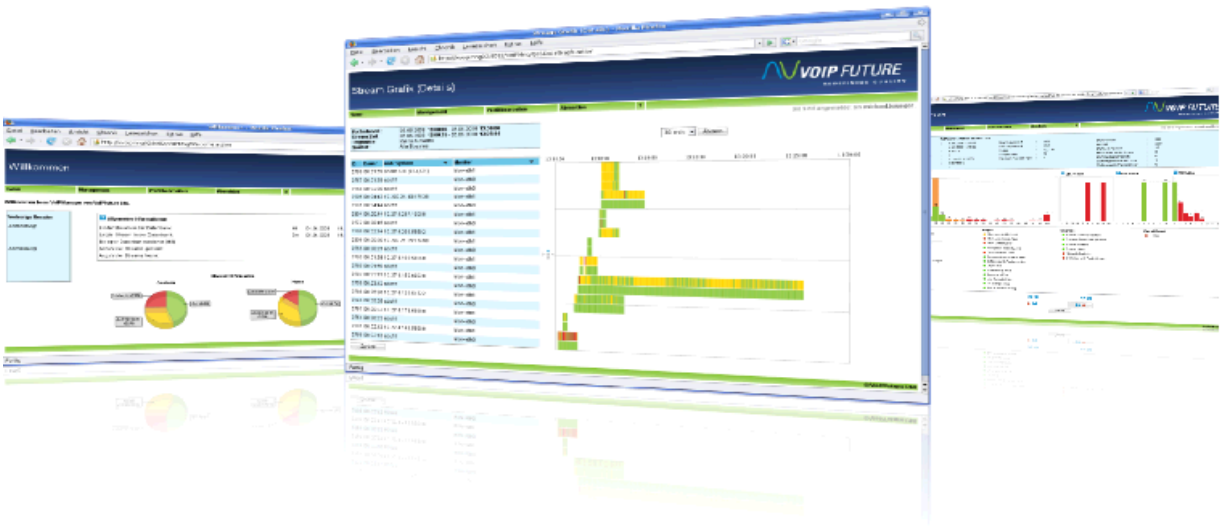


Abbildung 2.9: Ansichten des VoIPFuture Manager (Quelle: VoIPFuture)

cap2vad

Mit den freien Anwendungen tcpdump[9] und Wireshark[10] kann Netzwerk-Verkehr auf einer Netzwerkschnittstelle aufgezeichnet und in sogenannte „.cap“-Dateien gespeichert werden. Mit dem Werkzeug „cap2vad“ ist es dann möglich VAD-Dateien aus diesen „.cap“-Dateien zu erzeugen. Die VAD-Dateien können danach im VoIPFuture Manager oder im VectorViewer zur Analyse und Diagnose verwendet werden.

VectorViewer

Der VoIPFuture VectorViewer wertet VAD-Dateien aus und erzeugt aus den enthaltenen Vektoren Auswertungen ähnlich denen des Managers. Im Gegensatz zu diesem ist der VectorViewer allerdings nur für die Diagnose weniger Streams geeignet.

[9] `http://www.tcpdump.org` (aufgerufen am 9.12.2008)
[10] `http://www.wireshark.org` (aufgerufen am 9.12.2008)

2.3 Android

Abbildung 2.10: Android - Open Source Project (vgl. Google, 2008b)

Nach einigen Gerüchten über ein Telefon von Google, häufig in Anspielung auf das iPhone von Apple, auch gPhone genannt, stellte Google am 5. November 2007 Android vor. In diesem Kapitel wird diese Lösung genauer betrachtet und vorgestellt.

2.3.1 Einordnung

Android ist kein abgeschlossenes handfestes Produkt, wie viele erwartet hatten. Google versucht einen anderen Weg als Apple zu gehen. Mit Android stellt Google eine Software-Plattform vor und versammelt einige namhafte Firmen unter der OHA um sich. Darunter Hardware-Hersteller wie HTC, Samsung, Motorola und LG Electronics, aber auch Provider wie Sprint und T-Mobile.[11] Allerdings tritt zum Zeitpunkt dieser Arbeit fast ausschließlich Google als Quelle für Informationen oder neue Entwicklungen in Erscheinung.

Android ist nicht als abgeschlossene Einheit gedacht, sondern eine offene Plattform. Im Gegensatz zu den bisherigen Mobiltelefon-Plattformen, soll diese nicht nur durch einzelne Anwendungen erweiterbar sein. Unter Android sind grundsätzlich alle Applikationen austauschbar. So ist es möglich selbst den Desktop, auch „Home Screen“ genannt, durch einen eigenen zu ersetzen. Google legt dabei nur die Rahmenbedingen fest und stellt einzelne Rahmenapplikationen, wie zum Beispiel eine Telefonie-Anwendung und einen Webbrowser, zu Verfügung. Dabei legt Google großen Wert darauf, dass diese Anwendungen einfach und zuverlässig miteinander interagieren können.

Außerdem ist es Google wichtig, dass die Plattform nicht fragmentiert. Das heißt, es existiert nur eine Version einer Anwendung, die möglichst auf allen Installationen von Android laufen soll. Das soll auch die Arbeit der

[11] Unter `http://www.openhandsetalliance.com/oha_members.html` ist eine vollständige Liste aller Mitglieder zu finden. (aufgerufen am 15.9.2008)

Entwickler vereinfachen, die sich damit auf die Funktionen konzentrieren können. Ermöglicht wird das durch die Dalvik Virtual Machine[12] (VM) (vgl. Kapitel 3.2.3), in der plattformunabhängiger Bytecode ausgeführt wird. Dadurch sind auch die Anwendungen unabhängig von der verwendeten Hardware. (vgl. Brady, 2008)

2.3.2 Entwicklung für Android

Zum Konzept von Android gehört, dass eine Vielzahl von Erweiterungen und Anwendungen für die Android-Plattform entwickelt werden. Für diesen Zweck ist es wichtig viele Entwickler für Android zu begeistern.

2.3.2.1 Programmiersprache Java

Anwendungen für Android werden in der Programmiersprache Java entwickelt, die durch den Sun Java Compiler danach durch einen Bytecode-Konverter in das androidspezifische Dex-Format übersetzt wird (vgl. Kapitel 3.2.3). Dabei steht für die Programmierung in Java eine umfangreiche Android-Application Programming Interface[13] (API) zur Verfügung. Es umfasst weitestgehend den Umfang der Java Standard Edition[14] (J2SE), realisiert durch die Benutzung des Apache Harmony Frameworks[15]. Bestandteile wie zum Beispiel Swing, SWT und AWT fehlen vollständig und wurden zum Teil durch androidspezifische Alternativen ersetzt. Von Graphical User Interface (GUI)-Komponenten abgesehen, sind für eine Portierung von J2SE-Anwendungen auf die Android-Plattform oft nur kleine Anpassungen notwendig.

2.3.2.2 Literatur über Android

Zum Zeitpunkt dieser Arbeit gibt es keine veröffentlichten Bücher und auch sonst keine ausführlicheren Texte über Android. Lediglich vereinzelte Einführungen sind erhältlich, allerdings beschreiben diese veraltete Versionen des SDK. Als Entwickler ist man daher auf die Dokumentation von Google, sowie zahlreiche Blogs und Foren angewiesen. Vermutlich wird sich diese Situation ändern und bald auch umfangreiche Dokumentationen von verschiedenen Anbietern verfügbar sein. Bei vielen Problemen helfen zudem klassische Java-Bücher zu J2SE weiter.

[12] Eine virtuelle Laufzeitumgebung für Programme auf einem Host-System. Bekannte Beispiele sind die SUN Java Virtual Machine oder Smalltalk 80.

[13] Eine definierte Schnittstelle einer Bibliothek oder Anwendung, durch die sich diese auf Quellcode-Ebene von anderen Entwicklern verwenden lässt.

[14] Die in der Regel verwendete Java-Version von Sun. Sie wird für Anwendungen auf dem Desktop verwendet.

[15] `http://harmony.apache.org/` (aufgerufen am 29.9.2008)

2.3.2.3 Unterstützung durch Google

Google stellt auf YouTube Videos bereit, die Entwicklern den Einstieg erleichtern und für die Android-Plattform begeistern sollen[16]. Zudem wurde ein SDK veröffentlicht, der einen Emulator und einige Entwicklungswerkzeuge enthält (vgl. Kapitel 3.2.1). Darüber hinaus gibt es für Fragen der Entwickler verschiedene Google Groups[17], sowie einen IRC-Channel im freenode-IRC-Netz[18].

Außerdem stellt Google eine umfangreiche Dokumentation, Referenz und Beispiele unter `http://code.google.com/android/documentation.html` und im Android Development Kit (ADK) bereit.

2.3.2.4 Android Developer Challenge

Um möglichst früh und schnell viele Entwickler für die neue Plattform zu begeistern, schrieb Google einen mit 10 Millionen US-Dollar dotierten Programmierwettbewerb aus: die „Android Developer Challenge“. Die erste von zwei Runden in diesem Wettbewerb, genannt „Round 1“, endete am 14. April 2008 und es beteiligten sich 1.788 Projekte aus über 70 Ländern. Aus diesen wählte Google die 50 besten Projekte aus, die als Belohnung in den folgenden Monaten eine besondere Unterstützung von Google genießen durften. Sie erhielten zum Beispiel Zugriff auf nichtöffentliche Versionen des Android-SDK. Am 28. August 2008 wurden schließlich die Gewinner der „Round 2“ veröffentlicht (vgl. Abbildung 2.11)[19].

Nach der Fertigstellung der ersten Android-Geräte, voraussichtlich gegen Ende 2008, soll mit der so genannten „Challenge II“ der zweite Teil des Wettbewerbs starten. Dieser ist unabhängig von Teil eins.

2.3.3 Anwendungen und besondere Funktionen von Android

Für Android werden zur Zeit nur grundlegende Anwendungen von Google ausgeliefert. So ist neben einem einfachen Telefonbuch und einer Telefonie-Anwendung auch ein auf dem webkit-Framework[20] basierender Webbrowser enthalten. Darüber hinaus ist es seit SDK 0.9_r1 möglich, SMS sowie E-Mails zu empfangen und zu versenden.

Diese Lösungen sind allerdings nur als ein Beispiel zu sehen, da jeder Hersteller seine eigenen Entwicklungen einsetzen kann. Außerdem ist es durch das offene Konzept von Android für den Anwender möglich, die vorgegebenen Programme durch alternative Versionen zu ersetzen. Es besteht damit ein großer Unterschied zu vergleichbaren Plattformen, bei denen das in der Regel nicht oder nur sehr eingeschränkt möglich ist.

Android zeichnet sich unter anderem dadurch aus, dass verschiedene Anwendungen sehr gut miteinander agieren und kommunizieren können. Dabei werden Abhängigkeiten mit Hilfe von sogenannten „Intents“

[16] `http://www.youtube.com/watch?v=ENRcZcnoptM` (aufgerufen am 8.9.2008))
[17] `http://source.android.com/discuss` (aufgerufen am 8.9.2008)
[18] `http://freenode.net` (aufgerufen am 8.9.2008)
[19] Die Liste der Gewinner ist unter `http://code.google.com/android/adc_gallery/` zu finden. (aufgerufen am 8.9.2008)
[20] `http://webkit.org/` (aufgerufen am 9.9.2008)

(a) cab4me (b) Ecorio (c) Wertago

Abbildung 2.11: Drei der Hauptgewinner der „Android Developer Challenge I Round 2“

weitestgehend verhindert. Dabei definiert eine Anwendung alle „Intents“, die diese verarbeiten kann. Die aufrufende Anwendung referenziert nicht direkt die Ziel-Anwendung, sondern verwendet für den Aufruf einen „Intent“.

2.3.4 Hardware

Generell ist Android für keine spezielle Hardware konzipiert, sondern soll im Extremfall sogar ohne Display laufen. Allerdings gelten diese Vorgaben noch nicht für das aktuelle SDK 1.0r1. Das SDK wurde besonders in den letzten Monaten vor der endgültigen Veröffentlichung im Herbst 2008 auf das erste Android-Handy „HTC G1“ angepasst.

2.3.4.1 Anforderungen

Zur Zeit läuft Android auf einem ARMv5TE-Prozessor. Das ist keine generelle Einschränkung der Plattform, sondern bezieht sich nur auf die zur Zeit von Google zur Verfügung gestellten binären Dateien, wie zum Beispiel der Dalvik VM. Eine grundlegende Anforderung an die Hardware für Android ist die Lauffähigkeit eines Linux-Kernels. Da dieser aber bereits auf viele verschiedene Plattformen portiert wurde, gibt es eine große Auswahl von unterstützten Prozessor-Typen. Laut inoffiziellen Angaben von Google-Mitarbeitern würde eine Portierung auf andere Prozessor-Typen wie einem v4t-ARM-Prozessor nur eine bis maximal vier Wochen dauern. Es bedarf dafür hauptsächlich der Änderung von Compiler-Schaltern sowie den Anpassung von Assembler-Quellcode.

Ein „Low End Android Device“ sollte mindestens 64 MB Hauptspeicher besitzen. Nach dem Start der „low level“-Services stehen dann auf solch einem System noch 40 MB zur Verfügung. Weitere 20 MB werden danach für die „high level“-Services, wie zum Beispiel dem Phone-Service, benötigt. (vgl. Bornstein, 2008)

2.3.4.2 Android-Telefone

Während dieser Arbeit wurde am 23. September das erste Android-Telefon, das G1 von HTC (vgl. Abbildung 2.12) vorgestellt. Es is seit dem 22. Oktober in den USA erhältlich und wird seit November auch in Großbritannien verkauft. Das G1 verfügt neben einem großen Touchscreen über eine ausziehbare Tastatur, bis zu 8 GB Speicher auf microSD-Karten, sowie die übliche Ausstattung eines Smartphones wie WLAN, Bluetooth und UMTS-Unterstützung. Als Besonderheit gibt es neben dem GPS Modul und einem Lagesensor einen eingebauten Kompass.

Abbildung 2.12: Android auf dem HTC G1 (Quelle: `http://www.htc.com`)

Von Samsung wird für die Jahreswende 2008/2009 ebenfalls ein Modell angekündigt, allerdings gibt es darüber genauso wenig Informationen wie über zu erwartende Modelle von den OHA-Mitgliedern Motorola und LG Electronics.

2.3.4.3 Android auf bestehender Hardware

Außerdem gab es bereits kurz nach Veröffentlichung des ADK Versuche von Anwendern, Android auf bestehenden Telefonen laufen zu lassen. Das setzte allerdings einige Anpassungen voraus und aufgrund der fehlenden Offenlegung des Quellcodes kommt es zu einigen Einschränkungen. Berichten zufolge war es trotzdem möglich Android auf dem Nokia N800/N810 zu starten.

Nach der Veröffentlichung des kompletten Quellcodes ist zu erwarten, dass es inoffizielle Versionen und Anpassungen für die verschiedensten Telefone und Geräte geben wird.

2.3.5 Kritische Betrachtung von Android

Doch auch Android ist nicht vor jeder Kritik gefeit, und so muss Google auch mit unzufriedenen Entwicklern leben, die vor allem die Nachteile der Plattform in den Fokus rücken. Diese sollen im Folgenden kurz erläutert werden.

2.3.5.1 Mangelnde Informationspolitik

Im Sommer 2008 gab es von Google-Seite immer weniger Informationen und Veröffentlichungen zu Android. Zwischen März und August bot Google keine neuen Versionen des SDK an und gab auch keine weiteren Details bekannt. Außerdem kamen Gerüchte auf, dass die Gewinner der ersten Runde der Android Developer Challenge mit internen Versionen versorgt wurden, was später unbeabsichtigt von Google bestätigt wurde (vgl. Google, 2008c). Das führte zu einer zunehmenden Unzufriedenheit vieler Entwickler. Auch wurde erst sehr spät eine „Roadmap“[21] veröffentlicht.

2.3.5.2 Fehlende Funktionen der Android-API

Obwohl die Android-API viele Anforderungen abdeckt, fehlen verschiedene Funktionen. So wird das Android SDK in der Version 1.0 laut Google keine Bluetooth-API für Entwickler enthalten. Einfache Bluetooth-Anwendungen wie Headsets werden allerdings trotzdem unterstützt. Des Weiteren gibt es keinen Support für native Anwendungen sowie Bibliotheken und deren Anbindung an Dalvik VM-Anwendungen per Java Native Interface[22] (JNI) (vgl. Kapitel 3.2.8). Es wurde allerdings ein natives SDK angekündigt, das JNI vollständig und korrekt unterstützt (vgl. Guy, 2008). Als Grund für die fehlenden oder entfernten Funktionen nannte Google die mangelnde Zeit bis zur Veröffentlichung des SDK in Version 1.0 (vgl. Morril, 2008b).

Ein virtuelles Keyboard fehlt als Funktion ebenfalls in Version 1.0. Es stellt sich die Frage, ob es dadurch erst einmal keine Android-Telefone ohne Tastatur geben wird, oder ob in Kürze ein weiteres Android-Release mit einer virtuellen Tastatur für Texteingaben veröffentlicht wird. Besonders interessant ist die Frage deshalb, weil für die Telefonie-Anwendungen ein virtuelles Nummernfeld für Eingaben bereits enthalten ist.

Insgesamt erscheint die Veröffentlichung etwas übereilt um ein vollständiges System zu erstellen. Besonders die erst spät entfernten Funktionen erwecken den Eindruck, als ob Google Zeitprobleme gehabt hätte. Allerdings kommt Android, verglichen etwa mit dem iPhone, verspätet auf den Markt, so dass eine Verschiebung nicht in Frage kam.

[21] vgl. `http://source.android.com/roadmap` (aufgerufen am 24.11.2008)

[22] Eine Schnittstelle, mit der man aus der Java Virtual Machine eine native Bibliothek ansprechen kann. Es ist auch möglich aus einer nativen Bibliothek oder Anwendung Javacode auszuführen.

2.3.5.3 Wie offen ist „Open“?

Neben der Unzufriedenheit mit der Informationspolitik und fehlenden API-Funktionen steht Google auch in der Kritik, lange Zeit nur den Quellcode des angepassten Kernels von Android offengelegt zu haben, obwohl die Offenlegung von der OHA für die komplette Android-Plattform umfassend angekündigt worden war (vgl. Aliance, 2008). Seit dem 21. Oktober 2008 ist der vollständige Quellcode unter `http://source.android.com` verfügbar.

Außerdem werden nicht wie ursprünglich angekündigt alle Anwendungen vollkommen gleichberechtigt sein. Laut Informationen von Google-Mitarbeitern werden zumindest die Telefonie-Anwendungen nicht ohne weiteres austauschbar sein, da sowohl die Telefonie-API als auch eine API für SMS fehlt. Letztere war in den SDK vor Version 1.0 noch vorhanden. Darüber hinaus verwenden noch andere Anwendungen des Release 1.0 nicht öffentliche, bzw. auf spezielle Zertifikate beschränkte Methoden und Klassen der API. Dazu gehören die Kontakte und die Android Market-Anwendung. Google behält sich auch das Recht vor, Anwendungen die gegen Bestimmungen verstoßen, von den Telefonen zu löschen[23].

Viel größer ist aber das Problem zu bewerten, dass Android zwar eine offene Plattform bilden soll, ein offene Plattform aber nicht zwangsläufig bedeutet, dass auch die speziellen Lösungen für die Geräte offen und beliebig anzupassen sind. So ist ein *root*-Zugang auf einem HTC G1 nicht unterstützt und in verschiedenen Firmware-Revisionen nur durch ausnutzen von Sicherheitslücken zu erreichen. Es ist auch nicht möglich den öffentlich verfügbaren Quellcode anzupassen, da in Diesem gerätespezifische Treiber und verschiedene Google-Anwendungen fehlen. Um nationalen Bestimmungen bezüglich der Einhaltung von Vorschriften für Mobiltelefone nachzukommen, werden diese Treiber vermutlich auch in Zukunft nicht offen zugänglich sein. Vielmehr bietet Android für Hersteller denen die Spezifikationen der betroffenen Chipsätze zugänglich sind, die Möglichkeit eine offene Plattform für ihr Gerät zu nutzen.

Der Ansatz von Google und der OHA verspricht eine interessante neue Mobiltelefon-Plattform, aber durch die Einschränkungen bezüglich der Offenheit unterscheidet sich diese nicht sehr von Alternativen wie zum Beispiel Symbian, vor allem dann, wenn Symbian nach der Überführung in die Symbian Foundation in Kürze ebenfalls geöffnet wird.

2.4 Voice over IP-Lösungen für Mobiltelefone

Durch die Verbreitung von günstigen Internetzugängen für das Mobiltelefon wie zum Beispiel UMTS Datentarife, bietet sich durch VoIP eine Alternative zu klassischen Mobiltelefonaten. Allerdings wird VoIP von den Mobiltelefon-Netzbetreibern in Deutschland behindert. So untersagt Vodafone in seinen Geschäftsbedinungen die Nutzung von VoIP-Diensten. In England hingegen gibt es mit „3“ einen Mobilfunkanbieter, der speziell VoIP anbietet. (vgl. Minnerup, 2008)

[23] Zu finden in den Market Policies `http://www.google.com/intl/en_us/mobile/android/market-policies.html` (aufgerufen am 17.10.2008)

Im Folgenden werden die bestehenden Möglichkeiten für VoIP und die VoIP-Qualitätsmessung auf Mobiltelefonen erläutert.

2.4.1 Voice over IP unter Android

VoIP ist unter Android von Google nicht vorgesehen und wird durch die API nicht unterstützt. Auch nach der Veröffentlichung des HTC G1 gibt es keine VoIP-Lösungen oder Möglichkeiten zur Messung der Qualität von VoIP auf der Android-Plattform. Es sind allerdings VoIP-Lösungen für Android angekündigt und weitere werden vermutlich folgen[24].

2.4.2 Voice over IP auf vergleichbaren Mobiltelefon-Plattformen

Auf nur wenigen Mobiltelefonen sind VoIP-Anwendungen im Auslieferungszustand vorhanden, zum Beispiel auf den Nokia-Telefonen N80, e51 und e60 unter dem Symbian Betriebssystem. Auch die Telefone von Samsung und der BlackBerry unterstützen VoIP (vgl. Minnerup, 2008). Auf anderen Telefonen und Plattformen wie Windows Mobile muss VoIP-Software nachträglich installiert werden, ist dann aber voll funktionsfähig.

Auf dem iPhone gibt es zur Zeit (September 2008) nur eine VoIP-Lösung die offiziell im „App-Store“ erhältlich ist.[25]. In dieser Anwendung findet allerdings keine Messung der VoIP-Qualität statt. Darüber hinaus gibt es noch VoIP-Software, die über inoffizielle Kanäle für angepasste iPhones angeboten wird. Dieser Vertrieb wird aber aktiv von iPhone-Vertriebspartnern wie T-Mobile behindert[26].

2.4.3 Voice over IP-Qualitätsmessung auf Mobiltelefonen

Viele der bekannten Werkzeuge für die VoIP-Qualitätsmessung werden nicht auf den Endgeräten, sondern ausschließlich innerhalb der Netzwerk-Infrastruktur eingesetzt. So ist lediglich eine Analyse der Qualität innerhalb des Netzwerks und keine vollständige Ende-zu-Ende-Analyse möglich. Außerdem untersuchen diese Lösungen vor allem die QoE- und QoS-Parameter.

Nur „VQmon/EP“[27] bietet eine embedded Lösung direkt auf den Endgeräten, womit sich die Qualität Ende-zu-Ende analysieren lässt. Diese Lösung konzentriert sich aber auf QoE mit der Analyse der Audiodaten im DSP. Des Weiteren werden noch RTCP und über die Signalisierung erhaltene QoS-Parameter ausgewertet. Im Gegensatz zur VoIPFuture Library werden dabei nicht die RTP-Pakete analysiert.

[24] `http://www.voxofon.com/news.html` (aufgerufen am 2.10.2008)

[25] `http://www.apple.com/iphone/appstore/content/traveltop9application.html` (aufgerufen am 18.9.2008)

[26] `http://www.heise.de/newsticker/meldung/115747` (aufgerufen am 17.9.2008)

[27] `http://www.telchemy.com/vqmonep.html` (aufgerufen am 28.10.2008)

3 Analyse

In diesem Kapitel werden in der fachlichen Analyse die erwarteten Szenarios vorgestellt und darauf aufbauend die nicht-funktionalen und funktionalen Anforderungen geklärt. In der darauf folgenden technischen Analyse werden die Voraussetzungen der Android-Plattform analysiert, bestehende SIP- und RTP-Bibliotheken vorgestellt die für die zu entwickelnde Android-VoIP verwendet werden können und die Abhängigkeiten der VoIPFuture Library geklärt.

3.1 Fachliche Analyse

Wie in Kapitel 2.4.1 festgestellt und von der Aufgabenstellung in Kapitel 1.1 gefordert, ist es notwendig für VoIP-Qualitäts-Messungen unter Android eine VoIP-Anwendung inklusive eines SIP- sowie RTP-Stacks zu entwickeln. Dafür sind zunächst die erwarteten Szenarios genauer zu untersuchen.

3.1.1 Zu erwartende Szenarios

Im Rahmen dieser Arbeit werden nur grundlegende VoIP-Szenarios untersucht, da sich komplexe Szenarios vor allem auf der Ebene der SIP-Signalisierung abspielen. Die Signalisierung spielt bei QoA und QoD allerdings nur eine untergeordnete Rolle. So ist es für eine Qualitätsmessung zum Beispiel nicht notwendig, eine Registrierung bei einem SIP-Proxy zu ermöglichen. Vielmehr wird die VoIP-Anwendung nur in einem Endpunkt-zu-Endpunkt-Szenario verwendet (vgl. Abbildung 3.1).

3.1.1.1 Use Case „VoIP-Telefonat durchführen“

Für die Anwendung gibt es keinen Unterschied zwischen einem Anwender, der ein VoIP-Telefonat durchführt, und einem VoIP- oder Netzwerk-Spezialisten der einen Testanruf tätigt, um zum Beispiel die Infrastruktur des Netzes zu testen.

Eigenständiger Gesprächsaufbau

Der Anwender baut in diesem Use Case eine neue Verbindung auf, indem eine Gesprächsanfrage an den entfernten User-Agent versendet wird. Dabei muss sowohl auf eine Annahme, als auch auf eine Ablehnung des Gesprächs korrekt reagiert werden.

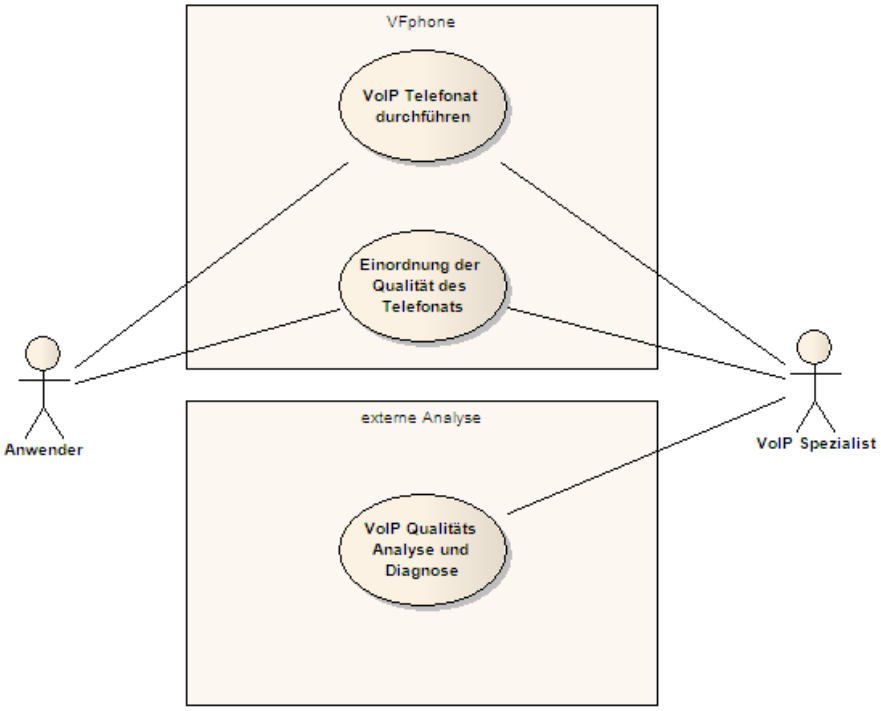


Abbildung 3.1: Use Case Diagramm der VoIP-Anwendung

Beantwortung einer Gesprächsanfrage

Bei Eingang einer Gesprächsanfrage benachrichtigt die Anwendung den Anwender und signalisiert den Empfang der Anfrage auch der Gegenstelle. Der Anwender entscheidet nun über die Annahme des Gesprächs und die Anwendung teilt die Annahme der Gegenstelle mit.

Abbau einer Verbindung

Am Ende des jeweiligen Gesprächs wird die Verbindung auf Wunsch des Anwenders aktiv abgebaut oder es reagiert die Anwendung auf eine Anfrage der Gegenstelle zum Abbau und beendet das Gespräch.

3.1.1.2 Use Case „VoIP Qualitätsanalyse und -diagnose“

Für die genaue Analyse und Diagnose werden die Analysevektoren der VoIPFuture Library benötigt. Diese werden bei jedem Gespräch regelmäßig von der VoIPFuture Library angefordert und in eine Datei exportiert. Diese Dateien können in Werkzeuge von VoIPFuture, wie zum Beispiel dem VoIPFuture Manager, importiert werden, um genaue Diagnosen abzuleiten.

Gleichzeitig wird schon während des Telefongesprächs die aktuelle Qualität über eine Einteilung in „Gut“, „Warnung“, „Schlecht“ und „Undefiniert“ im Display des Telefons angezeigt. Dadurch erhält der Anwender eine direkte Rückmeldung.

3.1.2 Nicht-funktionale Anforderungen des VoIP-Clients

Die nicht-funktionalen Anforderungen bestimmen den Rahmen und die grundlegenden Eckpunkte, welche die VoIP-Anwendung erfüllen muss. Sie sind ein wesentlicher Bestandteil der Software und beeinflussen daher auch maßgeblich den Entwurf und die Realisierung.

3.1.2.1 Performanz und Robustheit

Für die Qualität einer VoIP-Verbindung ist Performanz und Robustheit der Anwendung besonders essentiell. Hier ist besonders der RTP-Stack hevorzuheben. Wie schon in Kapitel 2.2.3 festgestellt ist es bei VoIP wichtig, die RTP-Pakete möglichst gleichmäßig, gemäß der Sample Rate, auszusenden.

Aber auch die Verzögerung der Pakete darf einen bestimmten Wert nicht überschreiten. Nach der ITU-T-Empfehlung G.114 sollte die Signalverzögerung für ein gutes Signal 150ms nicht überschreiten, unter 400ms ist das Signal noch akzeptabel (vgl. Trick und Weber, 2007). Das bedeutet, dass Echtzeit-Anforderungen auf die VoIP-Anwendung zutreffen. Dabei sollte sich die Anwendung auch von parallel laufenden Anwendungen nicht beeinträchtigen lassen.

Laut Google muss für eine gute Performanz vor allem vermieden werden, nicht notwendige und häufig Objekte zu erzeugen. Besonders auf temporäre Objekte sollte verzichtet werden (vgl. Google, 2008a). Neben der Wiederverwendung von Objekten ist eine weitere Möglichkeit die Objektdeklaration als `static`.

3.1.2.2 Erweiterbarkeit und Änderbarkeit

Die Erweiterbarkeit der VoIP-Anwendung muss ebenfalls beachtet werden. Wie in Kapitel 3.1.3 geschildert, gibt es beim Entwurf der Anwendung vor allem eine Ausrichtung auf die Möglichkeit der VoIP-Qualitätsmessung unter Android. Daher muss diese für eine weitere Verwendung leicht zu erweitern sein.

Es sollte des Weiteren möglich sein, zusätzliche Funktionen hinzuzufügen. Aber auch die verwendeten Bibliotheken müssen leicht auszutauschen sein. Das gilt auch für die Wartung von sonstigen Funktionen und Bestandteilen.

3.1.2.3 Portierbarkeit und Geräteunabhängigkeit

Eine Anwendung die für Mobiltelefone konzipiert wird, sollte zwecks einer großen Verbreitung möglichst leicht auf weitere Mobiltelefone portierbar sein. Das ist vor allem für die VoIP-Qualitätsmessung auf einem Mobiltelefon wichtig, da nur mit einer Messung an beiden Endpunkten eine vollständige Überwachung der Qualität möglich ist.

Für eine Anwendung auf einem Android-System spielt die Portierbarkeit allerdings nur eine untergeordnete Rolle, da das Framework mit der Virtual Machine die vorhandene Hardware vollständig kapselt. Aufgrund dessen ist jede Android-Anwendung automatisch weitgehend geräteunabhängig.

Eine Besonderheit bringt die Verwendung des JNI (vgl. Kapitel 3.2.8) mit sich. Dadurch wird auf eine hohe Portierbarkeit verzichtet, da native Bibliotheken immer von bestimmten Prozessor-Typen abhängig sind. Es ist daher für den nativen Teil der Anwendung wichtig, möglichst wenige Abhängigkeiten aufzuweisen (vgl. Kapitel 3.2.8.3).

3.1.2.4 Sicherheitsanforderungen

Sicherheit ist eine der wichtigsten nicht-funktionalen Anforderungen. Dieses gilt sowohl für die Sicherheit der Daten, als auch für die Sicherheit der Anwendung gegenüber bewussten Angriffen, fehlerhaften Benutzereingaben und sonstigen kritischen Fehlfunktionen der Umgebung oder der Anwendung. Zwar erscheint zum Beispiel eine Sicherheit gegenüber fehlerhaften Benutzereingaben im Rahmen einer prototypischen Entwicklung für einen VoIP-Qualitätstest nicht als zwingend. Trotzdem sollte diese aber nicht übersehen werden, auch im Zusammenhang mit der geforderten Erweiterbarkeit und Änderbarkeit.

Viele Anforderungen werden dabei von der Dalvik VM im Zusammenspiel mit dem Linux Kernel abgedeckt (vgl. Kapitel 3.2.4).

Neben der Sicherheit der Anwendung liegt das Augenmerk für eine VoIP-Anwendung vor allem auf der ständigen Verfügbarkeit, um auf eingehende Telefonate reagieren zu können (vgl. Kapitel 3.1.3.2).

3.1.2.5 Weitere nicht-funktionale Anforderungen

Auf Anforderungen wie Korrektheit, Vollständigkeit, Benutzbarkeit und Skalierbarkeit kann wie in jeder anderen Anwendung auch in einem Android-VoIP-Client nicht verzichtet werden. Allerdings spielen diese Anforderungen für die Betrachtung der VoIP-Qualität nur eine geringe Rolle. Aus diesem Grund werden sie auch im Rahmen dieser Arbeit nicht weiter betrachtet. Allerdings kann man sie im Hinblick auf eine weitere Verwendung und die Erweiterbarkeit der Anwendung nicht vollkommen außer Acht lassen.

3.1.3 Funktionale Anforderungen des VoIP-Clients

Die Android-VoIP-Anwendung muss im Rahmen dieser Arbeit die folgenden funktionalen Anforderungen erfüllen.

3.1.3.1 „Must have“-Anforderungen

„Must have“-Anforderungen sind unbedingt benötigte Anforderungen an die VoIP-Anwendung um VoIP-Qualitätstests zu ermöglichen, die Szenarios aus Kapitel 3.1.1 zu gewährleisten und die in Kapitel 3.1.2 vorgestellten nicht-funktionalen Anforderungen erfüllen zu können.

Auf- und Abbau einer SIP-Verbindung

Der Auf- und Abbau einer SIP-Verbindung ist eine der absolut zentralen Funktionen für die Messung der VoIP-Qualität. Die Anforderung beinhaltet die Unterstützung für ein `INVITE` und die dazu passenden Antworten (vgl. Tabelle 3.1), als auch den Abbau einer Verbindung über das Kommando `BYE`. Für die korrekte Funktion der Anwendung ist dabei die Unterstützung von SIP-Transaktionen und SIP-Dialogen unerlässlich.

SIP Kommando	Typ	Beschreibung
`INVITE`	Anfrage	Einladung eines SIP-Endpunkts zu einem Gespräch
`OK`	Antwort	Bestätigung einer Anfrage
`ACK`	Antwort	Bestätigung eines empfangenen SIP-Pakets
`TRYING`	Antwort	Antwort auf ein `INVITE`
`RINGING`	Antwort	Mitteilung über ein Klingeln beim Endgerät
`CANCEL`	Antwort	Abbruch eines SIP-Dialogs
`BYE`	Anfrage	Abbau eines SIP-Dialogs
`OPTIONS`	Anfrage	Abfrage der unterstützten SIP-Kommandos sowie des aktuellen Status

Tabelle 3.1: Unterstützte SIP-Anfragen und -Antworten

Neben der Unterstützung von SIP ist auch das Aushandeln des Audioformats und der RTP-Adressen und -Ports über den „3-Way-Handshake“ mittels SDP (vgl. Abbildung 2.2) von essentieller Bedeutung für die Übertragung der Audiodaten.

Übertragung von Audiodaten über RTP

Die Übertragung der Audiodaten über RTP und die Verwendung von QoA und QoD spielt für diese Arbeit eine zentrale Rolle. Dabei müssen die Daten vor und nach der Übertragung in ein per SDP ausgehandeltes Audio-Format umgewandelt und entsprechend den ebenfalls ausgehandelten Portnummern und Adressen übertragen werden.

Analyse der RTP-Pakete für QoA und QoD

Nach dem Versand und Empfang der jeweiligen RTP-Pakete ist es für QoA notwendig, diese der VoIPFuture Library zu übergeben.

Für eine Qualitätsmessung ist auch die Verwendung von „tcpdump“ oder „wireshark“ möglich. Allerdings wäre dann eine Portierung von „tcpdump“ auf die Android-Umgebung notwendig. Damit ist außerdem nur eine Verwendung in einer Laborumgebung möglich, da die Daten zuerst exportiert und dann auf einem weiteren System analysiert werden müssen.

Bei der Verwendung der VoIPFuture Library auf dem jeweiligen Mobiltelefon ist eine sofortige Analyse möglich.

3.1.3.2 „Should have“-Anforderungen

Die beschriebenen Anforderungen sind nach Kapitel 3.1.1 wünschenswert, aber nicht zwingend erforderlich für diese Arbeit und VoIP-Qualitätstests.

Verbindung zu einem beliebigen Endpunkt aufbauen

Die Oberfläche der VoIP-Anwendung soll den Aufbau einer Verbindung zu einem beliebigen Endpunkt ermöglichen. Das erleichtert zum einen die Entwicklung, zum anderen auch die Messungen der VoIP-Qualitätstests. Darüber hinaus ist es in einer weiteren Verwendung der Anwendung notwendig für Endanwender.

Für die Qualitätsmessung ist es hilfreich, auf der Gegenseite zur Android-Anwendung eine unabhängige SIP-Anwendung zu verwenden, um mögliche Einschränkungen und Fehler in der Android-VoIP-Anwendung zuverlässig zu erkennen. Allerdings muss diese unabhängige Anwendung nicht fehlerlos arbeiten, da es mit Hilfe der Lösungen von VoIPFuture möglich ist, deren Qualität zu messen und damit eventuelle Fehler zu erkennen und zu berücksichtigen.

SIP- und RTP-Stack zu jeder Zeit ansprechbar

Besonders für eine weitere Verwendung ist es sinnvoll, wenn die Anwendung zu jeder Zeit auf eingehende Anfragen reagiert. Andernfalls wäre es nur möglich ein Gespräch zu führen oder einen eingehenden Anruf anzunehmen, solange die VoIP-Anwendung im Vordergrund läuft.

Die Analysevektoren der VoIPFuture Library exportieren

Die VoIPVector-Daten der VoIPFuture Library werden in eine Datei exportiert, um diese in externen VoIPFuture-Diagnose-Werkzeugen genauer auszuwerten.

3.1.3.3 „Nice to have“-Anforderungen

Diese Anforderungen sind im Rahmen dieser Arbeit nicht notwendig, aber für eine weitere Verwendung außerhalb möglicherweise sinnvoll.

Qualitätsanzeige

Es ist möglich dem Benutzer schon während des Telefongesprächs die aktuell gemessene und zu einem einzelnen Wert zusammengefasste Qualität über Icons mitzuteilen. Die Anzeige veranschaulicht dem Anwender außerdem die Durchführung der Qualitätsmessung.

Erzeugung von Fehlermustern

Zum Test der Analysewerkzeuge und der Implementierungen ist es wünschenswert, Fehlermuster wie zum Beispiel Paketverlust, eine fehlerhafte Paketreihenfolge und veränderbare Paketlaufzeiten parametrisierbar zu erzeugen.

Unterstützung von Mikrofon und Lautsprecher

Im Rahmen der vorliegenden Arbeit ist es während der Entwicklung für die VoIP-Qualitätsmessung hilfreich, ohne Aufwand definierte Audiodaten abspielen zu können. Allerdings ist die Unterstützung von Mikrofon und Lautsprecher ist für eine produktive VoIP-Anwendung unverzichtbar, um dem Anwender eine Kommunikation zu ermöglichen.

Unterstützung von RTCP

Für eine weitergehende Analyse der VoIP-Verbindung kann es hilfreich sein die, von anderen Teilnehmern über RTCP versendeten, QoS- und QoE-Parameter auszuwerten. Zusätzlich ist die Unterstützung von RTCP für die Teilnahme an VoIP-Telefonkonferenzen notwendig.

Außerdem können die Ergebnisse der Analyse und Diagnose der VoIPFuture Library über RTCP mit anderen User-Agents ausgetauscht werden oder an einen zentralen Monitor weitergegeben werden.

3.2 Technische Analyse

In den nächsten Abschnitten werden die technischen Gegebenheiten für die Entwicklung eines VoIP-Clients und die Messung der VoIP-Qualität unter Android geklärt. Dafür wird die Android-Plattform näher analysiert und der technische Aufbau untersucht. Darüber hinaus werden mögliche SIP- und RTP-Bibliotheken vorgestellt und die Voraussetzungen für die VoIPFuture Library geklärt.

3.2.1 Android Development Kit

Google legt großen Wert darauf, dass ein Entwickler nicht alle Interna eines Mobilgerätes kennen muss und sich so voll auf die eigene Anwendungs-Entwicklung konzentrieren kann. Zu diesem Zweck steht ein umfangreiches SDK mit vielen speziellen Werkzeugen und Hilfen zur Verfügung. Als SDK für Android wird das ADK von Google (vgl. Google, 2008a) bereit gestellt. Das ADK beinhaltet neben den Java-Klassen der Android-API auch viele Werkzeuge für Entwickler:

dx wandelt vom Java Compiler erzeugte „.class“-Dateien in das Dalvik-VM-Binärformat „.dex“ um.

aapt erstellt ein Android-Archiv „.apk“, in dem alle Dateien einer Android-Anwendung zusammengefasst sind. Zu vergleichen ist das Archiv mit einer Java-„.jar“-Datei.

aidl erstellt „.aidl“-Dateien für die Interprocess-Kommunikation (vgl. Kapitel 3.2.6).

adb ermöglicht den Zugriff auf das Android-Dateisystem im Emulator und auf dem Telefon. Zudem ist es möglich, ein Kommando und eine Kommandozeile im Emulator oder auf dem Telefon auszuführen.

ddms ermöglicht den Zugriff des Entwicklers auf das laufende Android-System. Dabei ist es möglich, Prozesse und Threads zu überwachen und diese in den Debug-Modus zu schalten. Außerdem bietet ddms den Zugriff auf das Dateisystem, wie auch auf die Android-Logdateien.

3.2.1.1 Emulator

Neben den vorgestellten Werkzeugen und der API stellt der Emulator das zentrale Element des ADK dar. Er ermöglicht die einfache und schnelle Entwicklung und den Test von Android-Anwendungen ohne zusätzliche Hardware direkt auf dem PC des Entwicklers. Die vollständige Android-Umgebung inklusive des Kernels, der nativen Bibliotheken und den Standard-Anwendungen ist in ihm enthalten.

Der Emulator basiert auf dem Open Source-Projekt „qemu“[1], emuliert die Android-Umgebung auf einem ARM-Prozessor und ermöglicht den Zugriff auf das Netzwerk und emulierten SD-Karten. Zusätzlich ist es möglich, den Empfang von SMS zu simulieren und Telefonate zu steuern. Auch die Netzwerkbandbreite kann entsprechend der Vorgaben limitiert werden. (vgl. Google, 2008a)

3.2.1.2 Android Eclipse-Plugin

Google bietet zusätzlich zu dem ADK ein umfangreiches Eclipse-Plugin, das viele Aufgaben für den Entwickler automatisiert. So wird der Build-Prozess mit der Übersetzung durch den Java-Compiler, dx, aapt sowie das Deployment durch adb und dem Start der Anwendung auf dem Emulator in einem Schritt zusammengefasst. Außerdem bindet das Plugin das ddms-Werkzeug direkt in die Eclipse-Entwicklungsumgebung ein und erlaubt mit dem Debugger einen einfachen Zugriff auf die Anwendung während der Laufzeit.

3.2.2 Aufbau und Design von Android

Android hat eine dreischichtige Architektur (vgl. Abbildung 3.2). Dabei beschreibt Android nicht nur eine API, sondern beinhaltet auch des Betriebssystem mit nativen Bibliotheken und dem für Android speziell angepassten und erweiterten Linux-Kernel. Die unterste Schicht enthält den Kernel mit den Gerätetreibern. In der mittleren Schicht befinden sich die nativen Bibliotheken, die von der obersten Schicht, der Dalvik VM, verwendet werden. (vgl. Morrill, 2008)

[1] `http://bellard.org/qemu/` (aufgerufen am 27.10.2008)

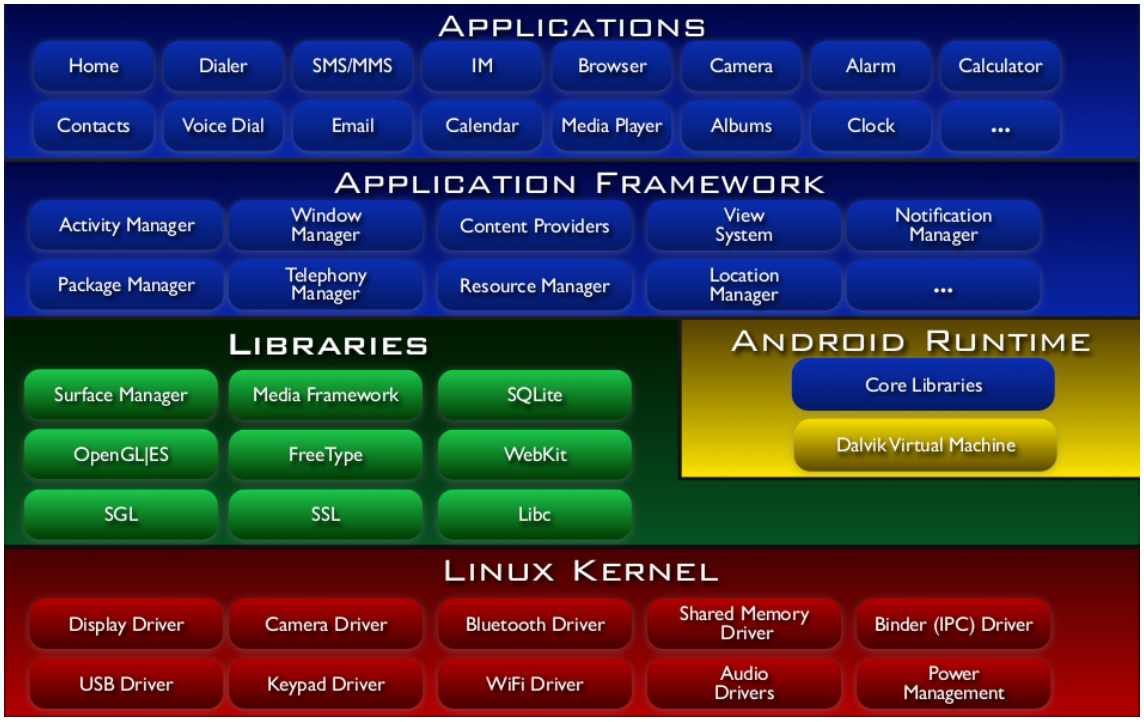


Abbildung 3.2: Architektur von Android (vgl. Google, 2008a)

3.2.2.1 Kernel-Schicht

Die Kernel-Schicht von Android besteht ausschließlich aus dem von Google angepassten Linux Kernel, zur Zeit (SDK 1.0r1) in der Version .2.6.25-00350-g40fff9a. Im Kernel befinden sich neben den Treibern für die Hardware und Unterstützung für die MSM7xxx Chips von Qualcomm auch die grundlegenden Implementierungen des Berechtigungskonzepts von Android. Des Weiteren wurde der Kernel von Google um die Unterstützung für die Android-Inter-Process Communication (IPC) erweitert.

3.2.2.2 Bibliotheken-Schicht

Die meisten nativen Bibliotheken in der mittleren Schicht von Android sind, gegenüber Implementierungen auf anderen Plattformen, stark beschnitten und konzentrieren sich ausschließlich auf die von der Dalvik VM benötigten Funktionen. Dabei baut Google auf eine sehr stark abgespeckte libc auf, *Bionic* genannt, die von BSD-Implementierungen abgeleitet wurde. Diese Implementierung wurde vor allem auf embedded Linux-Geräte angepasst. (vgl. Bornstein, 2008)

In der aktuellen Version wird nicht von Google vorgesehen oder unterstützt, dass Anwendungen diese Schicht durch eigene Bibliotheken erweitern können, es ist allerdings trotzdem grundsätzlich möglich. Außerdem wird sich die Situation nach der angekündigten Veröffentlichung des vollständigen Quellcodes durch Google ändern und angepasste Versionen von Android mit weiteren nativen Bibliotheken realisierbar sein.

Neben Standard-Bibliotheken, wie zum Beispiel der libc und der libmath, befinden sich in dieser Schicht zusätzlich noch die nativen Implementierungen für grundlegende Funktionen wie eine Portierung von SQLite oder OpenGL die dann von der API innerhalb der Dalvik VM verwendet werden.

Die Dalvik VM selbst kann auch dieser Schicht zugeordnet werden, da sie als eine native Anwendung aufgeführt wird und direkt auf dem Kernel und den nativen Bibliotheken aufbaut.

3.2.2.3 Anwendungs-Schicht

In der dritten und obersten Schicht befinden sich Anwendungen, die direkt vom Benutzer verwendet werden. Darüber hinaus gehören alle Services und die Bibliotheken, die direkt vom Entwickler über die Android-API angesprochen werden können, in diese oberste Schicht. Dadurch lässt sie sich nochmals in zwei verschiedene Ebenen aufteilen. Zum einen gibt es die Anwendungen und die Services. Zum anderen gehören die Services des Framworks zu dieser Schicht, wie zum Beispiel dem Window Manager und verschiedenen Content Providern.

3.2.3 Dalvik Virtual Machine

Die Laufzeitumgebung und damit den Kern des Android-Frameworks bildet die Dalvik VM. Dieser Abschnitt beschäftigt sich mit den Grundlagen und stellt die Unterschiede zur Java VM von Sun heraus.

Die Dalvik VM wurde von Google-Mitarbeitern vollständig neu entwickelt und implementiert die Kern-Bibliotheken eines J2SE unter Android. Dabei ist die Dalvik VM für mobile Geräte vor allem für die ARM Architektur optimiert (vgl. Brady, 2008). Außerdem sind Portierungen auf andere Architekturen möglich und sollen in Zukunft auch durchgeführt werden. Für alle hardwarenahen Funktionen wie „threading“ und „memory managment“ wird auf den fest zum Android-Framework gehörenden Linux-Kernel zurückgegriffen. Dabei ist die VM so konzipiert, dass sie auch auf einem langsamen Prozessor mit wenig Hauptspeicher laufen kann und möglichst wenig Strom verbraucht.

Auf der Android-Plattform wird jede Anwendung in einer eigenen Instanz der VM gekapselt. So ist gewährleistet, dass sich die verschiedenen Anwendungen nicht gegenseitig beeinflussen können und damit das gesamte System nicht durch eine einzelne instabile Anwendung abstürzen kann. Dadurch können aber Funktionen wie IPC nicht innerhalb der VM realisiert werden, sondern müssen in den Kernel ausgelagert werden.

3.2.3.1 .dex Dateien

Der Quellcode wird zuerst durch den Sun Java Compiler compiliert und dann durch einen Konverter in einen dalvikspezifischen Bytecode übertragen, bei dem auf möglichst geringen Speicherverbrauch geachtet wurde. Das Resultat wird in Dalvik Executable-Dateien (.dex)[2] abgelegt. Das Vorgehen hat den Vorteil, dass nicht nur Quellcode in der Java-Programmiersprache benutzt werden kann, sondern auch Quellcode in anderen

[2] Eine Beschreibung des .dex Formats ist bei den Handouts zu (vgl. Bornstein, 2008) zu finden.

Sprachen, der mit bestehenden Compilern in .class Dateien übersetzt wurde. Zur Zeit funktioniert das Umwandeln allerdings nicht für Dateien die mit dem Python-zu-Java-Compiler Jython[3] erstellt wurden, da der Dalvik-Compiler den damit erzeugten Java-Bytecode nicht akzeptiert (vgl. Justin, 2008).

3.2.3.2 Speicherbelegung einer VM

Ein grundlegendes Problem für die VM ist, dass nur wenig Speicher auf einem Mobiltelefon zur Verfügung steht. Die Dalvik VM muss auch mit nur insgesamt zur Verfügung stehenden 64MB laufen, die des Weiteren von mehreren, vollkommen unabhängigen und in verschiedenen Adressräumen laufenden Prozessen geteilt werden.

Deshalb versucht man Speicher zu sparen, indem Wiederholungen in den .dex-Dateien vermieden werden. So können in mehreren .class-Dateien einer Anwendung, die vom Sun Java-Compiler erzeugt wurde, die gleichen Methodensignaturen vorhanden sein. Android führt diese verschiedenen .class-Dateien zu einer einzelnen .dex-Datei zusammen und spart damit die zusätzlichen Signaturbeschreibungen ein. Das Resultat wird auch als „shared constant pool“ bezeichnet. Dadurch ist eine unkomprimierte .dex-Datei weniger als halb so groß wie eine damit vergleichbare unkomprimierte .class-Datei.

Beim Start einer Anwendung unter Android führt der Zygote-Prozess einen normalen Unix `fork()` aus und die Anwendung wird in dem erzeugten Zygote-Prozess gestartet. Der Speicher der Core Libs wird so von allen Dalvik VM und dem initialen Zygote-Prozess gemeinsam verwendet.

Dadurch, dass für jede Anwendung ein eigener, separater Prozess und damit auch separater Speicher vorhanden ist, muss es auch mehrere Garbage Collectors geben. Gleichzeitig sollten diese aber auch den gemeinsam genutzten Speicher der einzelnen Prozesse respektieren. Trotzdem kann es passieren, dass für einen neuen Prozess nicht mehr genügend Speicher zur Verfügung steht. Das normale Verhalten vom Linux Kernel wäre in diesem Fall, den gerade gestarteten Prozess zu beenden, da für diesen nicht genügend Speicher zur Verfügung steht. Android kennt aber die jeweils zur aktuellen Zeit nicht mehr dringend benötigten Prozesse, zum Beispiel alle zur Zeit nicht sichtbaren und angehaltenen Anwendungen. Dadurch kann die Android-Plattform diese Prozesse beenden, und der neu zu startetende Prozess kann ausgeführt werden. Man kann Android deswegen als „Virtual Process System“ anstelle eines „Virtual Memory Systems“ bezeichnen. (vgl. Hackborn und Parks, 2007)

3.2.3.3 Performanz der Virtual Machine

Android wurde für Prozessoren mit nur 100MHz bis 500MHz entwickelt, mit einer Bus-Geschwindigkeit von 100MHz sowie 16-32kB Daten Cache. Diese Daten entsprechen ungefähr einem Desktop-System von vor circa 10 Jahren. Daher muss die Dalvik VM entsprechend optimiert werden, um ausreichende Performanz zu bieten.

[3] `http://www.jython.org/Project/` (aufgerufen am 16.9.2008)

Im Gegensatz zur Java J2SE VM besitzt die Dalvik VM keinen „Just in Time Compiler“. Also wird der Bytecode nicht zur Laufzeit übersetzt, sondern lediglich interpretiert. Zum einen verringert sich durch das fehlende Übersetzen der Speicherverbrauch und zum anderen werden Prozessor-Zyklen bei der Ausführung eingespart und die Reaktionszeit deshalb stark verkürzt. Allerdings fehlen dadurch viele Möglichkeiten der Optimierung, wie zum Beispiel „caching“ von bereits ausgeführten und übersetzten Anwendungsteilen. Deshalb werden Anwendungen unter Android bereits bei der Installation optimiert (vlg. Kapitel 3.2.3.3). Es gibt aber auch für die Dalvik VM Planungen, diese durch einen „Just in Time Compiler“ zu erweitern (vgl. Morril, 2008a). Eine vollkommen interpretierte VM gilt im Vergleich zu einer „Just in Time“- oder „Compile-Ahead“-VM als langsamer. (vgl. Acher, 2003)

Aufwändige Operationen werden auf der Android-Plattform in native Bibliotheken ausgelagert. Für verschiedene Bereiche, wie zum Beispiel Grafik und Audio, ist in der Regel eine Hardware Unterstützung vorhanden (vgl. 3.2). Bei Anwendungen die für spezielle Aufgaben besondere Performanz benötigen rät Bornstein (2008), per JNI (vgl. Kapitel 3.2.8) eine native Bibliothek anzusprechen.

Aber auch der Entwickler ist gefordert, auf eine gute Performanz seiner Anwendung zu achten. So sollte er eine häufige Iteration mit Hilfe eines Iterators vermeiden und stattdessen über ein Array iterieren, da ein Array schneller von der Dalvik VM verarbeitet werden kann. Google gibt neben diesem Tipp noch weitere im Rahmen der Dokumentation. (vgl. Google, 2008a)

Lesen und interpretieren von XML ist normalerweise aufwändig. Trotzdem wird XML an vielen Stellen der Android-Plattform genutzt, da es im Entwicklungsprozess hilfreich ist. Um trotzdem eine zu große Last zu vermeiden, werden die XML-Dokumente zur Compile-Zeit ausgewertet und optimiert in einem Binärformat gespeichert. (vgl. Hackborn und Parks, 2007)

Vorbereitungen während der Installation

Es wird bereits während der Installation von einer Anwendung Arbeit verrichtet, die dadurch beim Ausführen eingespart werden kann. Dabei wird zum Beispiel der Code verifiziert und die Indizes und Offsets geprüft (type und reference safety).

Neben der Verifikation wird auch der Code optimiert. Auf einer ARM-Architektur ist zum Beispiel kein byte-swapping bzw. padding notwendig. Eine weitere Optimierung ist das direkte Einbetten von nativen Methoden („inlining“) und das Entfernen von leeren Methoden.

Außerdem wird bei der Installation jeder Anwendung eine jeweils auf dem System eindeutige Benutzer-ID vergeben. Diese spielt für Berechtigungen, zum Beispiel auf Dateisystemebene, eine wichtige Rolle.

Infinit Register Machine

Die Dalvik VM ist im Gegensatz zur stackbasierten Sun Java VM registerbasiert. Eine registerbasierte VM kann Anwendungen um bis zu 26,5 Prozent schneller ausführen als eine stackbasierte VM(vgl. Shi u. a., 2005). In diesen Wert eingerechnet sind schon die Nachteile durch den auf registerbasierten virtuellen Maschinen in der Regel benötigten größeren Bytecode. Dadurch, dass die Dalvik VM registerbasiert ist, gibt es

30 Prozent weniger Aufrufe und 35 Prozent weniger „Code Units“, allerdings mit dem Nachteil, dass 35 Prozent mehr Bytes im „instruction stream“ benötigt werden. Dadurch benötigt in einer .dex-Datei eine Code-Unit zwei Bytes, anders als in der Sun Java VM, wo eine Code Unit nur ein Byte benötigt. Diese Ersparnis ist der Grund, warum Sun sich für eine stackbasierte VM entschieden hat (vgl. Acher, 2003).

3.2.4 Sicherheitskonzept von Android

Sicherheit hat besonders für ein offenes System wie Android eine sehr große Bedeutung. Anwendungen für das iPhone werden einer zentralen Prüfung von Apple unterzogen und gelangen erst danach in den Apple „App Store“[4]. Eine solche zentrale Prüfung ist für Android nicht geplant, vielmehr muss das System selber für die Sicherheit sorgen und die verschiedenen Programme gegeneinander und gegenüber sensiblen Daten absichern.

Zusätzlich zu dieser Absicherung sollte es dem Benutzer auch gestattet werden, das Laufzeitverhalten der Programme zu kontrollieren. Dafür bietet Android ein spezielles Berechtigungssystem.

3.2.4.1 Vorteile durch die virtuelle Maschine und Garbage Collection

Durch die virtuelle Maschine, repräsentiert durch die Dalvik VM, muss der Entwickler seinen angeforderten Speicher nicht selbst freigeben. Darum kümmert sich der Garbage Collector, der den Speicher verwaltet. Nicht mehr benötigte Objekte werden automatisch freigegeben. So wird verhindert, dass eventuell die Laufzeitumgebung durch eine Freigabe von nicht alloziertem Speicher unvorhergesehen beendet wird oder, dass es Speicherlecks durch vergessene `delete()`- oder `free()`-Aufrufe gibt. Das ist auch ein wichtiger Teil des Java-Security-Models. (vgl. Venners, 2000)

Außerdem fehlen in der Programmiersprache Java, der Sun VM und auch unter Android Speicherzeiger und frei adressierbarer Speicher vollkommen. Dadurch können durch den Entwickler verursachte fehlerhafte Speicherzugriffe nicht vorkommen.

3.2.4.2 Absicherung der Prozesse

Einen zentralen Aspekt für die Absicherung der Prozesse untereinander spielt für Android der Linux-Kernel. Dieser schottet einzelne Prozesse umfangreich gegeneinander ab und garantiert ein stabiles System, auch wenn ein Prozess abstürzt. Dabei ist jede einzelne Android-Anwendung durch die Dalvik VM zusätzlich in einer eigenen Sandbox[5] gekapselt, die eine zusätzliche Barriere gegen Programmierfehler und Angriffsversuche darstellt.

[4] `http://www.apple.com/de/iphone/appstore` (aufgerufen am 16.9.2008))

[5] Die Sandbox ist ein abgeschlossenes und von der Umgebung abgeschottetes System, in dem Anwendungen ausgeführt werden.

Die bei der Installation vergebenen Benutzer-Nummern (vgl. Kapitel 3.2.3.3) werden verwendet, um die Daten auf den Dateisystemen abzusichern. So ist es nicht möglich, Dateien außerhalb des Anwendungsverzeichnisses zu erzeugen oder auf die Verzeichnisse der anderen Anwendungen zuzugreifen.

3.2.4.3 Berechtigungen

Es müssen nicht nur die Programme untereinander und gegen Fehlfunktionen abgesichert werden, sondern es muss auch dem Benutzer gestatten werden, die vollständige Kontrolle über sein Gerät zu behalten. Android beinhaltet für diesen Fall ein umfangreiches Berechtigungssystem. Der Entwickler legt seine benötigten Berechtigungen zur Entwicklungszeit fest und der Anwender muss diese für eine Installation bestätigen (vgl. Abbildung 3.3).

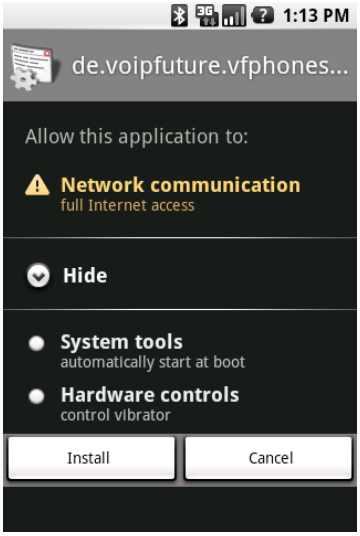


Abbildung 3.3: Abfrage der Berechtigungen bei der Installation des VoIP-Services VFphoneService

Allerdings ist es mit diesem System nicht möglich, als Benutzer nur Teile dieser Berechtigungen zu vergeben. Wünscht sich der Entwickler für nur einen kleinen Teil seiner Anwendung zum Beispiel einen Internet-Zugriff, kann der Anwender nur entscheiden, ob er diesem generell zustimmt und die Anwendung installiert, oder sie im anderen Fall gar nicht benutzen kann.

3.2.5 Anwendungen unter Android

Anwendungen unter Android lassen sich in vier Elemente aufteilen. Dabei muss eine Anwendung nicht alle Teile beinhalten, sondern besteht aus einer beliebigen Zusammenstellung dieser Elemente. (vgl. Google, 2008a)

3.2.5.1 Activity

Eine einzelne Bildschirmseite der Oberfläche einer Anwendung gehört zu einer Activity. Auf einem Desktop-Betriebssystem kann man eine Activity mit einem Fenster einer Anwendung vergleichen. Die meisten Anwendungen bestehen aus mehreren dieser Seiten, welche jeweils durch eine eigene Activity repräsentiert werden. Um auf eine neue Bildschirmseite zu gelangen, wird die dazu gehörige Activity gestartet. Dabei ist es sowohl möglich, Objekte der startenden Activity zu übergeben, als auch beim Beenden der Activity Objekte als „Antwort“ zu erhalten.(vgl. Google, 2008a)

Lifecycle einer Activity

Der Lifecycle einer Activity kann unter Android durch den Entwickler nicht beeinflusst werden. Dieser kann allerdings auf verschiedene Events mit dem Überschreiben von Methoden reagieren (vgl. Abbildung 3.4). Sobald eine Activity nicht länger im Vordergrund läuft, wird der Programmfluss angehalten. In diesem Status kann die Activity aber weiterhin angezeigt und nur teilweise von einer Sub-Activity verdeckt werden. Wird die Activity nicht mehr auf dem Bildschirm angezeigt, kann das System die Activity jederzeit beenden und aus dem Speicher entfernen. (vgl. Google, 2008a)

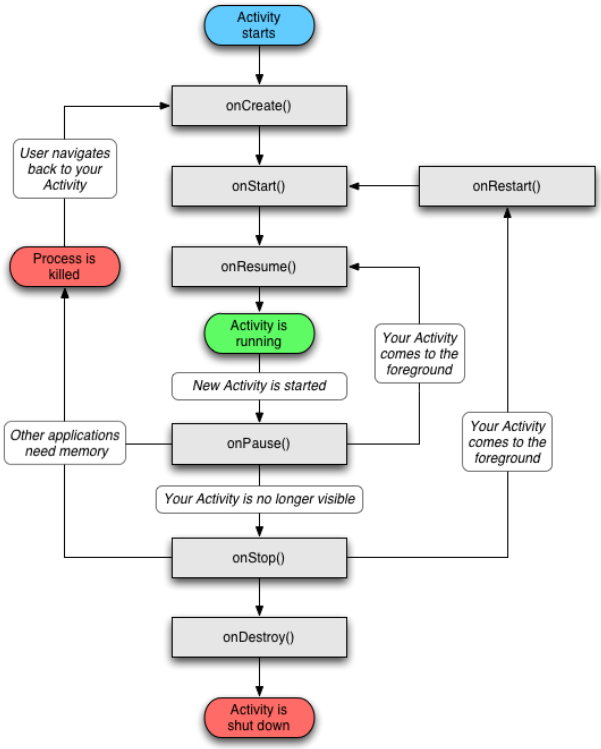


Abbildung 3.4: Lifecycle einer Activity (vgl. Google, 2008a)

Intent und Intent Filter

Der Klasse *Intent* kommt auf der Android-Plattform eine besondere Rolle zu. Sie wird verwendet um neue Activities zu starten. Diesen können dabei Objekte übergeben werden und sie können auch nach deren Beendigung Ergebnisse zurückliefern.

Es ist über einen *Intent Filter* möglich anzugeben, welche Nachrichten die Activity verarbeiten kann. Damit kann man das Intent-System mit einer einfachen Service-Oriented Architecture (SOA) vergleichen. Es ist möglich, das mehrere Activities den gleichen Intent-Typ verarbeiten können. In dem Fall erhält der Anwender eine Auswahlliste dieser Activities.(vgl. Google, 2008a)

3.2.5.2 Broadcast Intent Receiver

Neben Activities können auch *Broadcast Intent Receiver* Intents erhalten und verarbeiten. Allerdings beinhalten diese keine grafische Benutzeroberfläche. Sie können aber Benachrichtungen an den Benutzer senden oder Activities bzw. Services starten. (vgl. Google, 2008a)

3.2.5.3 Service

Die fehlende Fähigkeit einer Activity im Hintergrund zu arbeiten, wird durch Services behoben. Ein Service hat prinzipiell eine unlimitierte Laufzeit. Ein Service kann allerdings weder eine grafische Benutzeroberfläche beinhalten noch direkt vom Benutzer oder dem System gestartet werden. Das ist nur durch einen Aufruf aus einer Activity oder einem Broadcast Intent Receiver möglich.

Unter Android sollten alle Prozesse, die ständig oder aber unabhängig von der Oberfläche einer Activity ablaufen sollen, in einen Service verlagert werden (vgl. Google, 2008a). Ein Beispiel für einen Service ist das Abspielen eines Musikstücks. In diesem Szenario möchte der Benutzer eventuell im Hintergrund Musik hören, während er gleichzeitig mit einem Browser Webseiten besucht.

3.2.5.4 Content Provider

Zusätzlich zu den Dateien, SQLite-Datenbanken und weiteren Datenspeichern, die jede Activity und jeder Service verwenden kann, ist es unter Android möglich, einen *Content Provider* bereit zustellen. Dieser ermöglicht es, anwendungsübergreifend Daten abzulegen und abzurufen. (vgl. Google, 2008a)

3.2.6 Inter-Process-Communication im Android-Framework

Intents ermöglichen den Austausch von Informationen nur beim Start oder beim Beenden einer Anwendung. Das Android Framework bietet mit Android Interface Definition Language (AIDL) darüber hinaus eine Lösung für IPC an, um Daten und Objekte auch während der Laufzeit auszutauschen. Sie ist vergleichbar mit COM und Corba, allerdings deutlich leichtgewichtiger.

Mittels AIDL werden Objekte durch „Marshalling“ in primitive Datentypen umgewandelt und unter den Prozessen ausgetauscht. Ermöglicht wird das durch einen androidspezifischen Service, der im Kernelkontext läuft. Dieser Binder, basierend auf dem früheren Projekt OpenBinder[6] der Firmen Be[7] und Palm[8], ermöglicht den Austausch von Daten zwischen verschiedenen Android-Prozessen, obwohl diese, wie in Kapitel 3.2.4 beschrieben, ansonsten vollständig voneinander getrennt laufen. (vgl. Google, 2008a; Hackborn, 2008)

3.2.7 Threads innerhalb einer Android-Anwendung

Lang andauernde Berechnungen oder blockierende Aufgaben und Netzwerkzugriffe sollten unter Android vom Hauptthread mit der Benutzeroberfläche getrennt und in seperate Threads ausgelagert werden, um Verzögerungen in der Benutzeroberfläche zu verhindern. Darüber hinaus werden AIDL-Aufrufe ebenfalls grundsätzlich in seperaten Threads ausgeführt.

Von diesen seperaten Threads aus ist der direkte Zugriff auf den Thread der Benutzeroberfläche nicht möglich. Die Lösung für das Problem bietet die Android *Handler*-Klasse. Diese Klasse bietet einen Zugriff auf die *Message Queue* des Threads, über die man dem Thread Nachrichten zukommen lassen kann. (vgl. Google, 2008a)

3.2.8 Java Native Interface (JNI)

Trotz einer umfangreichen Java-API unter Android, besteht für viele Aufgaben die Notwendigkeit mit nativen Bibliotheken und Anwendungen zu interagieren. Die Notwendigkeit gilt sowohl für bereits bestehende und schwierig auf Java zu portierende Bibliotheken, als auch für performanzkritische Aufgaben (vgl. Kapitel 3.2.3.3).

[6] `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html` (aufgerufen am 28.10.2008)
[7] `http://www.beincorporated.com/` (aufgerufen am 4.12.2008)
[8] `http://www.palm.com/de/de/` (aufgerufen am 4.12.2008)

3.2.8.1 Einführung in JNI

Durch JNI ist es möglich, aus der Programmiersprache Java heraus auf plattformspezifischen Code auszuführen und Daten in Form von Java-Objekten oder primitiven Datentypen auszutauschen. Des Weiteren ist es ebenso möglich, aus dem nativen Code heraus auf Java-Klassen zuzugreifen und deren Methoden zu verwenden (invocation API). Dazu zählen auch Exceptions, die aus nativen Code heraus geworfen und innerhalb von Java behandelt werden können.

Allerdings wird von JNI nur die Sprache C gut unterstützt. Der Aufruf von C++-Funktionen ist zwar problemlos, aber eine Abbildung von Java-Klassen auf C++-Klassen oder von Java-Exceptions auf C++-Exceptions fehlt vollständig.

3.2.8.2 Vorteile von JNI

Durch die Nutzung von nativen Bibliotheken kann Code-Duplizierung verhindert und durch spezielle native Implementierungen die Performanz gesteigert werden. Außerdem ist es durch JNI möglich, besondere Funktionen der Plattform zu nutzen, für die es keine Abbildung in Java gibt. Ein Beispiel hierfür ist die Vergabe von Dateiberechtigungen aus Java heraus, dass plattformabhängig ist und bisher nicht von der Sun JVM unterstützt wird. Man kann nun ein natives Interface schreiben, das die jeweilge plattformabhängige Methode zur Vergabe dieser Rechte umsetzt.

3.2.8.3 JNI unter Android

Das Android-Framework basiert in weiten Teilen auf JNI. Viele der Funktionen der Android-API werden per JNI an native Bibliotheken weitergeleitet (vgl. Brady, 2008). Allerdings beinhaltet die Dalvik VM eine spezielle Implementierung von JNI. Gegenüber dem Standard-JNI in Sun Java-Umgebungen wurden einige Verbesserungen in die Dalvik VM eingebaut, um JNI möglichst effizient zu gestalten (vgl. Bornstein, 2008).

Dokumentation und Unterstützung

Bisher ist JNI unter Android von Google weder dokumentiert, noch wird es offiziell unterstützt. Es ist zur Zeit trotzdem nutzbar, wenn auch mit Einschränkungen. Darüber hinaus besteht die Gefahr, dass Google noch Änderungen an der Schnittstelle durchführt und damit bisherige Implementierungen nicht mehr mit einem zukünftigen SDK laufen. Eventuell auftretene Probleme bei der Benutzung von nativen Bibliotheken über JNI müssen daher von den Entwicklern selbst ohne Unterstützung und Hilfe von Google gelöst werden.

Für nativen Code steht nur eine unvollständige und undokumentierte libc zur Verfügung, sowie weitere native Bibliotheken, über die bisher nur wenig bekannt ist. Auf der Android-Plattform gibt es des Weiteren nicht den GNU-Linker, sondern einen speziell von Google entwickelten. Dieser Android-Linker verwendet „prelinking[9]“

[9] Mit „prelinking“ wird ein Verfahren bezeichnet, dass mit Hilfe von „relocating“ die Performanz-Probleme beim Laden von dynamischen Bibliotheken versucht zu umgehen.

von nativen Bibliotheken und hat eine im Vergleich zum GNU-Linker abweichende Anordnung der Text und Daten Bereiche der nativen Bibliotheken.[10]

Sicherheit bei der Benutzung von JNI

Durch die Benutzung von JNI verlässt man die Pfade des Frameworks und umgeht damit die Sicherheitsfunktionen der Dalvik VM. Allerdings sind Absicherungen wie die speziellen Rechte der Anwendungen nicht nur in der Dalvik VM implementiert, sondern auch im Kernel. Ein Beispiel hierfür ist die Berechtigung einer Anwendung, auf das Netzwerk zuzugreifen. Es ist die Berechtigung „android.permission.INTERNET“ notwendig, die auch innerhalb des Android-Kernel und für native Bibliotheken greift.

Neben den Berechtigungen der Anwendungen basiert das Sicherheitskonzept von Android (vgl. Kapitel 3.2.4) außerdem auf einer eindeutigen User- und Group-ID für jede Anwendung und die Abschottung der einzelnen Prozesse im Kernel. Daher wird auch der native Code nur mit diesen Benutzerrechten ausgeführt und gegenüber anderen Prozessen abgeschirmt. Außerdem werden sogar Bibliotheken im selben Prozess untereinander abgeschirmt und können nicht untereinander auf Funktionen zugreifen (vgl. Given, 2008). Somit wird das Konzept des Sandboxings auch für die nativen Bibliotheken verwendet.

Die größte Einschränkung ist die fehlende Absicherung gegenüber Speicherverletzungen durch nativen C-Code. Ein solcher Fehler kann nicht wie bei Bytecode innerhalb der Dalvik VM durch Exceptionhandling abgefangen werden, sondern beendet sofort den kompletten Prozess inklusive der Dalvik VM. Auch muss sich der Entwickler wie unter C und C++ üblich, vollständig um das Speichermanagment kümmern, wodurch es leicht zu schwerwiegenden Fehlern kommen kann.

Portabilität von nativen Bibliotheken

Durch die Nutzung von nativen Bibliotheken mittels JNI wird die Portabilität der Anwendungen erheblich eingeschränkt. Android-Anwendungen, die die Android-API verwenden, sind durch das Framework vollkommen unabhängig von der eingesetzten Hardware. Die nativen Bibliotheken müssen dagegen für jede Zielplattform kompiliert und eventuell angepasst werden. Darüber hinaus ist die Installation von Native Libraries nicht abschließend geklärt, da das Framework zur Zeit keine zusätzlichen nativen Bibliotheken unterstützt (vgl. Kapitel 5.4.1).

3.2.8.4 Alternativen zu JNI

Wie dargestellt ist die Benutzung von JNI nicht ohne Probleme möglich. Es gibt mehrere Lösungsmöglichkeiten, um JNI zu umgehen.

[10] Eine Anleitung zum linken von nativen Android-Bibliotheken ist unter `http://honeypod.blogspot.com/2007/12/shared-library-hello-world-for-android.html` zu finden.

Portierung von Bibliotheken nach Java und Anpassungen an die Android-API

Die auf den ersten Blick sicherste und beste Lösung wäre eine Portierung der gewünschten nativen Bibliothek in die Dalvik VM. Allerdings ist das nicht in allen Fällen möglich. So kann die Benutzung einer nativen Bibliothek sehr wohl Sinn machen, sei es aus Gründen der Performanz oder auch um eine gleiche Code-Basis zu erhalten. Änderungen in der nativen Bibliothek müssten nach einer Portierung auch in der Dalvik-Version durchgeführt werden. Ferner kommt besonders bei größeren Bibliotheken die Portierung einer komplett neuen Implementierung gleich.

Kommunikation mit nativem Code über BSD-Sockets

Um JNI als Schnittstelle zu umgehen aber den Code als native Bibliothek zu erhalten, gibt es noch die Möglichkeit die Kommunikation zwischen nativer Bibliothek und Dalvik VM-Anwendung über BSD-Sockets zu realisieren. Das erfordert allerdings neben speziellen Schnittstellen in beiden Teilen auch einen ständig laufenden nativen Daemon. Dieser müsste wiederum gestartet werden, wofür derzeit keine Lösung existiert. Man könnte entweder das System-Image ändern und ein Startscript einrichten, oder den Daemon wiederum per JNI starten.

Dieser Weg könnte bestehende Probleme des Library Loaders der Dalvik VM umgehen und unter Umständen wäre es hiermit möglich, auch komplexere C- oder C++-Bibliotheken unter Android zu benutzen.

3.2.8.5 Fazit

Trotz der ausgeführten Probleme bietet es sich an, die native VoIPFuture Library über JNI anzusprechen. Eine vollständige Portierung auf Java würde den Rahmen dieser Arbeit übersteigen und unter Umständen weitere Probleme mit sich bringen. Hier sind vor allem Code-Duplizierung und mangelnde Performanz zu nennen.

3.2.9 SIP- und RTP-Bibliotheken

Im Folgenden werden mögliche SIP-Bibliotheken vorgestellt. Neben der Unterstützung von SIP ist auch eine Media Library wichtig, welche die Audiodaten per RTP überträgt.

3.2.9.1 Entwicklung einer neuen SIP- und Media-Library

Neben der Verwendung der angeführten Bibliotheken bestände die Möglichkeit der Implementierung einer eigenen SIP- und Media-Library. Allerdings wäre der Aufwand erheblich höher, bis ein erster Erfolg, also eine funktionsfähige VoIP-Anwendung unter Android, erzielt würde.

3.2.9.2 pjsip

Der Einsatz von pjsip[11] ist eine mögliche Alternative. Diese ist vollständig in C geschrieben und es werden Wrapper für C++ sowie Python bereitgestellt. pjsip zeichnet sich durch die geringe Größe und eine hohe Performanz aus. Das ist vor allem für ein Embedded-Gerät wie ein Android-Telefon interessant. Außerdem wurde pjsip auf viele Architekturen portiert, darunter auch die Android-Zielarchitektur armv5b in Verbindung mit einem Linux. Es gibt gute Erfahrungen von VoIPFuture-Geschäftspartnern mit dieser Bibliothek, so dass wenige Probleme beim Einsatz zu erwarten sind.

pjsip beinhaltet eine umfangreiche Unterstützung für SIP. Darunter befinden sich die Authentifizierung und die Unterstützung von UDP sowie TCP inklusive TLS. Neben SIP bietet pjsip eine Media Library, die die Audiodaten encodieren kann und unter anderem einen anpassbaren Jitter Buffer beinhaltet. Für diese Arbeit und die damit für eine SIP- und RTP-Bibliothek verbundenen Anforderungen bietet pjsip einen ausreichenden Support.

Ein weiterer Vorteil von pjsip ist die zur Zeit immer noch aktive Entwicklung, wie an den Änderungen im Repository zu sehen ist[12]. Zu beachten ist, dass pjsip in der aktuellen Version unter der GNU General Public License[13] (GPL) steht und damit die Verwendung eingeschränkt ist.

Die Realisierung in C ist ein Problem unter Android, da pjsip dadurch nur per JNI zu benutzen ist. Diese Einschränkung würde einen zusätzlichen Aufwand und nicht voraus zu sehende Probleme bedeuten (vgl. Kapitel 3.2.8). Zusätzlich gibt es ungeklärte Abhängigkeiten von einer vollständigen libc und vermutlich anderen C-Bibliotheken. Da in Android keine vollständige libc enthalten ist und auch viele andere Bibliotheken fehlen, ist ein hoher Aufwand bei der Portierung zu befürchten.

3.2.9.3 jSip

jSip[14] zielt darauf ab, die SIP-Spezifikationen nach RFC2543 sowie einige Erweiterungen zu implementieren. Die Erweiterungen betreffen vor allem Instant Messaging. Die Ursprünge wurden von C++ auf Java portiert und dann weiter in Java ausgebaut. Allerdings konzentriert sich jSip vollständig auf die Signalisierung in SIP und besitzt keine Media-Library-Funktionalität. Außerdem hat es den Anschein als sei die Entwicklung dieser Bibliothek eingestellt worden, da die letzte öffentliche Aktivität von diesem Projekt auf das Jahr 2002 datiert.

3.2.9.4 JAIN-SIP

JAIN-SIP[15] implementiert laut deren Entwicklern die RFC 3261 vollständig. Allerdings sind die SDP-Elemente der Bibliothek JAIN-SDP noch nicht vollständig fertig gestellt. Im Oktober 2008 wurde jain erfolgreich ohne

[11] `http://www.pjsip.org/` (aufgerufen am 28.8.2008)
[12] `http://trac.pjsip.org/repos/timeline` (aufgerufen am 28.8.2008)
[13] Eine weit verbreitete und von der Open Source Initiative zertifizierte Open Source Lizenz der Free Software Foundation.
[14] `http://jsip.sourceforge.net/` (aufgerufen am 28.8.2008)
[15] `https://jain-sip.dev.java.net/` (aufgerufen am 24.10.2008)

weitere Anpassungen unter Android verwendet[16]. Wie auch bei jSip fehlt eine Media-Library mit einem RTP-Stack.

3.2.9.5 MjSip

MjSip[17] ist eine komplett in Java umgesetzte SIP-Bibliothek die auch einen einfachen RTP-Stack beinhaltet. MjSip entstand im Department of Information Engineering an der Universität von Parma und an der „DIE - University of Roma 'Tor Vergata'“ und steht unter der GPL in Version 2 oder höher.

Hughes Systique Corporation[18] hat diese Bibliothek auf das Android Framework portiert[19]. Allerdings wurde die Funktionalität mangels Audio-Unterstützung im Android-Emulator kaum getestet (vgl. Roychowdhury, 2008).

3.2.9.6 Fazit

Da die MjSip-Bibliothek alle nötigen Funktionen aufweist und darüber hinaus ohne weitere Änderungen unter Android lauffähig ist, wird für sie diese Arbeit verwendet. Von der zunächst präferierten pjsip-Bibliothek wurde vor allem wegen der problematischen Portierung Abstand genommen.

Leider ergaben im Rahmen dieser Arbeit Tests mit MjSip , dass die RFC3261 nicht vollständig implementiert ist. So fehlte die Unterstützung für den SIP *OPTIONS*-Request. Zudem sind auch einige der Konzepte dieser Bibliothek nicht vollständig implementiert. Unter Android ergeben sich mit MjSip des Weiteren erhebliche Probleme mit der Performanz, da sehr viele Objekte erzeugt werden. So dauerte das Erzeugen einer SIP-Antwort acht Sekunden.

3.2.10 VoIPFuture Library unter Android

Zur Messung der VoIP-Qualität mittels QoA und QoD wird die VoIPFuture Library verwendet. Diese muss im Zuge dieser Arbeit auf das Android-Framework portiert werden, um sie direkt in die VoIP-Anwendung integrieren zu können.

Laut den Entwicklern der VoIPFuture Library, wurde diese ausschließlich in ANSI-C implementiert. Zudem werden nur geringe Teile der *libc* und zusätzlich nur die *libm* verwendet. Aus diesem Grund sollten nur wenige Probleme bei der Portierung auftreten. Allerdings ist es für die Verwendung auf aktueller Android-Hardware und im Emulator notwendig, die VoIPFuture Library für einen ARM-Prozessor zu übersetzen. Vor dieser Bachelorarbeit wurde die Library ausschließlich unter der x86-Architektur getestet und verwendet.

[16] `http://jeanderuelle.blogspot.com/2008/10/jain-sip-is-working-on-top-of-android.html` (aufgerufen am 24.10.2008)
[17] `http://www.mjsip.org/` (aufgerufen am 4.6.2008)
[18] `http://www.hsc.com/` (aufgerufen am 4.6.2008)
[19] `http://www.hsc.com/resourceCenter/whitepapers.aspx` (aufgerufen am 4.6.2008)

4 Entwurf der Android-VoIP-Anwendung

In diesem Kapitel wird die zu entwickelne Android-VoIP-Anwendung entworfen. Dafür werden die anfallenden Aktivitäten skizziert, die Anwendungs-Architektur erläutert und die einzelnen Komponenten dieser Anwendung vorgestellt.

4.1 Aktivitäten

In der Android-VoIP-Anwendung gibt es relativ wenige und außerdem wenig komplexe Aktivitäten. Sie behandeln ausschließlich die Ablaufsteuerung eines Anrufs.

Der Export der Analysedaten wird als weitere Aktivität in dieser Arbeit nicht behandelt, da es für die Untersuchung mittels eines Emulators nicht notwendig ist. Im Fall des Einsatzes der VoIP-Anwendung auf einem Telefon außerhalb einer Laborumgebung müsste dafür eine Lösung geschaffen werden.

4.1.1 Eingehender Anruf

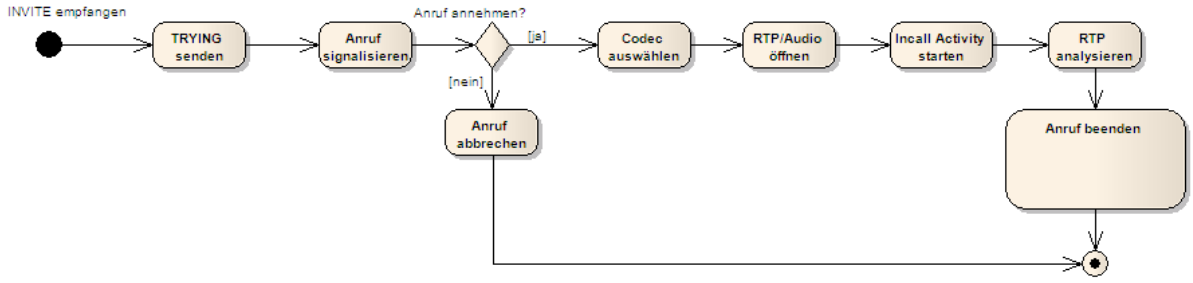


Abbildung 4.1: Aktivitätsdiagramm eines eingehenden Anrufs

Bei einem eingehenden Anruf (vgl. Abbildung 4.1) wird der Gegenstelle nach dem Empfang der Einladung zuerst der Empfang mittels eines `TRYING` über SIP signalisiert. Danach wird der Anwender durch einen Dialog, einer Vibration und einem Klingelton informiert. Falls die Benachrichtigung des Anwenders erfolgreich ist, wird der Gegenstelle mit einem `RINGING` die Benachrichtigung signalisiert und der Anwender entscheidet über die Annahme oder Ablehnung des Anrufs. Bei einem Fehler wie auch bei Ablehnung des Anrufs durch den Anwender, sendet das System ein `CANCEL` und der Anruf wird vollständig beendet.

Bei Annahme des Anrufs wird dieser weiter aufgebaut, indem ein passender Codec ausgewählt wird, die RTP-Verbindungen in beide Richtungen geöffnet werden und die Audio-Ein- und -Ausgabe gestartet wird. Abschließend wird eine Activity zur Anzeige der Anruf-Informationen geöffnet, die RTP-Pakete analysiert und die Diagnose-Werte in der Activity angezeigt. Der Anruf wird beendet, wie in Kapitel 4.1.3 beschrieben.

4.1.2 Ausgehender Anruf

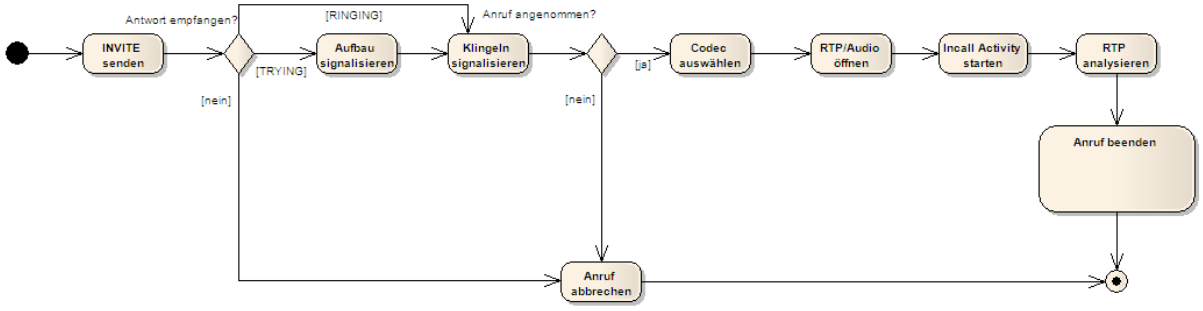


Abbildung 4.2: Aktivitätsdiagramm eines ausgehenden Anrufs

Bei einem vom Anwender gewünschten ausgehenden Anruf (vgl. Abbildung 4.2) wird zuerst eine SIP-Verbindung zur Gegenstelle aufgebaut. Sollte das erfolglos sein oder keine Antwort eingehen, wird der Anruf sofort abgebrochen und dieser Vorgang dem Anwender mitgeteilt.

Bei Empfang einer `TRYING`-Nachricht wird der Anwender über den Aufbau des Anrufs informiert und auf den Empfang der `RINGING`-Mitteilung gewartet. Falls die Gegenstelle diesen Schritt überspringt und sofort ein `RINGING` sendet, wird die Anzeige des Anruf-Aufbaus übersprungen. Wenn der Anwender diesen Vorgang abbricht oder die `RINGING`-Mitteilung ausbleibt, wird der Anruf abgebrochen und der Abbruch der Gegenstelle durch ein `CANCEL` signalisiert.

Danach wird dem Anwender das Klingeln signalisiert und auf die Bestätigung durch die Gegenstelle gewartet. Bei Empfang der Bestätigung werden wie bei einem eingehenden Anruf die RTP-Verbindungen in beide Richtungen geöffnet, die Audio-Ein- sowie Ausgabe begonnen und die zusätzliche Activity gestartet. Während des Anrufs werden die RTP-Pakete mittels der VoIPFuture Library analysiert und am Ende der Anruf beendet.

4.1.3 Beenden eines Anrufs

Das Beenden eines Anrufs durch die Gegenstelle unterscheidet sich hinsichtlich der Aktivitäten nur gering vom Beenden durch den Anwender der Android-VoIP-Anwendung (vgl. Abbildung 4.3). Nur die Signalisierung des `BYE`-Kommandos über SIP ist dabei unterschiedlich. Entweder wird das Kommando empfangen und das Android-System muss darauf reagieren oder der Anwender beendet einen Anruf aktiv und das System muss der Gegenstelle mittels `BYE` das Ende der Verbindung signalisieren.

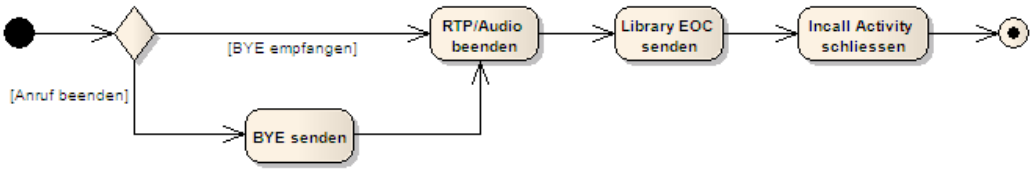


Abbildung 4.3: Aktivitätsdiagramm über das Beenden eines Anrufs

Danach werden die RTP-Verbindungen sowie die Audio-Ein- und -Ausgabe beendet, der VoIPFuture Library das Ende mitgeteilt und die Analysedaten in eine Datei exportiert. Am Ende muss darüber hinaus die Activity, die während des Anrufs angezeigt wurde, geschlossen werden und die Haupt-Activity erneut angezeigt werden.

4.2 Architektur

Dieses Kapitel beschreibt die Architektur der VoIP-Anwendung für die Android-Plattform. Sie wird anhand der Analyse (vgl. Kapitel 3) entworfen und dient der Umsetzung der VoIP-Qualitätanalyse im Rahmen dieser Arbeit.

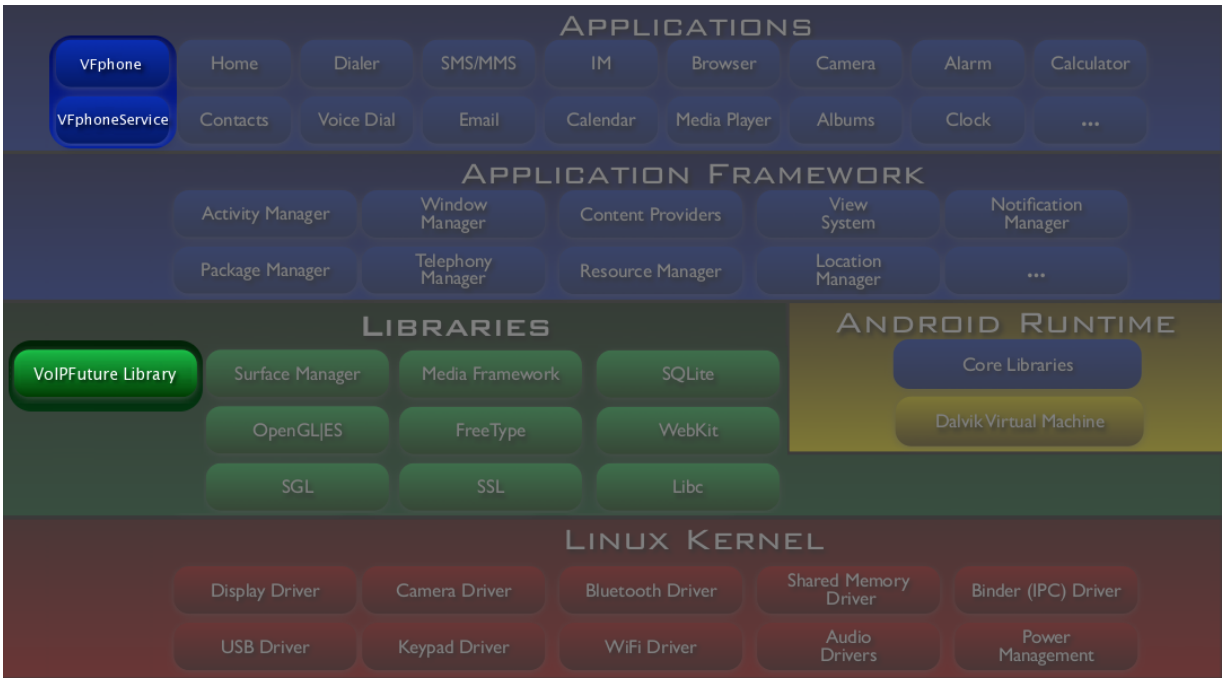


Abbildung 4.4: Die Bestandteile der Architektur, eingeordnet in das Android-Framework (vgl. Google, 2008a)

Um die in Kapitel 3.1.3.2 festgestellte Anforderung nach einem jederzeit reagierenden SIP- und RTP-Stack zu erfüllen, ist es unter Android notwendig diesen Stack in einen Service zu kapseln. Durch den zu jeder Zeit

im Hintergrund laufenden Service kann auf neue VoIP-Anfragen reagiert werden und eine bestehende VoIP-Verbindung weitergeführt werden auch wenn die VoIP-Anwendung nicht im Vordergrund läuft. Allerdings kann ein Service unter Android keine Benutzerschnittstelle enthalten, so dass die Activity *VFphone* notwendig ist. Damit wird die VoIP-Anwendung in drei Bestandteile gegliedert: die Activity *VFphone*, den Service *VFphone-Service* und die native VoIPFuture Library, die vom *VFphoneService* gesteuert wird. Die VoIPFuture Library befindet sich als native Bibliothek nach diesem Entwurf in der mittleren Schicht der Android-Architektur (vgl. Kapitel 3.2.2). Die Activity und der Service dagegen liegen in der obersten der drei Schichten (vgl. Abbildung 4.4).

Die Anwendungslogik im Service ist in diesem Konzept strikt von der Oberfläche und Benutzerschnittstelle in der Activity getrennt. Dadurch wird eine lose Kopplung und damit gute Wartbarkeit und Erweiterbarkeit erreicht (vgl. Kapitel 3.1.2).

Unter Android ist es nicht möglich, Bibliotheken ohne dazugehörende Anwendung zu installieren. Stattdessen müssen die Bibliotheken in einen Service oder eine Activity gekapselt werden und durch diesen anderen Anwendungen zur Verfügung gestellt werden. Der „VFphoneService“ enthält deshalb neben der Anwendungslogik mit der SIP- und RTP-Bibliothek MjSip auch die native Bibliothek für QoA und QoD, mit der über JNI kommuniziert wird. Die Activity *VFphone* beinhaltet die Benutzerschnittstellen und steuert den Service mittels AIDL (vgl. Abbildung 4.5).

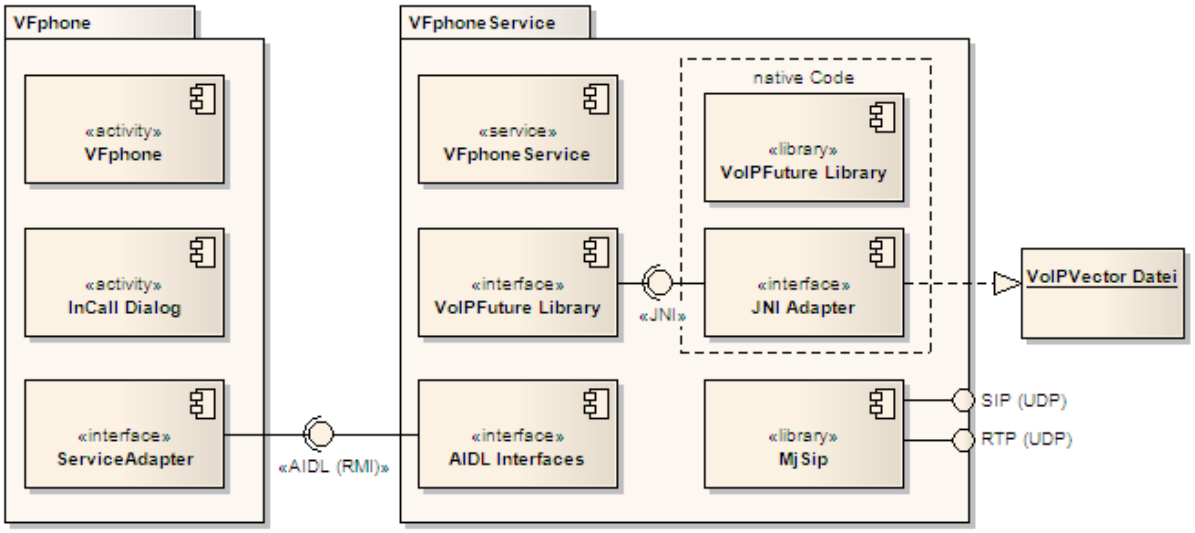


Abbildung 4.5: Komponenten- und Kommunikations-Diagramm

Für eine Erweiterung ist es möglich, die native VoIPFuture Library mit der JNI-Schnittstelle in einen dedizierten Service auszulagern, um weiteren Anwendungen die Funktionalität zur Verfügung zu stellen. Im Rahmen dieser Arbeit wurde darauf allerdings aufgrund der zusätzlichen Komplexität und noch bestehenden Einschränkungen[1] bei der Entwicklung mit Hilfe von AIDL verzichtet. Außerdem würde ein Service unun-

[1] Es bestanden zum Zeitpunkt dieser Arbeit Probleme im Eclipse Android Plugin mit Objekten die per AIDL ausgetauscht werden.

terbrochen laufen und im Speicher gehalten, obwohl er lediglich die Funktionalität einer Bibliothek mit sich bringt.

4.3 Komponenten

In Kapitel 4.2 wurde für die Android-VoIP-Anwendung eine Architektur mit zwei voneinander getrennten Anwendungsteilen eingeführt: der Service *VFphoneService* und die Activity *VFphone*. In diesem Kapitel werden diese Anwendungsteile jeweils beleuchtet und deren genauer Aufbau vorgestellt.

4.3.1 Die VFphoneService-Anwendung

Dieser Service beinhaltet die Anwendungslogik, Integration für Android und dient der Steuerung der verwendeten Bibliotheken MjSip sowie der VoIPFuture Library. Er wird beim Systemstart mittels einer Activity gestartet und läuft danach ununterbrochen um jederzeit eingehende Anrufe signalisieren zu können.

4.3.1.1 Android-Integration und Interaktion (VFphoneService)

Die Komponente für die Android-Integration und -Interaktion verbindet die weiteren Komponenten und integriert diese in die Android-Umgebung.

Im einzelnen besteht sie aus den folgenden Klassen (vgl. Abbildung 4.6):

VFphoneService Diese Klasse initialisiert den Service und die native VoIPFuture Library und startet den SIP-Service um eingehende Anfragen beantworten zu können.

AudioService Der AudioService bietet eine Schnittstelle zum Start sowie Stopp der Audio-Ein- und -Ausgabe, Klingelton- und Vibrations-Steuerung.

SipService Das Interface ISipServiceInterface wird durch diese Klasse implementiert und bietet die generelle Anrufsteuerung, die die Audio- und SIP-Steuerung vereint. Zudem instantiiert es die *Call*-Klasse der MjSip-Bibliothek und registriert in dieser den VFCallListener.

VFCallListener Diese Klasse implementiert einen Listener des MjSip-Stacks, mit Hilfe dessen auf SIP-Ereignisse reagiert werden kann.

BootCompletedIntentReceiver Diese Klasse implementiert einen *BroadcastReceiver*, um beim Systemstart den VFphoneService zu starten. Diese besondere Activity ist notwendig, da ein Service nicht gleichzeitig als *BroadcastReceiver* arbeiten kann.

VFServiceActivity Diese Activity ist ausschließlich eine Hilfe im Entwicklungsprozess, um vom Android-Eclipse-Plugin den Service erneut starten zu können. Das ist zur Zeit nur für Activities möglich.

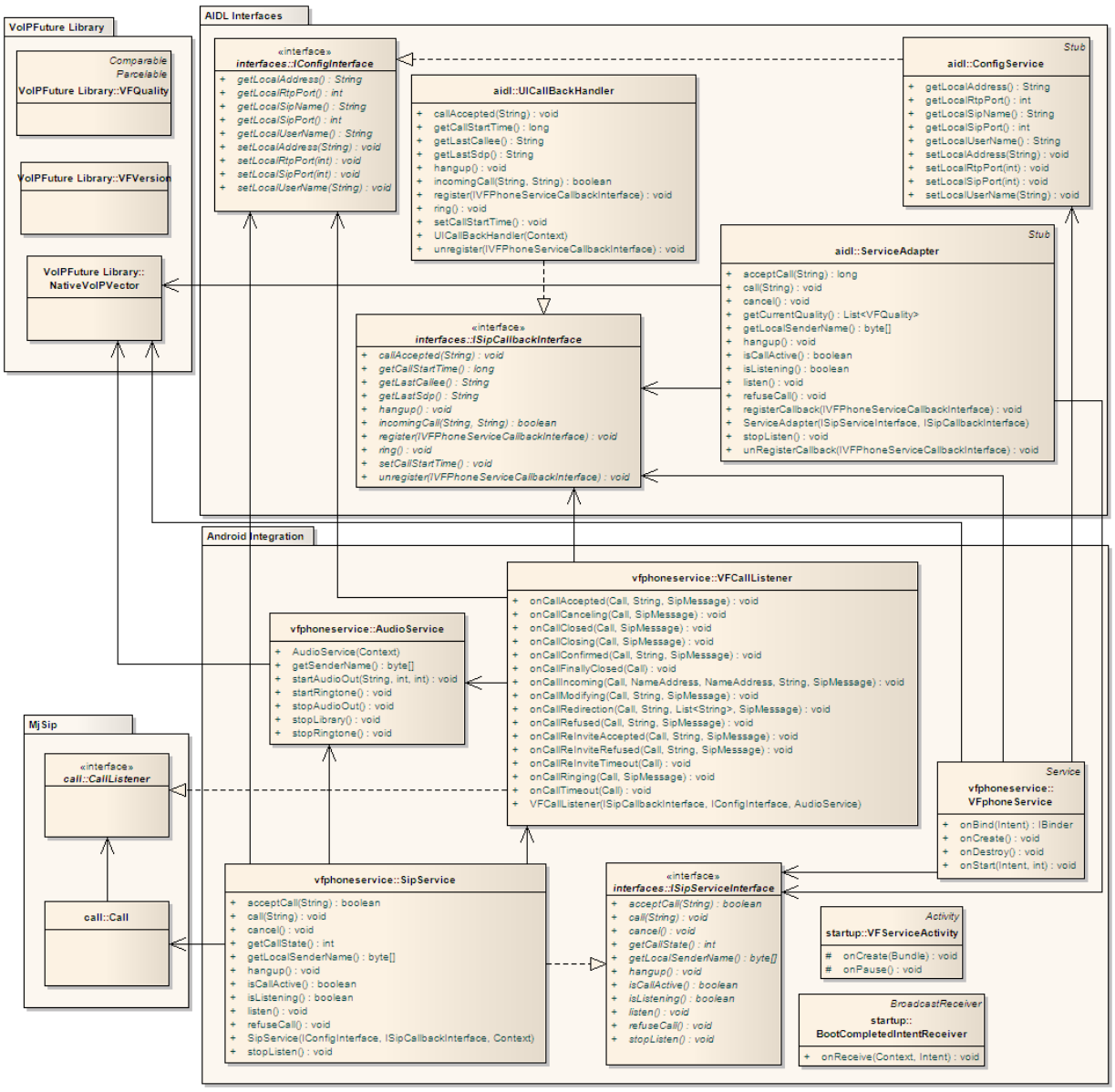


Abbildung 4.6: Klassendiagramm des VFphoneService

4.3.1.2 AIDL-Interfaces

Die AIDL-Interfaces haben eine hohe Kopplung mit der Komponente für die Android-Integration. Eine Auflösung der Kopplung würde die Komplexität stark erhöhen, so dass im Rahmen dieser prototypischen Entwicklung darauf verzichtet wurde.

In dieser Komponente werden die Schnittstellen für die weiteren Android-Anwendungen zusammengefasst. Über diese ist es möglich, den Service zu steuern und die MjSip-Bibliothek, die SIP-Konfiguration und auch die VoIPFuture Library zu verwenden. Auf Meldungen des Services kann mittels eines „Callback“ reagiert

werden, indem diese Schnittstellen in anderen, dafür registrierten Anwendungen, aufgerufen werden. Ein Beispiel für solch eine Meldung ist die Nachricht über einen eingehenden Anruf.

In den AIDL-Interfaces werden die folgenden Klassen zusammengefasst (vgl. Abbildung 4.6):

ServiceAdapter Diese Klasse implementiert die AIDL-Schnittstelle „IVFPhoneServiceInterface“ und stellt den Activities die VoIP-Funktionalität des Services bereit.

UICallBackHandler Der UICallBackHandler implementiert die AIDL-Schnittstelle „ISipCallbackInterface“ und schickt den registrierten Activities mit Hilfe eines *Handlers* (vgl. Kapitel 3.2.7) und einer *RemoteCallbackList* Nachrichten über den Zustand der VoIP-Verbindungen.

ConfigService Diese Klasse implementiert die AIDL-Schnittstelle „IConfigInterface“ und bildet die Schnittstelle zur Konfiguration des Services.

4.3.1.3 MjSip

Die Android-Version von MjSip wurde in den *VFphoneService* integriert. Für die Verwendung von MjSip ist es notwendig, das Interface *CallListener* zu implementieren und bei der Verwendung der *Call*-Klasse in dieser zu registrieren (vgl. Kapitel 4.3.1.1). Darüber hinaus ist es vorgesehen einen Aufruf der VoIPFuture Library in den RTP-Sender und -Empfänger zu implementieren, um die Header aller RTP-Pakete zu analysieren.

4.3.1.4 VoIPFuture Library

Die Portierung der VoIPFuture Library besteht aus drei Bestandteilen und ähnelt dem Adapter-Entwurfsmuster (vgl. Abbildung 4.7). Sie beinhaltet die auf die ARM-Architektur portierte VoIPFuture Library, den in C geschriebenen nativen JNI-Adapter und der in Java geschriebenen Schnittstelle für den Zugriff über JNI.

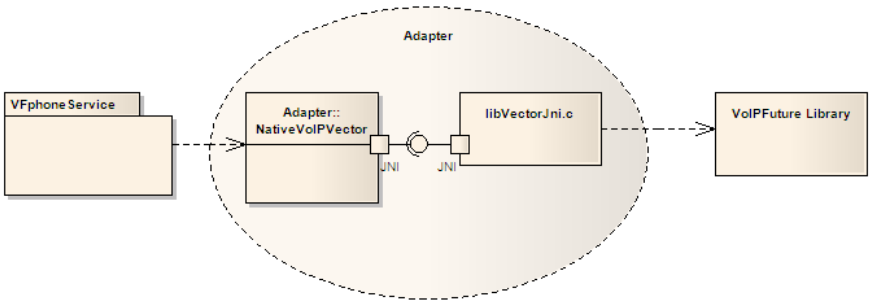


Abbildung 4.7: Das Adapter-Entwurfsmuster, angewendet auf VFphoneService

Der native JNI-Adapter und die Java-Schnittstelle bilden zusammen den Adapter des Entwurfsmusters. Die Auftrennung des Adapters auf zwei Teile ist aufgrund der unterschiedlichen Umgebungen, zum einen der Dalvik VM mit in Java geschriebenem Quellcode und zum anderen der nativen Umgebung mit C-Quellcode, notwendig.

Java Schnittstelle

Die Java-Schnittstelle der VoIPFuture Library kapselt alle Zugriffe auf die native Bibliothek und übersetzt die erhaltenen Daten in Java-*Enum*-Typen. Dabei werden für den Transport über JNI die vom Android-*Parcelable*-Interface abgeleiteten Klassen *VFQuality* sowie *VFVersion* bereitgestellt. Das Interface *Parcelable* wird dabei für die Kommunikation über AIDL zwischen dem Service und der Activity benötigt.

Den Kern der Schnittstelle bildet die Klasse *NativeVoIPVector*, die den Zugriff auf die Bibliothek über „static"-Methoden erlaubt.

Außerdem fällt diesem Teil noch die Aufgabe der Installation der nativen Bibliothek auf der Android-Plattform zu. In der aktuellen Version der Android-API wird das noch nicht unterstützt, so dass hierfür eine eigene Lösung notwendig ist (vgl. Kapitel 5.4.1).

Nativer Adapter

Der native Teil des Adapters ist, wie auch die VoIPFuture Library, vollständig in ANSI-C geschrieben und kapselt den Zugriff über JNI auf nativer Ebene. Dabei werden die Datentypen der VoIPFuture Library in die in Kapitel 4.3.1.4 beschriebenen Klassen *VFQuality* und *VFVersion* konvertiert. Bei Fehlern werden entsprechende Exceptions aus den erhaltenen Fehlercodes erzeugt und über JNI geworfen.

4.3.2 Die VFphone-Anwendung

Diese Anwendung beinhaltet die Oberfläche für das VoIP-System und die Anbindung an den VFphoneService. Sie lässt sich in die drei Komponenten *VFphone Activity*, *Incall Activity* und *Service Adapter* gliedern.

Für die Verwendung des VoIP-Systems von anderen Android-Anwendungen ist es möglich, die *VFphone Activity* durch einen Intent mit dem „sip://" URL-Schema zu starten. Für diesen Zweck beinhaltet die *VFphone Activity* einen Intent-Filter.

Die Verbindung zwischen der *VFphone Activity* und dem *Service Adapter* ist über das Beobachter (Observer)-Entwurfsmuster gelöst. Dafür wurden die Observer-Klassen aus `java.util.*` durch entsprechende Interfaces ersetzt und die Klasse *ObservableMulticaster* für den Aufruf des jeweiligen Observers eingeführt. Die Java-Klassen sind für diesen Fall nicht ohne weiteres zu verwenden, da in Java keine Mehrfachvererbung möglich ist und die *VFphone Activity* zwingend von der Klasse *Activity* erben muss (vgl. Abbildung 4.8).

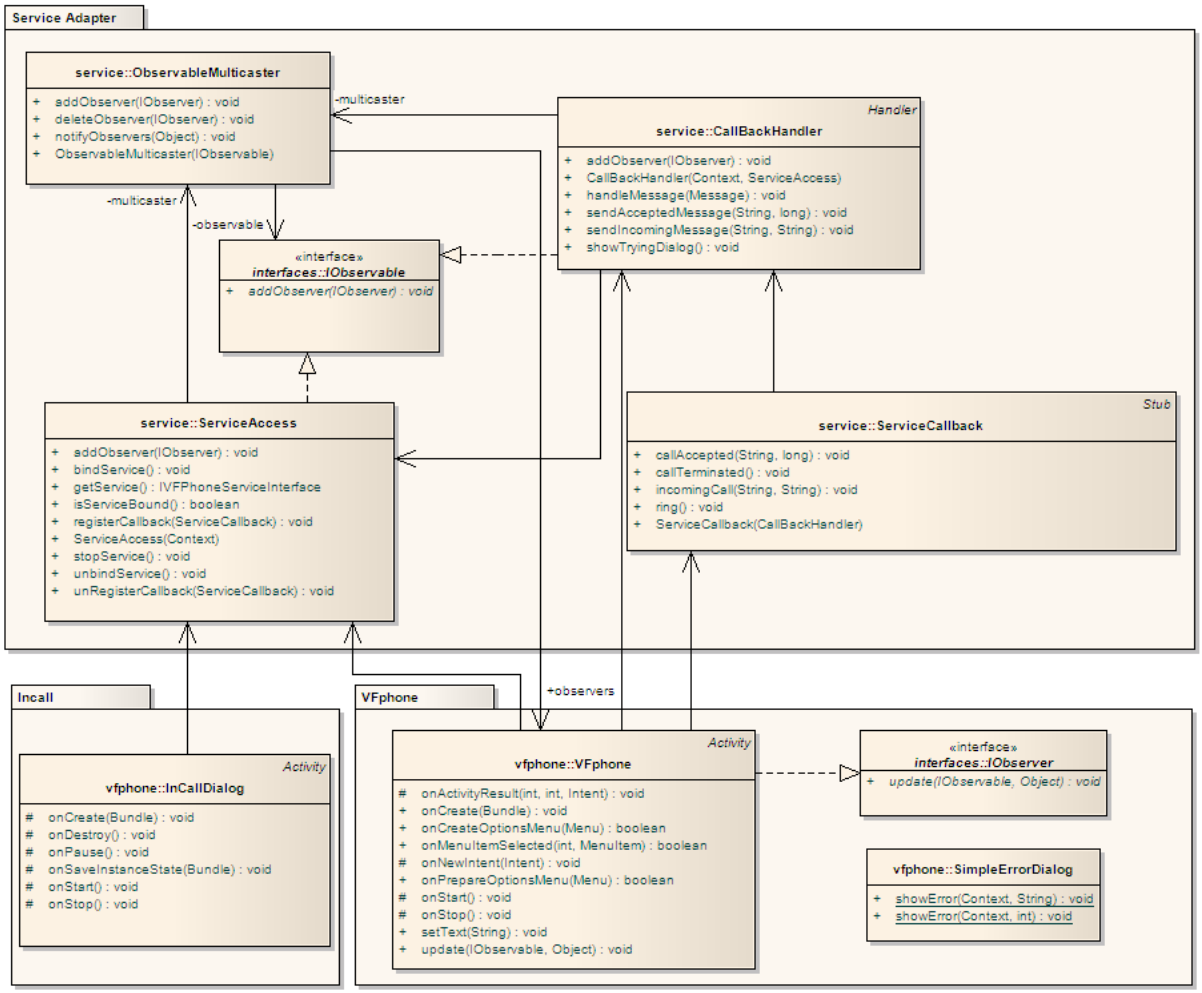


Abbildung 4.8: Klassendiagramm der VFphone Activity

4.3.2.1 VFphone Activity

Diese Komponente besteht vor allem aus der Activity *VFphone*, welche die Oberfläche zur Steuerung der Anrufe und des Services beinhaltet. Diese Activity bildet die zentrale Komponente, über die die VFphone-Anwendung gestartet wird. Sie entspricht im Observer-Entwurfsmuster dem Observer und wird von den Klassen des *Service Adapter* über neue Ereignisse benachrichtigt.

Außerdem lässt sich noch der *SimpleErrorDialog* zur der *VFphone Activity* zählen. Er kapselt den Aufruf eines *android.os.Dialog* für die Anzeige kurzer Meldungen.

4.3.2.2 Incall Activity

Die *Incall Activity* besteht aus der *InCallDialog*-Activity für die Anzeige der Informationen während eines Anrufs. Neben den Informationen zum Anrufer und zur Anrufdauer, visualisiert sie auch die Qualität der Verbindung über Ampelfarben.

4.3.2.3 Service Adapter

Diese Komponente stellt die Kommunikation mit dem Service her und beinhaltet im Einzelnen die folgenden Klassen und Funktionen (vgl. Abbildung 4.8):

ServiceAccess Diese Klasse bindet die *VFphone Activity* über AIDL-IPC an den Service. Dabei ist es möglich diesen sowohl zu starten als auch zu stoppen. Sie verwaltet die Verbindung zum Service durch die *ServiceConnection* und erlaubt damit den Aufruf dessen Methoden. Außerdem registriert sie die Klasse *ServiceCallback* beim Service und implementiert das „Observable“ aus dem Observer-Entwurfsmuster.

ServiceCallback Der „ServiceCallback“ implementiert das Callback-Interface des *VFphoneService* um die *VFphone Activity* mit Hilfe des *CallBackHandler* zu benachrichtigen.

CallBackHandler Diese Klasse implementiert einen *Handler* (vgl. Kapitel 3.2.7), der im Thread der *VFphone Activity* läuft und diese als „Observable“ im Observer-Entwurfsmuster benachrichtigt.

5 Realisierung der Android-VoIP-Anwendung

In diesem Kapitel wird zuerst der in dieser Arbeit entwickelte Prototyp der Android-VoIP-Anwendung vorgestellt. Danach werden besondere Aspekte der Entwicklung hervorgehoben und geschildert. Hierzu gehören die Optimierungen der MjSip-Bibliothek, der AIDL-Implementierung zwischen *VFphone* und *VFphoneService* und die Implementierung der JNI-Schnittstelle unter Java und C. Innerhalb des Abschnitts über die JNI-Lösung wird zudem die im Verlauf dieser Arbeit entwickelte Lösung zur Installation einer nativen Bibliothek unter Android vorgestellt, für die keine *root*-Rechte notwendig sind.

5.1 Prototyp der Android-VoIP-Anwendung

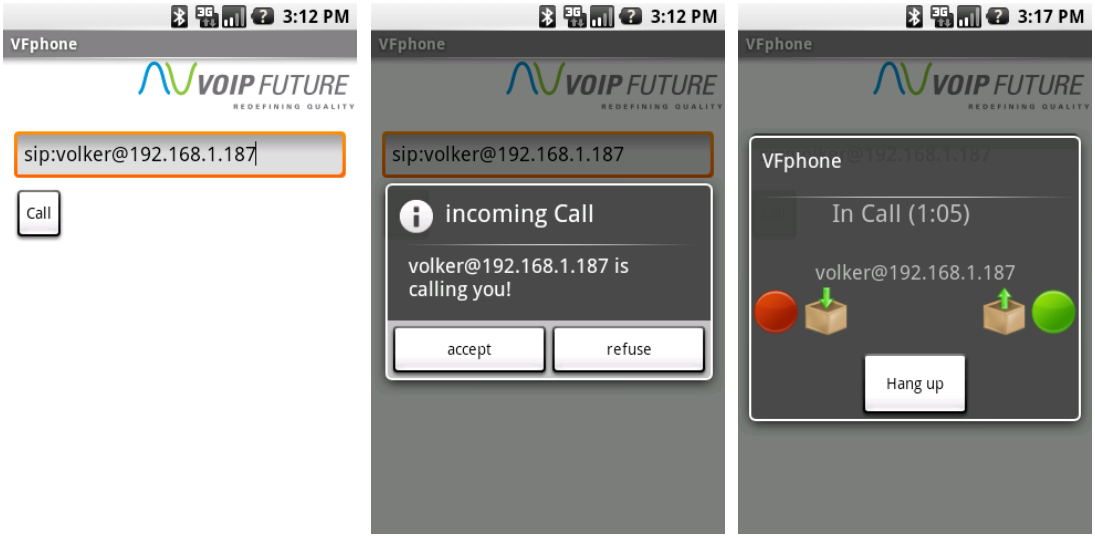
In dieser Arbeit findet aus mehreren Gründen lediglich eine prototypische Entwicklung statt. Das hat mehrere Gründe. Neben der begrenzten Zeit liegt die Ursache dafür auch in der unsicheren und nicht klar definierten Situation bezüglich Android zu Beginn der Arbeit im Juni 2008.

Es war lange Zeit nicht klar, welchen Umfang das finale SDK für Android haben würde. Aber auch die fehlende Hardware für einen Test der Anwendung und der Funktionen ließ die Entwicklung ausschließlich für eine Laborumgebung zu. Der Fokus lag vor allem auf der Durchführung von VoIP-Qualitätstests unter Android und damit auch auf der Portierung der VoIPFuture Library auf einen ARM-Prozessor und das Android-Framework.

5.1.1 Beschreibung des Prototypen

In der Android-VoIP-Anwendung *VFphone* (vgl. Abbildung 5.1a) ist es möglich, Anrufe zu beliebigen Endpunkten zu starten, eingehende Anrufe aufzubauen und auch abzulehnen (vgl. Abbildung 5.1b).

Während einer Verbindung werden die RTP-Pakete von der VoIPFuture Library analysiert und die Diagnose über die vier Farben *grün* (gut), *gelb* (Warnung), *rot* (schlecht) und *weiß* (undefiniert) für beide Richtungen der Verbindung getrennt angezeigt (vgl. Abbildung 5.1c).



(a) Hauptansicht (b) eingehender Anruf (c) während einer Verbindung

Abbildung 5.1: Screenshots der VoIP-Anwendung im Emulator

5.1.2 Einschränkungen des Prototypen

Der Prototyp hat mehrere Einschränkungen, weshalb er nicht als vollständiger SIP-User-Agent bezeichnet werden kann. Allerdings wurde bei der Entwicklung vorgesehen, die im folgenden beschriebenen Einschränkungen über Erweiterungen zu beheben.

5.1.2.1 Nur direkte Ende-zu-Ende-Verbindungen

Die VoIP-Anwendung unterstützt ausschließlich direkte Ende-zu-Ende-Verbindungen. Es ist daher nicht möglich, über eine Registrierung bei einem SIP-Proxy Verbindungen herzustellen. Allerdings unterstützt die verwendete MjSip-Bibliothek grundsätzlich Verbindungen über einen SIP-Proxy, es fehlt lediglich eine Umsetzung in der VoIP-Anwendung.

5.1.2.2 Keine detaillierte Diagnose

Die VoIP-Anwendung unterstützt keine detaillierte Diagnose der von der VoIPFuture Library gesammelten Analyse-Vektoren. Diese Vektoren werden allerdings als VAD-Dateien auf dem Android-Telefon gespeichert und können zur Diagnose manuell auf einen PC kopiert werden, um sie dort in den VoIPFuture Manager zu importieren oder mit dem VoIPFuture VectorViewer zu betrachten.

Eine Lösung für die fehlende detaillierte Diagnose kann der Versand der Analyse-Vektoren über RTCP oder über eine weitere Netzwerkverbindung sein. Hier besteht aktuell noch weiterer Forschungsbedarf, da es für diese Lösung zur Zeit kein Beispiel gibt.

5.1.2.3 Einschränkungen des Emulators

Der Emulator bietet einem Entwickler wie in Kapitel 3.2.1.1 beschrieben eine gute Umgebung zum Testen der Anwendungen. Es ist jedoch nicht alles möglich. So funktioniert zur Zeit der Entwicklung im Rahmen dieser Arbeit das Mikrofon des Emulators nicht. Daher werden die Audiodaten aus einer Datei gelesen und beim Empfang auch in eine Datei geschrieben. Dieses Vorgehen erleichtert darüber hinaus auch den Test der Anwendung während der Entwicklung, da automatisch Audiodaten versendet werden (vgl. Kapitel 3.1.3.3). Außerdem liegen die Audiodaten in dieser Datei bereits im richtigen Audioformat vor, so dass eine Umwandlung durch einen Codec während der Laufzeit entfällt.

5.1.2.4 Eingeschränkte Konfiguration

Der Prototyp der VoIP-Anwendung bietet keinerlei dynamische Konfiguration während der Laufzeit. Alle Parameter müssen direkt im Quellcode des VFphoneService eingestellt werden. Allerdings ist die AIDL-Schnittstelle für die Konfiguration implementiert. Daher ist es möglich, die Konfiguration durch die Anwendungen zu realisieren.

5.1.2.5 Einschränkungen durch native Bestandteile

Native Bibliotheken werden von Google nicht unterstützt. In jeder neuen Version des ADK können Änderungen an der JNI-API und auch an der libc (Bionic) durchgeführt werden. Es gibt des Weiteren keine gesicherte Portabilität auf andere Umgebungen als den Android-Emulator. Eine Lauffähigkeit auf dem HTC G1 und anderer Hardware kann nur vermutet werden.

5.2 Optimierung von MjSip für Android

Anhand der Portierung von MjSip lässt sich beispielhaft die Optimierung und Anpassung einer Java-Anwendung auf das Android-Framework beschreiben. Trotz der bereits erfolgten Portierung für Android durch die Hughes Systique Corporation[1] bestanden erhebliche Probleme bei der Verwendung von MjSip.

Es fehlte die Unterstützung für die `OPTIONS`-Nachricht, die in RFC 3261[2] optional gefordert wird und für Tests mit dem „sipsak“-Werkzeug[3] notwendig ist. Diese Funktion wurde im Rahmen dieser Arbeit implementiert.

Des Weiteren war es nicht möglich auf die Log-Ausgaben der MjSip-Bibliothek zuzugreifen. Unter Android sind Ausgaben nach `STDOUT` und `STDERR` nicht zu erreichen. Diese Ausgaben werden vollständig durch die

[1] `http://www.hsc.com/` (aufgerufen am 4.6.2008)
[2] `http://www.ietf.org/rfc/rfc3261.txt` (aufgerufen am 25.11.2008)
[3] `http://sipsak.org/` (aufgerufen am 25.11.2008)

```
/* Tag für die Log Ausgabe setzen */
final static String LOG_TAG = "TEST";

try {
   throw new Exception();
} catch (Throwable t) {
   Log.w(LOG_TAG, t);
}
```

Listing 5.1: Beispielhafte Ausgabe eines Stacktrace einer Exception über das Logcat-Framework

Oberfläche versteckt und sind auch nicht über die Debug-Werkzeuge zugänglich. Daher wurden alle bisherigen Ausgaben mit `System.out.print()` und `Exception.printStackTrace())` durch einen Aufruf des Android spezifischen „logcat“-Frameworks ersetzt (vgl. Listing 5.1).

Vor allem bestanden Probleme mit der Performanz der MjSip-Bibliothek bei der Verwendung unter Android. Eine Lösung für eine bessere Performanz soll im Folgenden anhand der Klasse BaseMessage geschildert werden.

5.2.1 Optimierung der Klasse BaseMessage

Nach der Implementierung der `OPTIONS`-Nachricht vergingen zwischen dem Senden der Anfrage und dem Empfangen der Antwort über sieben Sekunden. Nach der Optimierung der Klasse `org.zoolu.sip.message.BaseMessage` konnte eine Verringerung der Antwortzeit auf unter 400ms erzielt werden.

```
private HashMap<String, String> _headerMap = new HashMap<String, String>();

public String headersToString() {
   StringBuilder result = new StringBuilder();
   for (Entry<String, String> entry : _headerMap.entrySet()) {
      result.append(entry.getKey());
      result.append(':');
      result.append(entry.getValue());
      result.append("\r\n");
   }
   return result.toString();
}
```

Listing 5.2: Ausgabe der Map des Message-Header in einen String (aus BaseMessage.java)

Dabei stützt sich die hier geschilderte und im Rahmen dieser Arbeit durchgeführte Optimierung vor allem auf zwei Maßnahmen:

- die erzeugten Objekte für jede empfangene und versendete Nachricht stark zu verringern

- und den Header einer Nachricht nur noch beim Empfang zu parsen.

In der MjSip-Version der Hughes Systique Corporation wurde für jedes Element eines SIP-Headers ein eigenes Objekt erzeugt und nicht im Nachricht-Objekt gespeichert. Dieses Vorgehen führte zu einer Vielzahl von erzeugten komplexen Objekten, die außerdem noch bei jedem Zugriff kopiert und erneut erzeugt wurden, und unbenutzte Eigenschaften voneinander erbten. Bei der im Rahmen dieser Arbeit durchgeführten Optimierung, werden alle Einträge im SIP-Header ausschließlich als *String* in einer *HashMap* gespeichert (vgl. Listing 5.2). In den meisten Fällen kann auf eine Konvertierung in andere Typen verzichtet und dadurch Speicher und Rechenzeit gespart werden.

```
1  /** Parses A Header String into the Map */
2  private void splitHeaderToList(String msg) {
3     String[] arr = AndroidStringUtils.linePattern.split(msg);
4     _headerMap.clear();
5     String[] line;
6     int size = arr.length;
7     for (int i = 0; i < size; i++) {
8        line = AndroidStringUtils.columnPattern.split(arr[i], 2);
9        if (line.length <= 1) continue;
10       addHeader(line[0].trim(), line[1].trim());
11    }
12 }
13
14 /** Gets the first Header of specified name (Returns null if no Header is found) */
15 public String getHeader(String hname) {
16    if (_headerMap == null) return null;
17    return _headerMap.get(hname);
18 }
```

Listing 5.3: Parsen eines SIP-Headers und Suche nach einem Header-Feld in der Map (aus BaseMessage.java)

Darüber hinaus wird eine empfangene SIP-Nachricht in der neu entwickelten und optimierten Version nur noch ein Mal untersucht und sofort alle einzelnen Header-Einträge in der *HashMap* gespeichert (vgl. Listing 5.3). Dieses Vorgehen spart ab dem zweiten Zugriff auf ein beliebiges Element des Headers das aufwändige Parsing. In der ursprünglichen Version von MjSip wurde die Nachricht vollständig in einer *String*-Variablen gespeichert und bei *jedem* Zugriff auf ein Header-Element erneut analysiert und geparst.

5.2.2 Verwendung eines Timers für den RTP-Sender

Um beim Versand der RTP-Pakete eine möglichst hohe Isochronität zu gewährleisten, wurde die Schleife mit einem `Thread.sleep()` durch einen *Timer* ersetzt. Der *HighPriorityTimer* startet in festgelegten Intervallen den *HighPriorityTimerTask*, unabhängig von der Dauer seiner Ausführung. Durch diese Umstellung wird die Verantwortung für die gleichmäßige Ausführung auf das Android-Framework übertragen und man erhofft sich

```
1  /**read next block from input_stream, copy it into the buffer and set sequence number and
       payload in rtp_packet */
2  private int prepareNextPacket() {
3     try {
4        num = input_stream.read(buffer, 12, buffer.length - 12);
5        if (num > 0) {
6           rtp_packet.setSequenceNumber(seqn++);
7           rtp_packet.setPayloadLength(num);
8        }
9     } catch (IOException e) {
10       Log.w(LOG_TAG, e);
11       this.cancel();
12    }
13    return num;
14 }
```

Listing 5.4: Vorbereitung des nächsten RTP-Pakets (aus HighPriorityTimerTask.java)

eine hohe Isochronität. Der Erfolg dieser Maßnahme ist in Kapitel 6 durch Auswertung der VoIP-Qualitätstests zu prüfen.

Das folgende RTP-Paket wird bereits direkt nach dem Versand für den nächsten Aufruf des *HighPriorityTimerTasks* vorbereitet und dadurch eine unterschiedlich lange Verzögerung durch diese Methode vermieden (vgl. Listing 5.4). Das ist allerdings nur möglich, da die Audiodaten nicht durch ein Mikrofon aufgezeichnet werden, sondern aus dem vorbereiteten Stream in einer Datei gelesen werden.

5.3 Inter-Process Communication unter Android mittels AIDL

Für die Kommunikation zwischen der Activity *VFphone* und dem Service *VFphoneService* wird AIDL verwendet. Dafür ist es notwendig, in der Activity eine *ServiceConnection* aufzubauen. Über diese wird der Service gestartet und an die Activity gebunden (vgl. Listing 5.5). Darüber hinaus erhält die Activity über die *ServiceConnection* den Zugriff auf die exportierten AIDL-Schnittstellen.

Da die *ServiceConnection* und die Oberfläche der Activity in separaten Threads laufen, ist es nicht möglich, direkt aus einer Verbindung zum Service auf die Oberfläche der Activity zu zugreifen. Beim Aufruf von Methoden der Activity-Klasse *ServiceCallback* vom Service aus, stellt das ein Problem dar. Entsprechende Zugriffe müssen in einem *Handler* gekapselt werden, der im Thread der Oberfläche läuft. Der *Handler* nutzt dafür Nachrichten und die „Message Queue“ des Threads, um die Aufrufe zu bearbeiten.

Eine AIDL-Schnittstelle wird durch eine „.aidl“-Datei spezifiziert. Diese ähnelt von der Sytntax her einem Java-Interface und wird durch das „aidl“-Werkzeug in ein Java-Interface mit einer abstrakten „Stub“-Klasse übersetzt. Das Android-Eclipse-Plugin übernimmt den Aufruf des Werkzeugs automatisch vor dem Übersetzen des Quellcodes.

```
1  private IVFPhoneServiceInterface vfService = null;
2
3  public void bindService() {
4     serviceConnection = new ServiceConnection() {
5        public void onServiceConnected(ComponentName component, IBinder service) {
6           Log.i(LOG_TAG, "service connected");
7           vfService = IVFPhoneServiceInterface.Stub.asInterface(service);
8           multicaster.notifyObservers(true);
9        }
10       public void onServiceDisconnected(ComponentName component) {
11          Log.i(LOG_TAG, "service disconnected");
12          multicaster.notifyObservers(false);
13       }
14    };
15    Intent i = new Intent(IVFPhoneServiceInterface.class.getName());
16    context.startService(i);
17    if (!context.bindService(i, serviceConnection, 0)) {
18       Log.i(LOG_TAG, "NO SERVICE RUNNING");
19       vfService = null;
20       return;
21    }
22    Log.d(LOG_TAG, "bind successfull");
23 }
```

Listing 5.5: Verbindung aus der Activity zum Service herstellen (aus ServiceAccess.java)

5.4 JNI auf Android

Wie in Kapitel 4.3.1.4 beschrieben, wird die VoIPFuture Library über JNI angebunden. Dafür ist eine Java-Schnittstelle und eine native Schnittstelle notwendig, die jeweils die Umsetzungen in ihre Umgebungen erledigen. Dabei sind unter Android Besonderheiten bezüglich der Implementierung von JNI-Komponenten zu beachten die im Folgenden vorgestellt werden.

Zudem bedarf die Installation der nativen Bibliothek unter Android besonderen Augenmerks, da nicht in das Standard-Verzeichnis für native Bibliotheken `/system/lib` geschrieben werden kann.

5.4.1 Installation einer nativen Bibliothek

Es besteht unter Android keine automatisierte und von Google vorgesehene Möglichkeit zur Installation von nativen Bibliotheken. Im Rahmen dieser Arbeit wurde deshalb für dieses Problem eine Lösung erarbeitet.

Das Android-Framework bietet die Möglichkeit, beliebige Dateien als Ressourcen in die „.apk“-Datei zu packen. Auf diese Dateien kann zur Laufzeit über den Anwendungs-Kontext und eine eindeutige Identifikationsnummer (ID) zugegriffen werden. Außerdem steht jeder Anwendung ein Verzeichnis im Dateisystem zur

Verfügung, auf das die Anwendung im Gegensatz zum Verzeichnis `/system/lib` Schreib- und Leserechte besitzt.

```
1  /** deploys the native lib from raw-resources in case its not already in the data path and
       opens the lib  */
2  public static void createLib(int resourceId, Context context) throws LibraryErrorException,
       LibraryFailureException {
3     _context = context;
4     File f = context.getFileStreamPath(jniLibName);
5     Log.d(LOG_TAG, "check for library " + jniLibName + " in " + f.getParent());
6     if (!f.exists()) {
7        InputStream inStream = context.getResources().openRawResource(resourceId);
8        try {
9           OutputStream outStream = context.openFileOutput(jniLibName, Context.MODE_PRIVATE);
10          byte[] buffer = new byte[1024];
11          int bytesRead = 0;
12          while ((bytesRead = inStream.read(buffer)) >= 0) {
13             outStream.write(buffer, 0, bytesRead);
14          }
15          Log.i(LOG_TAG, "created " + jniLibName);
16       } catch (FileNotFoundException e) {
17          Log.w(LOG_TAG, e);
18       } catch (IOException e) {
19          Log.w(LOG_TAG, e);
20       }
21    }
22    openLibrary();
23 }
```

Listing 5.6: Deployment der nativen Bibliothek auf Android (aus NativeVoIPVector.java)

Für die Verwendung mit JNI ist es möglich, beim ersten Aufruf der Anwendung die native Bibliothek aus den Ressourcen zu lesen und in diesem Anwendungs-Verzeichnis zu speichern (vgl. Listing 5.6). Danach kann sie mittels der Methode `System.load()` geladen werden, wie im Folgenden geschildert wird.

5.4.2 JNI in der Java-Umgebung

Bei der Nutzung von JNI gibt es zwischen der Java-Umgebung unter Android und einer Sun-Java-Umgebung keine wesentlichen Unterschiede.

Allerdings ist es durch die problematische Installation (vgl. Kapitel 5.4.1) notwendig, anstatt `System.loadLibrary()` die Methode `System.load()` zu verwenden (vgl. Listing 5.7). Diese Methode verlangt den vollständigen Pfad zur nativen Bibliothek, lädt aber Bibliotheken aus beliebigen Verzeichnissen.

```
1  private static boolean loadLib() throws LibraryFailureException {
2     try {
3        Log.d(LOG_TAG, "try to load " + _context.getFileStreamPath(jniLibName).getAbsolutePath())
             ;
4        System.load(_context.getFileStreamPath(jniLibName).getAbsolutePath());
5        Log.v(LOG_TAG, "Library loaded successfully");
6        libraryFound = true;
7     } catch (UnsatisfiedLinkError ule) {
8        Log.w(LOG_TAG, "error loading lib!", ule);
9        throw new LibraryFailureException("error loading lib!");
10    }
11    return libraryFound;
12 }
```

Listing 5.7: Laden einer nativen Bibliothek (aus NativeVoIPVector.java)

Die Dalvik VM lädt die nativen Bibliotheken in voneinander getrennte Speicherbereiche. Aus diesem Grund können mehrere native voneinander abhängige Bibliotheken nicht aufeinander zugreifen, auch wenn sie jeweils in der VM geladen sind. Dieses Problem tritt allerdings nur auf, wenn die Bibliotheken in Verzeichnissen liegen, die dem Android-Linker unbekannt sind. Eine Lösung bietet die Verwendung von „static“-Bibliotheken. Daher ist die VoIPFuture Library statisch in die „libVectorJni“-Bibliothek gelinkt. Abhängigkeiten von Bibliotheken im Standard-Verzeichnis „/system/lib“ sind von dieser Einschränkung allerdings nicht betroffen, da sie vom Linker automatisch nachgeladen werden können.

```
1  /** init the library, must be called before any other method */
2  private static native void _openLibrary() throws LibraryErrorException, Exception;
3
4  public static void openLibrary() throws LibraryFailureException, LibraryErrorException {
5     checkLib();
6     try {
7        _openLibrary();
8     } catch (LibraryErrorException e) {
9        throw e;
10    } catch (Exception e) {
11       throw new LibraryFailureException(e.getLocalizedMessage());
12    }
13 }
```

Listing 5.8: Öffnen und Initialisieren der VoIPFuture Library in Java (aus NativeVoIPVector.java)

Für die Nutzung einer nativen Funktion wird diese mit ihrer Signatur und dem `native` Schlüsselwort im Java-Quellcode deklariert (vgl. Listing 5.8). Außerdem werden in der vorgestellten Realisierung zusätzlich Exceptions abgefangen und die Integer-Rückgabewerte in Java-Objekte konvertiert.

```
/** Wrapper for openLibrary. It also calls VF_Init() */
JNIEXPORT void JNICALL Java_de_voipfuture_library_jni_NativeVoIPVector__1openLibrary
  (JNIEnv *env, jclass javaClass) {
    VF_ErrCode err;
    err = VF_Open(&psLibHandle);
    if (checkResult(env,err,"error while opening library")) return;
    err = VF_Init(psLibHandle);
    if (checkResult(env,err,"error while initializing library")) return;
    __android_log_write(LOG_DEBUG,"NATIVE_LIB","library init done!");
}
```

Listing 5.9: Öffnen und Initialisieren der VoIPFuture Library in C (aus libVectorJni.c)

5.4.3 JNI und Android-Entwicklung in C

Für die Erstellung einer JNI-C-Schnittstelle kann mit dem Werkzeug „javah“ aus dem Sun Java-SDK eine Header-Datei erstellt werden, die für alle im Java-Quellcode deklarierten nativen Methoden die C-Signatur enthält. Diese Funktionen können danach implementiert werden und müssen alle notwendigen Konvertierungen von nativen Datentypen in entsprechende Java Objekte durchführen (vgl. Listing 5.9).

Für die Verwendung einer nativen Bibliothek auf derzeit verfügbarer Android-Hardware und im Emulator ist es notwendig, diese für den ARM-Prozessor zu übersetzen. Da die Entwicklung für diese Arbeit auf einem x86-System stattfand, wird für die Übersetzung eine ARM-Cross-Compiling-Toolchain[4] benötigt. Wie in Kapitel 3.2.8.3 beschrieben besitzt der Android-Linker Besonderheiten beim Reservieren der Speicherbereiche der nativen Bibliotheken, die mittels einer angepassten armelf_linux_eabi.xsc-Datei vom Linker der Toolchain berücksichtig werden werden. Alle notwendigen Schritte für die Übersetzung wurden im Rahmen dieser Arbeit in einem Makefile für GNU-Make zusammengefasst.

```
// defined in system/lib/libcutils.so
extern int __android_log_write(int prio, const char *tag, const char *text);
```

Listing 5.10: Benutzung von Logcat in C (aus libVectorJniUtils.h)

Die androidspezifische libc Bionic bietet eine undokumentierte Schnittstelle für die Verwendung des Log-Frameworks Logcat. Die Verwendung unterscheidet sich nicht von der Java-Schnittstelle und es ist damit leicht möglich, alle Protokoll- und Debug-Nachrichten in die Logcat-Umgebung zu schreiben (vgl. Listing 5.10).

[4] `http://www.codesourcery.com/gnu_toolchains/arm` (aufgerufen am 09.06.2008)

6 VoIP-Qualitätstests

In diesem Kapitel werden die für diese Arbeit durchgeführten VoIP-Qualitätstests und der Testaufbau beschrieben. Dabei sollen die Vorraussetzungen für die Qualitätstests aufgezeigt und die verwendeten Komponenten sowie deren Anordnung aufgelistet werden. Anschließend werden die daraus gewonnenen Ergebnisse vorgestellt.

6.1 Durchführung der Qualitätstests

Der Testaufbau für die VoIP-Qualitätstests wurde möglichst einfach gehalten. Die Tests wurden ausschließlich auf dem Android-Emulator durchgeführt und es wurde nicht auf spezifischer Android-Hardware getestet. Bei den Tests kommuniziert der Emulator über die virtuelle Netzwerk-Schnittstelle „TAP“ des Linux-Kernels mit dem Host-Rechner (vgl. Abbildung 6.1). Die Bandbreiten-Beschränkung und die speziellen Netzwerk-Emulationen (GPRS, EDGE oder UMTS) des Android-Emulators wurden bei den Qualitätstests nicht verwendet.

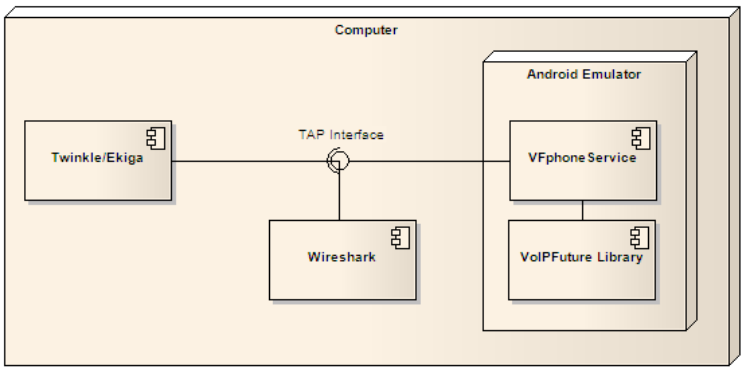


Abbildung 6.1: Verteilung der verschiedenen Komponenten für die VoIP-Qualitätstests

Die Tests wurden auf einem Intel Core2Duo E6750 und einem AMD Athlon 64 X2 5000+ durchgeführt. Um Eigenheiten der Android-Lösung auszuschließen und einen Vergleich zu anderen Softphone-Lösungen ziehen zu können, wurde als Gegenstelle das Softphone Twinkle[1] verwendet. Dabei liefen die beiden User-Agents und die Test-Werkzeuge gemeinsam auf dem selben Computer. Auf dem Intel Core2Duo befand sich

[1] Version 1.2 `http://www.xs4all.nl/~mfnboer/twinkle/index.html` (aufgerufen am 28.11.2008)

ein Debian sid auf dem Stand vom 1. Dezember 2008 mit einem „vanilla“-Linux-Kernel 2.6.27. Auf dem AMD Athlon wurde Ubuntu Intrepid Ibex auf dem Stand vom 1. Dezember 2008 mit dem Linux-Kernel 2.6.27-10-generic eingesetzt. Für die hier vorgestellten VoIP-Qualitätstests wurde das Audioformat G.711 µ-law mit einer Samplerate von 20ms verwendet.

Durch die Beschränkung auf ein System wurde die Beeinträchtigung der Messergebnisse durch weitere Netzwerk-Komponenten vermieden. Die Analyse konnte auf die an einer VoIP-Verbindung zwingend beteiligten Systeme konzentriert werden.

Um Analyse-Daten außerhalb des Android-Emulators zu gewinnen wurde das Netzwerk-Analyse-Werkzeug Wireshark[2] verwendet. Die aufgezeichneten Daten wurden durch das VoIPFuture-interne Werkzeug cap2vad[3] in VAD-Dateien umgewandelt. Außerdem wurden die Wireshark-Daten mit Hilfe des eines internen Werkzeugs visualisiert. Wireshark wurde auch während der Entwicklung für die Analyse und Validierung der übertragenen Pakete verwendet.

Die mit der Android-Anwendung und der VoIPFuture Library 1.2.2.0 gewonnenen Analyse-Daten wurden mit dem Android-Entwicklungs-Werkzeug „adb“ aus dem Emulator auf den Host-Rechner übertragen. Alle untersuchten Verbindungen wurden gleichzeitig unter Android mit Hilfe der VoIPFuture Library und auf dem Host-System mit Wireshark aufgezeichnet.

Die von der VoIPFuture Library und dem cap2vad Werkzeug erzeugten VAD-Dateien wurden mit dem VoIPFuture VectorViewer 1.6.0.198M ausgewertet.

Aufgrund eines Fehlers im Emulator des SDK 1.0r1 (vgl. Issue 906[4]) wurde für die Analyse eine Emulator-Version vom 31. Oktober 2008 aus dem *Version Control System* git[5] verwendet. Der Fehler verhinderte die Verwendung des TAP-Netzwerks mit dem Emulator und wurde in Revision fdd16a24445e56ccce4a29a6cec4a45886f6ad62 am 28. Oktober 2008 von Google behoben.

6.2 Ergebnisse der Qualitätstests

Im Folgenden werden die Ergebnisse der VoIP-Qualitätstests vorgestellt. Diese werden jeweils getrennt nach dem Absender der analysierten RTP-Pakete betrachtet. Die Absender sind die im Rahmen dieser Arbeit entwickelte Android-VoIP-Anwendung VFphone und das Linux-Softphone Twinkle.

Auch wenn sich die vorliegende Arbeit mit der VoIP-Qualitätsmessung unter Android beschäftigt, sind die Messungen von Streams des Softphones Twinkle dennoch interessant. So werden diese Streams zum Teil unter Android aufgezeichnet. Dabei werden zu Messungen durch Wireshark auf dem Host-Rechner unterschiedliche Ergebnisse erzielt, woran sich feststellen lässt, dass der Messpunkt für eine VoIP-Qualitätsmessung von großer Bedeutung sein kann. Des Weiteren zeigen diese Analysen, dass die VoIP-Qualität stark von der verwendeten Hardware abhängt.

[2] Version 1.0.3 `http://www.wireshark.org` (aufgerufen am 8.12.2008)
[3] Version 1.6.0 mit der VoIPFuture Library 1.4.255.0
[4] `http://code.google.com/p/android/issues/detail?id=906` (aufgerufen am 2.12.2008)
[5] `http://android.git.kernel.org/` (aufgerufen am 8.12.2008)

6.2.1 Streams von VFphone

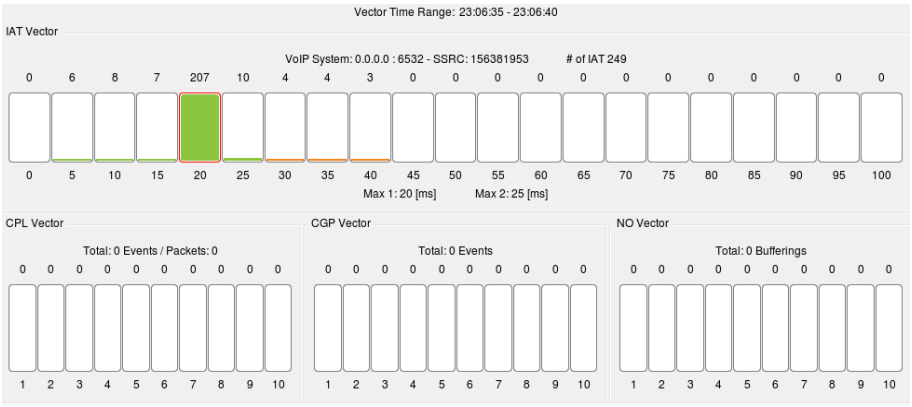


Abbildung 6.2: Verteilung der Pakete in einem mittleren Vektor eines Android-Streams, aufgezeichnet unter Android mit der VoIPFuture Library, ausgewertet im VoIPFuture VectorViewer

Die von der Android-VoIP-Anwendung VFphone erzeugten Streams enthalten eine für ein Softphone relativ gleichmäßige Verteilung der RTP-Pakete. In einem beliebigen Vektor aus der Mitte eines Streams wurden die meisten RTP-Pakete mit dem optimalen Abstand von 20ms empfangen (vgl. Abbildung 6.2). Der maximale Abstand der Pakete lag bei nur 42,5ms. Dieser Wert lässt sich auf der Seite des Empfängers ohne Probleme durch einen so genannten „Jitter-Buffer“ ausgleichen. Dieser puffert die empfangenen Pakete für eine kurze Zeit, um sie in korrekter Reihenfolge und im idealen zeitlichen Abstand wiedergeben zu können. Daher wurde auch eine „ausreichende“ Qualität (decent) mit einem noch akzeptablen Jitter von der VoIPFuture Library ermittelt (vgl. Abbildung 6.3).

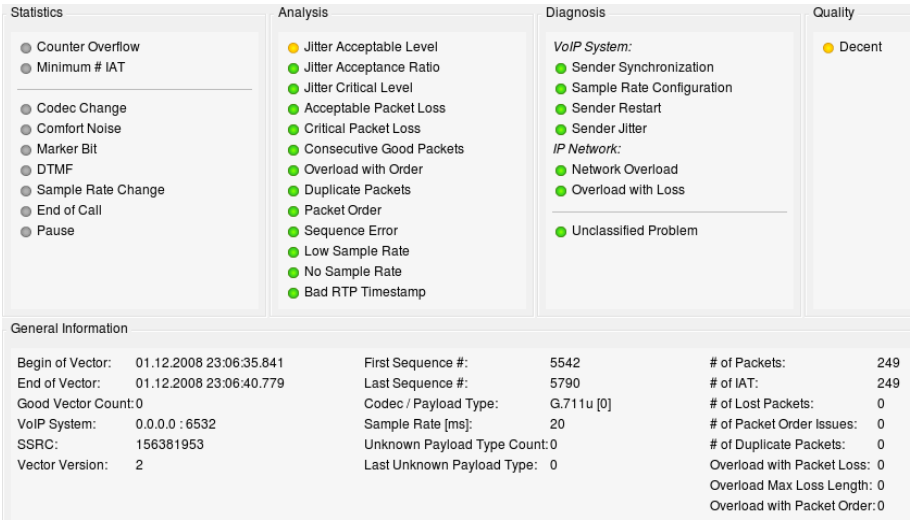


Abbildung 6.3: Statistik eines mittleren Vektors von einem Android-Stream, aufgezeichnet mit der VoIPFuture Library unter Android , ausgewertet im VoIPFuture VectorViewer

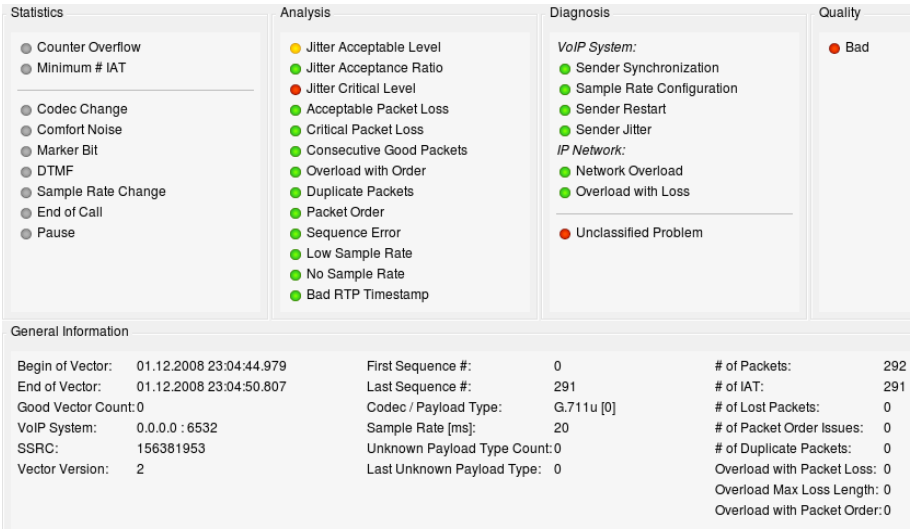


Abbildung 6.4: Statistik des ersten Vektors von einem Android-Stream, aufgezeichnet mit der VoIPFuture Library unter Android, ausgewertet im VoIPFuture VectorViewer

Trotzdem gibt es beim Start des Streams im ersten Vektor deutliche Störungen und einen kritischen Jitter (vgl. Abbildung 6.4). Dieser Effekt ist vermutlich auf den Start der Sub-Activity und andere Initialisierungen des VoIP-Systems zurückzuführen, die am Anfang einer VoIP-Verbindung durchgeführt werden. Dieser Vektor wird deshalb durch die VoIPFuture Library als schlecht („bad“) markiert. So ist in Abbildung 6.5 erkennbar, dass 13 Pakete mit einem Abstand von nur 0ms bis 2,5ms empfangen wurden und 36 Pakete mit einem Abstand von nur 2,5ms bis 7,5ms. Gleichzeitig traten auch sehr starke Verzögerungen auf und zwei Pakete hatten eine Interarrival Time von 100ms oder größer.

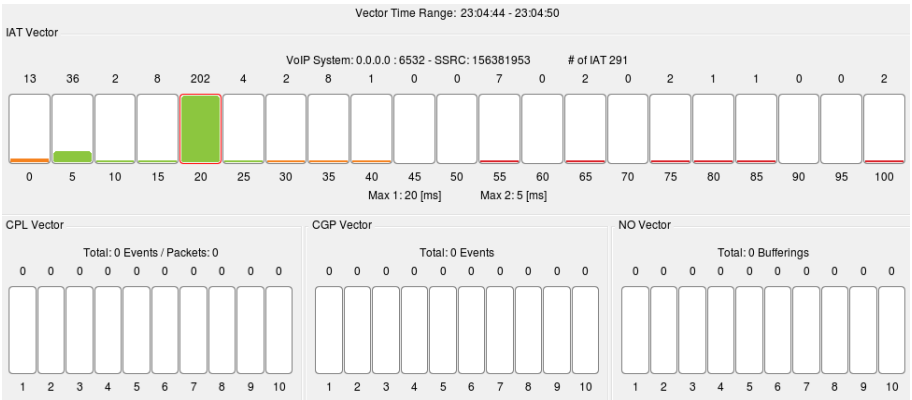


Abbildung 6.5: Verteilung der Pakete im ersten Vektor eines Android-Streams, aufgezeichnet mit der VoIPFuture Library unter Android, ausgewertet im VoIPFuture VectorViewer

Dieser Effekt zu Beginn der Verbindung ist auch deutlich in der Auswertung erkennbar (vgl. Abbildung 6.6). Dort steht jeder blaue Punkt für ein RTP-Paket. Dieser ist im zeitlichen Verlauf der Verbindung auf der X-

Achse eingeordnet und dem zeitlichen Abstand zum letzten Paket (Interarrival Time, vgl. Kapitel 2.2.3.4) auf der Y-Achse.

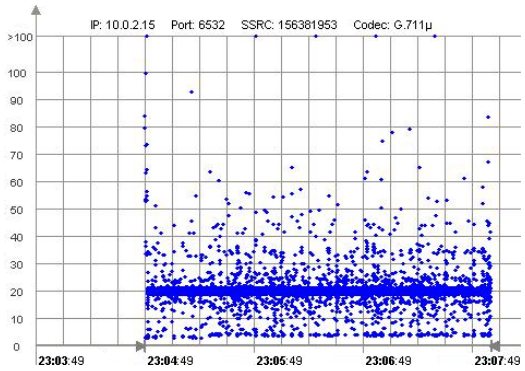


Abbildung 6.6: Android-Stream, aufgezeichnet mit Wiresharkt

Neben der Störung zu Beginn einer Verbindung sind in der Auswertung auch in regelmäßigen Abständen starke Ausschläge mit einer Interarrival Time von 100ms oder mehr zu erkennen. Dieser Effekt könnte auf die periodische Ausführung des Garbage-Collectors unter Android zurück zu führen sein. So finden sich im Logcat-Log ungefähr zu diesen Zeitpunkten Einträge in der folgenden Form: `DEBUG/dalvikvm(561): GC freed 19664 objects / 568072 bytes in 88ms`. Das bedeutet, dass der Garbage Collector jeden Durchlauf bis zu 20.000 Objekte löscht und dafür 70ms bis 100ms benötigt. Für die Qualität von VoIP kann dieser Zeitraum bereits kritisch sein.

Trotz der beschriebenen Probleme lässt sich auch in der Auswertung (vgl. Abbildung 6.6) eine deutliche Häufung der Pakete bei einer Interarrival Time von 20ms erkennen und damit auf einen zufriedenstellenden Stream bezüglich der Isochronität schließen.

Die vorgestellten Streams wurden auf dem Intel Core2Duo-System erzeugt. Für die Auswertungen von diesem System wurde sich aufgrund der höheren Performanz entschieden, wodurch weniger Beeinträchtigungen durch den Emulator zu erkennen sind. So gab es prinzipiell die gleichen Effekte auch in Qualitätstests auf dem AMD-System, allerdings mit stärkeren Auswirkungen und zusätzlichen Beeinträchtigungen durch andere Prozesse, die nicht reproduzierbar sind.

6.2.2 Streams des Linux-Softphones Twinkle

Verglichen mit der Android-Anwendung VFphone ergaben die Messungen des Linux-Softphones Twinkle relativ schlechte Streams. Das Ergebnis überrascht deshalb, da Twinkle nicht in einer VM oder einem Emulator läuft und deshalb mehr Einfluss auf die Hardware ausüben und diese direkt ansprechen könnte. Allerdings ist zu beachten, dass alle Tests jeweils auf einem einzigen System durchgeführt wurden. Aus diesem Grund ist es nicht möglich, Wechselwirkungen durch die verschiedenen Anwendungen auszuschließen. Um das zu verhindern, müssten die Tests im Rahmen weiterführender Untersuchungen auf verteilten Computern mit einem dedizierten Monitor durchgeführt werden.

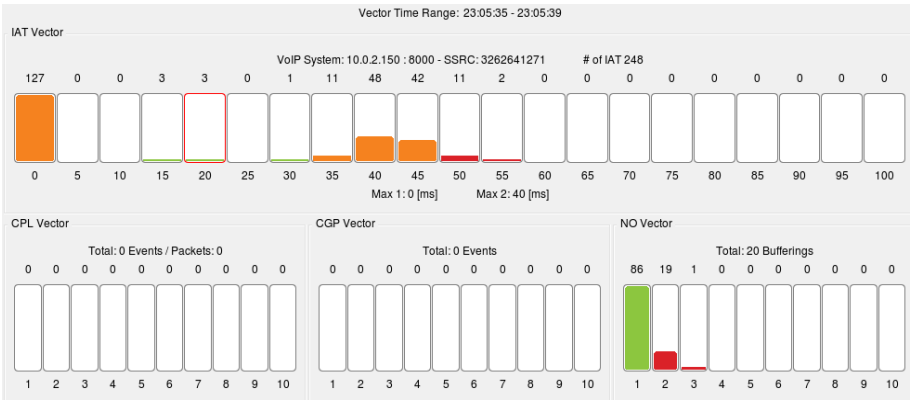


Abbildung 6.7: Verteilung der Pakete in einem mittleren Vektor eines Twinkle-Streams, aufgezeichnet mit Wireshark, ausgewertet mit cap2vad und im VoIPFuture VectorViewer

Bei der Messung mit Wireshark sind sehr deutliche Fehlermuster zu erkennen. Wie in Abbildung 6.7, welche die Messung durch Wireshark auf dem Intel Core2Duo-System mit Hilfe des VectorViewer abbildet, zu erkennen ist, stimmt die Interarrival Time der einzelnen Pakete in fast keinem Fall mit den erwarteten 20ms der Samplerate überein. Vielmehr ergeben sich starke Häufungen der Pakete bei 40ms und bei 0ms. Durch die große Anzahl der Pakete mit einer Interarrival Time von 0ms bis 2,5ms erkennt der VectorViewer auch 20 Mal ein Network Buffering. Danach traten ein Mal drei Pakete und 19 Mal zwei Pakete in direkter Folge auf.

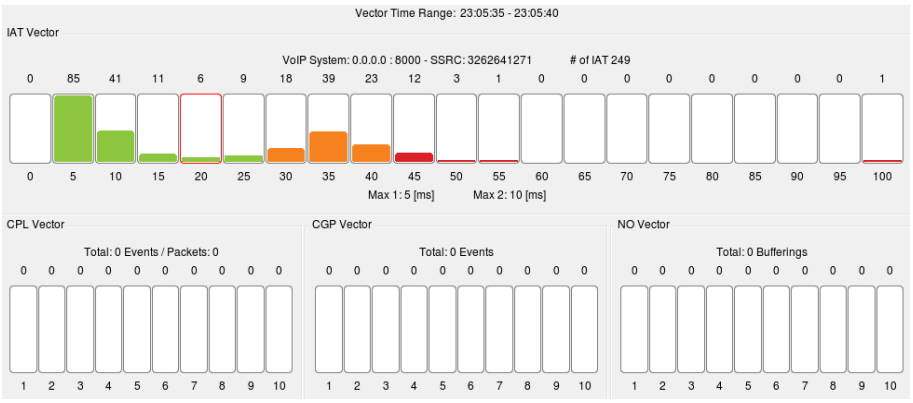
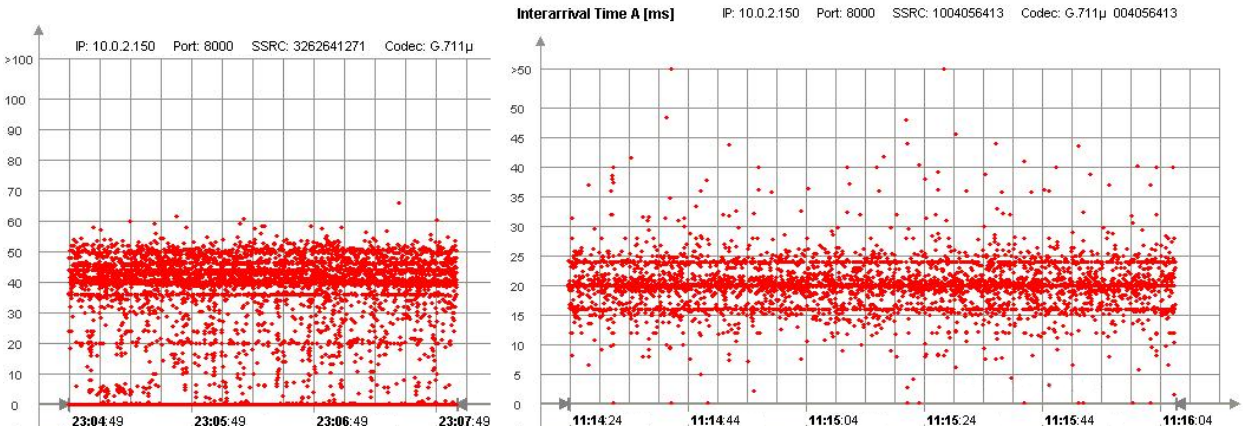


Abbildung 6.8: Verteilung der Pakete in einem mittleren Vektor eines Twinkle-Streams unter Android gemessen mit der VoIPFuture Library, ausgewertet im VoIPFuture VectorViewer

Wird der gleiche Zeitraum mit der VoIPFuture Library unter Android gemessen und im VectorViewer ausgewertet, ergibt sich ein leicht unterschiedliches Bild (vgl. Abbildung 6.8). Auch hier sind deutliche Abweichungen von den optimalen 20ms zu erkennen. Allerdings wird weder Network-Buffering erkannt, noch sind die Abweichungen so groß wie bei der Wireshark-Messung. Es bestehen deutliche Häufungen der Interarrival Time bei 5ms und 35ms.



(a) auf dem Intel Core2Duo-System

(b) auf dem AMD-System

Abbildung 6.9: Twinkle-Stream aufgezeichnet mit Wireshark

Außerdem lässt sich mit den Qualitätstests des Linux-Softphones Twinkle die Abhängigkeit der Messungen von der verwendeten Hardware verdeutlichen. So sind die schon beschrieben Abweichungen von der Interarrival Time auch auf Abbildung 6.9a erkennbar. Auf dem AMD-System hingegen sind diese Abweichungen sehr viel geringer und die Interarrival Time nähert sich den optimalen 20ms an (vgl. Abbildung 6.9b). Allerdings lässt sich in dieser Messung eine deutlich stärkere Streuung der Pakete oberhalb von 25ms erkennen. Dieser Effekt ist auch in der Anzeige der Maxima der Interarrival Time über alle Vektoren in Abbildung 6.10 zu erkennen. Die Auswertung zeigt noch deutlicher die Häufung der Pakete bei 20ms Interarrival Time, entsprechend der Samplerate.

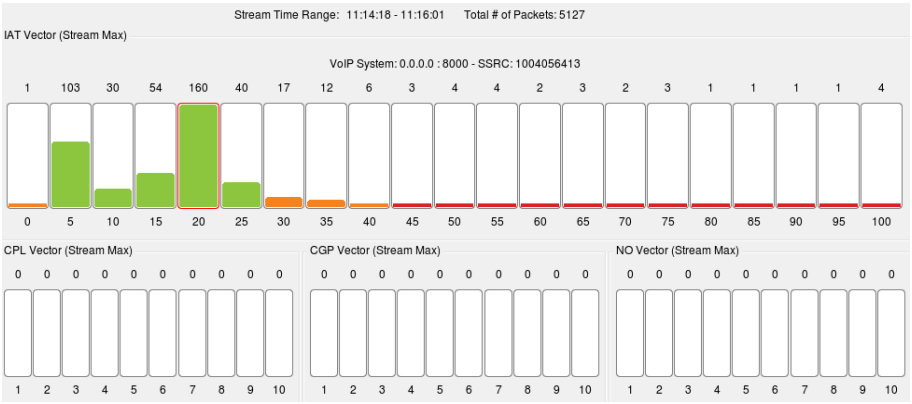


Abbildung 6.10: Maxima der Verteilung der Pakete über alle Vektoren eines Twinkle-Streams, mit der VoIPFuture Library unter Android gemessen, ausgewertet im VoIPFuture VectorViewer

Eine mögliche Ursache für die Unterschiede zwischen den Systemen könnten die unterschiedlichen Linux-Kernel-Versionen mit verschiedenen Schedulern sein. Aber auch verschiedene Hardware-Timer könnten für die Unterschiede verantwortlich sein. Zur abschließenden Klärung müssten weitere Untersuchen durchgeführt werden, die den Rahmen dieser Arbeit übersteigen würden.

7 Fazit

Abschließend werden die Ergebnisse der vorliegenden Arbeit vorgestellt. Des Weiteren gibt es einen Ausblick auf mögliche Erweiterungen und zusätzliche Untersuchungen.

7.1 Ergebnisse

Im Verlauf dieser Arbeit wurde die neue Mobiltelefon-Plattform Android ausführlich analysiert und deren Möglichkeiten, aber auch Einschränkungen, aufgezeigt. Die zuvor nicht vorhandene Unterstützung von VoIP konnte durch die Entwicklung einer Android-VoIP-Anwendung kompensiert werden. Außerdem verlief die Portierung der VoIPFuture Library auf diese Architektur erfolgreich, ebenso wie die Integration in den neuen User-Agent. Dabei konnten alle ermittelten Probleme bei der Realisierung von nativen Anwendungen und Bibliotheken unter Android gelöst und beide Welten, die native und die Java-Welt mit der Dalvik VM, erfolgreich mit Hilfe von JNI zusammengeführt werden. Hervorzuheben ist in diesem Zusammenhang die Installation einer nativen Bibliothek für eine Android-Anwendung, für die in dieser Arbeit eine auch im produktivem Umfeld funktionsfähige Lösung entwickelt wurde. Außerdem beweist der erfolgreiche Einsatz der VoIPFuture Library auf der Android-ARM-Architektur dessen Portierbarkeit.

Damit sind ausführliche VoIP-Qualitätstests unter Android möglich und wurden im Rahmen dieser Arbeit auch durchgeführt. Die realisierten Qualitätstests konnten schon während der Entwicklung Fehler in der Implementierung der SIP- und RTP-Bibliothek verdeutlichen und für eine frühe Behebung dieser Fehler sorgen. In diesen VoIP-Qualitätstests zeigte sich, dass die Android-Plattform gut für VoIP geeignet ist und auch die Qualitätsmessung erfolgreich mit neuen Erkenntnissen durchgeführt werden kann. Nur eine Messung auf spezieller Android-Hardware konnte mangels Ressourcen nicht ausgeführt und damit deren eventuelle Auswirkungen nicht geklärt werden.

7.2 Rückblick

Der Entwurf der Android-VoIP-Anwendung in einen Service und eine Activity hat sich bewährt. Diese Aufteilung führt zu einer geringen Kopplung, ohne dass zum Beispiel durch die Verwendung von AIDL zusätzliche Probleme bei der Realisierung entstanden.

Die Verwendung der SIP- und RTP-Bibliothek MjSIP war dagegen eine problematische Lösung. Neben den geschilderten Problemen mit der Performanz der Bibliothek unter Android und fehlender Funktionen, war

die Behebung von weiteren Fehlern an vielen Stellen erforderlich. Auch nach Abschluß dieser Arbeit sind nicht alle der Probleme gelöst. Durch den großen Code-Umfang der MjSip-Bibliothek, war es auch nicht praktikabel Unit-Tests mit Hilfe des JUnit-Frameworks durchzuführen. Zwar bietet das Android-Framework hierfür eine Unterstützung, aber die Erweiterung von MjSip um Unit-Tests hätte einen zu großen Aufwand für diese Arbeit bedeutet. Deswegen würde es sich bei einer erneuten Realisierung anbieten, entweder eine eigene SIP- und RTP-Bibliothek zu entwickeln oder pjsip zu verwenden, da die Nutzung von JNI nach dieser Arbeit nur wenige Probleme erwarten lässt.

7.3 Ausblick

Um die im Rahmen dieser Arbeit entwickelte und realisierte Android-VoIP-Anwendung in ein marktreifes Produkt zu verwandeln wäre es notwendig, die noch fehlenden „nice to have“-Anforderungen zu implementieren. Vor allem ist dabei die fehlende Unterstützung der Registrierung an SIP-Proxies zu erwähnen. Aber auch eine weitere Optimierung der MjSip-Bibliothek wäre notwendig, da diese noch mehr Fehler beinhaltet, die nicht abschließend gelöst werden konnten.

Hilfreich für die VoIP-Qualitätsmessung wäre auch eine Integration von parametrisierbaren Fehlermustern, die einen Test der VoIPFuture Library und die Nachstellung und Analyse von speziellen Szenarios ermöglichen.

Es muss auch geklärt werden für welche Zwecke die kontinuierliche Messung und Diagnose verwendet werden soll. So sind Funktionen wie ein automatisches Umschalten der Verbindung von zum Beispiel WLAN auf UMTS vorstellbar, um aufgetretene und diagnostizierte Fehler zu beheben. Aber auch eine fehlende Priorisierung der VoIP-Verbindung könnte erkannt und automatisch ohne Benutzereingriff behoben werden. Außerdem ließe sich das Netzwerk durch die Netzbetreiber mit Hilfe der Analyse und Diagnose direkt auf den Endgeräten, wie zum Beispiel Android-Mobiltelefonen, vollständig und umfassend überwachen. Damit ist eine frühzeitige Reaktion auf sich abzeichnende Probleme möglich.

Bezüglich der Android-Plattform wäre ein Test auf der nun existierenden Hardware wünschenswert, wie auch eine genauere Untersuchung der Plattform und der hier vorgestellten Realisierung.

Abkürzungsverzeichnis

ADK ..... *Android Development Kit*

AIDL ..... *Android Interface Definition Language*

ANSI ..... *American National Standards Institute*

API ..... *Application Programming Interface*

CoS ..... *Class of Service*

CSRC ..... *Contributing Source*

DoS ..... *Denial of Service*

GPL ..... *GNU General Public License*

GSM ..... *Global System for Mobile Communications*

GUI ..... *Graphical User Interface*

HTTP ..... *Hypertext Transfer Protocol*

IMS ..... *IP Multimedia System*

IP ..... *Internet Protocol*

IPC ..... *Inter-Process Communication*

ISDN ..... *Integrated Services Digital Network*

ITU-T ..... *ITU Telecommunication Standardization Sector*

J2EE ..... *Java Enterprise Edition*

J2SE ..... *Java Standard Edition*

JNI ..... *Java Native Interface*

LDP ..... *Label Distribution Protocol*

libc ..... *Standard Library C*

MGC ..... *Media Gateway Controller*

MOS ..... *Mean Opinion Score*

MPLS ..... *Multiprotocol Label Switching*

NGN ..... *Next Generation Network*

OHA ..... *Open Handset Alliance*

PESQ ..... *Perceptual Evaluation of Speech Quality*

POTS ..... *Plain Old Telephone System*

PSTN ..... *Public Switched Telephone Network*

QoA ..... *Quality of Analysis*

QoD ..... *Quality of Diagnosis*

QoE ..... *Quality of Experience*

QoS ..... *Quality of Service*

RSVP ..... *Resource Reservation Protocol*

RSVP-TE ..... *Resource Reservation Protocol-Traffic Engineering*

RTCP ..... *Real-time Transport Control Protocol*

RTP ..... *Real-time Transport Protocol*

SDK ..... *Software Development Kit*

SDP ..... *Session Description Protocol*

SIP ..... *Session Initiaton Protocol*

SOA ..... *Service-Oriented Architecture*

SSRC ..... *Synchronization Source*

TAP ..... *Test Access Port*

TCP ..... *Transmission Control Protocol*

ToS ..... *Type of Service*

UDP ..... *User Datagram Protocol*

UMTS ..... *Universal Mobile Telecommunications System*

VAD ..... *VoIPFuture Analytic and Diagnostic file format*

VM ..... *Virtual Machine*

VoIP ..... *Voice over IP*

Glossar

American National Standards Institute *Das US-amerikanische Gegenstück zum Deutsche Institut für Normung e.V. (DIN). Eine Stelle zur Normierung von Verfahrensweisen und Spezifikationen.*

Application Programming Interface *Eine definierte Schnittstelle einer Bibliothek oder Anwendung, durch die sich diese auf Quellcode-Ebene von anderen Entwicklern verwenden lässt.*

GNU General Public License *Eine weit verbreitete und von der Open Source Initiative zertifizierte Open Source Lizenz der Free Software Foundation.*

Hops *Mit Hops werden in Netzwerken die Wege von einem Knoten zum anderen bezeichnet.*

Internet Protocol *Ein Protokoll mit dem Rechner in einem Netzwerk adressiert werden können. Es erlaubt Verbindungen zwischen Rechnern aufzubauen und ist die Implementierung der Vermittlungsschicht des OSI-Modell.*

ITU Telecommunication Standardization Sector *Die ITU-T erarbeitet internationale technische Standards und Normen.*

Java Native Interface *Eine Schnittstelle, mit der man aus der Java Virtual Machine eine native Bibliothek ansprechen kann. Es ist auch möglich aus einer nativen Bibliothek oder Anwendung Javacode auszuführen.*

Java Standard Edition *Die in der Regel verwendete Java-Version von Sun. Sie wird für Anwendungen auf dem Desktop verwendet.*

Jitter *Die Schwankung der Übermittlungszeiten von IP-Paketen.*

Next Generation Network *Ein Telekomunikations-Netzwerk, das Daten-, Video- und Sprachkommunikation in einem Netz vereint. Realisiert wird es über das Internet Protocol.*

Open Handset Alliance *Zusammenschluss von über 30 Firmen unter der Federführung von Google. Von der Open Handset Alliance wird das Android-Framework entwickelt.*

OSI-Modell *Ein Netzwerk-Modell, das die Kommunikation vertikal in einzelne Schichten aufteilt, die aufeinander aufbauen.*

Plain Old Telephone System *Das analoge Telefonnetz, das elektromechanisch vermittelt wurde, nur reine Telefonie erlaubte, und dabei auch keine weiteren Dienstmerkmale unterstützte.*

prelinking *Mit „prelinking“ wird ein Verfahren bezeichnet, dass mit Hilfe von „relocating“ die Performanz-Probleme beim Laden von dynamischen Bibliotheken versucht zu umgehen.*

Public Switched Telephone Network *Das auch heute noch verwendete digital vermittelte Telefonnetz mit erweiterten Dienstmerkmalen.*

Quality of Service *Beschreibt die Qualität eines Dienstes in einem paketorientierten Netz aufgrund von festgelegten Parametern.*

Sandbox *Die Sandbox ist ein abgeschlossenes und von der Umgebung abgeschottetes System, in dem Anwendungen ausgeführt werden.*

Software Development Kit *Ein Software Development Kit ist die Zusammenstellung einer Umgebung aus verschiedenen Programmen und Werkzeugen für die Entwicklung von Software. Es kann wie im Fall von Android zum Beispiel einen Emulator, Bibliotheken und Erweiterungen für die Entwicklungsumgebung beinhalten.*

Test Access Port *Ein Gerät, das einen zusätzlichen Netzwerkport in eine Netzwerkverbindung schleift, ähnlich einem T-Stück in einer BNC-Verkabelung*

Virtual Machine *Eine virtuelle Laufzeitumgebung für Programme auf einem Host-System. Bekannte Beispiele sind die SUN Java Virtual Machine oder Smalltalk 80.*

Literaturverzeichnis

[Acher 2003] ACHER, Georg: *JIFFY - ein FPGA-basierter Java Just-in-time-Compiler für eingebettete Anwendungen*, Technische Universität München, Dissertation, 2003. – URL http://d-nb.info/970329814

[Aliance 2008] ALIANCE, Open H.: *Android*. 2008. – URL http://www.openhandsetalliance.com/android_overview.html

[Badach 2007] BADACH, Anatol: *Voice over IP - Die Technik: Grundlagen, Protokolle, Anwendungen, Migration, Sicherheit*. 3. erweiterte Auflage. Hanser, 2007. – ISBN 978-3-446-40666-7

[Bloch 2002] BLOCH, Joshua: *Effektiv Java programmieren*. Addison-Wesley, 2002. – ISBN 3-8273-1933-1

[Bornstein 2008] BORNSTEIN, Dan: Dalvik VM Internals. In: *Google I/O*, Google, Mai 2008. – URL http://sites.google.com/site/io/dalvik-vm-internals

[Brady 2008] BRADY, Patrick: Anatomy & Physiology of an Android. In: *Google I/O*, Google, Mai 2008. – URL http://sites.google.com/site/io/anatomy--physiology-of-an-android

[Detken 2002] DETKEN, Kai-Oliver: *Echtzeitplattformen für das Internet: Grundlagen, Lösungsansätze der sicheren Kommunikation mit QoS und VoIP*. Addison-Wesley, 2002. – ISBN 3-8273-1914-5

[Fey 2008] FEY, Jürgen: Android: Der Rivale. In: *LINUX Magazin, Beilage: Embedded Linux* 03/08 (2008), März, S. 11–15. – URL http://www.linux-magazin.de/themengebiete/special/embedded/android_der_rivale. – ISSN 1432-640 X

[Given 2008] GIVEN, David: *Problem loading C++ library in android*. 2008. – URL http://groups.google.com/group/android-internals/msg/7ee8c3621157b075

[Google 2008a] GOOGLE: *Android - An Open Handset Alliance Project*. 2008. – URL http://code.google.com/android

[Google 2008b] GOOGLE: *Android - Open Source Project*. 2008. – URL http://www.android.com

[Google 2008c] GOOGLE: *Final SDK build available (84853) deadline extended to Tuesday, August 5*. aufgerufen am 30.10.2008. 2008. – URL http://groups.google.com/group/android-discuss/browse_frm/thread/86337bb42ae18c72/455edc60c3452790

[Gordon 1998] GORDON, Rob: *Essential JNI: Java Native Interface*. Prentice Hall PTR, 1998. – ISBN 0-13-679895-0

[Gramlich 2008] GRAMLICH, Nicolas: *Android Programming*. release.002. anddev.org, 2008. – URL http://andbook.anddev.org

[Guy 2008] GUY, Romain: *What is google's official position on JNI ?* aufgerufen am 30.10.2008. 2008. – URL http://groups.google.com/group/android-developers/msg/9c7ffe8c2b0b12e2

[Hackborn 2008] HACKBORN, Dianne: *Android IDL support [C++, wire protocol spec info needed]*. aufgerufen am 30.10.2008. 2008. – URL http://groups.google.com/group/android-developers/msg/562cc0b254d96cec

[Hackborn und Parks 2007] HACKBORN, Dianne ; PARKS, Jason: *Google Developer Podcast Episode Twelve: Android with Dianne Hackborn and Jason Parks*. Dezember 2007. – URL http://google-code-updates.blogspot.com/2007/12/google-developer-podcast-episode-twelve.html

[Herold 1999] HEROLD, Helmut: *Linux-Unix-Systemprogrammierung*. 2. überarbeitete Auflage. Addison-Wesley-Longman, 1999. – ISBN 3-8273-1512-3

[Justin 2008] JUSTIN: *Java-Less Android Development? Dalvik documentation?* aufgerufen am 30.10.2008. 2008. – URL http://groups.google.com/group/android-beginners/msg/a567c21a4a72ce96?

[Kennke 2007] KENNKE, Roman: *Efficient JNI programming*. Juli 2007. – URL http://kennke.org/blog/2007/07/20/efficient-jni-programming-part-i/

[Li und Knudsen 2005] LI, Sing ; KNUDSEN, Jonathan: *Beginning J2ME: From Novice to Prefessional, Third Edition*. Apress, 2005. – ISBN 1-59059-479-7

[Liang 1999] LIANG, Sheng: *Java Native Interface: Programmer's Guide and Specification*. Addison Wesley, 1999. – URL http://java.sun.com/docs/books/jni/. – ISBN 0-201-32577-2

[Minnerup 2008] MINNERUP, Willi: Anwender warten auf den mobilen VoIP-Boom. In: *funkschau* 22/08 (2008), November, S. 6–9. – ISSN 0016-2841

[Morril 2008a] MORRIL, Dan: *Dalvik VM*. aufgerufen am 30.10.2008. 2008. – URL http://groups.google.com/group/android-developers/msg/6574e60b4add1fc4

[Morril 2008b] MORRIL, Dan: *Some information on APIs removed in the Android 0.9 SDK beta*. aufgerufen am 30.10.2008. 2008. – URL http://android-developers.blogspot.com/2008/08/some-information-on-apis-removed-in.html

[Morrill 2008] MORRILL, Dan: Inside the Android Application Framework. In: *Google I/O*, Google, Mai 2008. – URL http://sites.google.com/site/io/inside-the-android-application-framework

[Müller 2007] MÜLLER, Bernd: *Entwicklung und Untersuchung eines Java basierten VoIP Clients für mobile Endgeräte*, Fachhochschule Köln, Diplomarbeit, 2007. – URL http://dnserver.nt.fh-koeln.de/grebe/Diplom/DA/bernd_mueller_2007.pdf

[Roychowdhury 2008] ROYCHOWDHURY, Arjun: *SIP UA for Android (+stack + RTP) released*. aufgerufen am 4.6.2008. 2008. – URL http://blog.roychowdhury.org/2008/04/29/sip-ua-for-android-stack-rtp-released

[Rupp u. a. 2002] RUPP, Dr.-Ing. S. ; SIEGMUND, Gerd ; LAUTENSCHLAGER, Wolfgang: *SIP - Multimediale Dienste im Internet: Grundlagen, Architektur, Anwendungen*. dpunkt.verlag, 2002. – ISBN 3-89864-167-8

[Shi u. a. 2005] SHI, Yunhe ; CASEY, Kevin ; ERTL, M. A. ; GREGG, David: Virtual machine showdown: Stack versus registers. (2005). – URL `http://www.sagecertification.org/events/vee05/full_papers/p153-yunhe.pdf`

[Siegmund 2002] SIEGMUND, Gerd: *Next Generation Networks: IP-basierte Telekommunikation*. Hüthig, 2002. – ISBN 3-7785-3963-9

[Trick und Weber 2007] TRICK, Ulrich ; WEBER, Frank: *SIP, TCP/IP und Telekommunikationsnetze: Next Generation Networks and VoIP - konkret*. 3. Auflage. Oldenbourg Wissenschaftsverlag, 2007. – ISBN 978-3-486-58228-4

[Ullenboom 2008] ULLENBOOM, Christian: *Java ist auch eine Insel*. 7. aktualisierte Auflage. Galileo Computing, 2008. – URL `http://www.galileocomputing.de/openbook/javainsel7`. – ISBN 978-3-8362-1146-8

[VAD File Format 2008] KRÜGER, Michael: *VAD File Format*. 35. VoIPFuture, Hamburg: , 2008. – Documentation of the VAD file format

[Venners 2000] VENNERS, Bill: *Inside The Java Virtual Machine*. 2nd edition. McGraw-Hill Companies, 2000. – URL `http://www.artima.com/insidejvm/blurb.html`. – ISBN 978-0071350938

[VoIPVector Library Manual 2008] *VoIPVector Library Manual*. 275. VoIPFuture, Hamburg, 2008. – API Developer Guide - Library V1.2

[Wallingford 2005] WALLINGFORD, Ted: *Switching to VoIP*. O'Reilly, 2005. – ISBN 0-596-00868-6

[White Paper 2008] *VoIP Quality Analyse und Diagnose*. VoIPFuture, Hamburg, Januar 2008. – Technical White Paper, Revision 1.06

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 19.12.2008	
Ort, Datum	Unterschrift