Technische Hochschule Mittelhessen

Bachelor-Thesis

Stärkung des Beweiswertes von VoIP Aufzeichnungen durch digitale Signaturen

Hagen Lauer

Gießen, 31. Januar 2013

Betreuer: Prof. Dr. Michael Jäger
Gutachter: Dipl. Inf. Nicolai Kuntze

Inhaltsverzeichnis

1 Einleitung

Zusammenfassung

Ziel dieser Arbeit ist die Entwicklung einer Bibliothek, die es erlaubt, mittels digitaler Signaturen, VoIP Gespräche rechtssicher und nichtabstreitbar aufzuzeichnen. Diese Bibliothek soll später in einen existierenden VoIP Client für Android-Smartphones integriert werden können.
Grundlage dieser Umsetzung ist die Entwicklung eines Algorithmus durch das Fraunhofer Institut für sichere Informationstechnologie (SIT). Dieser Algorithmus basiert auf der Idee, ein Gespräch nicht wie bisher üblich am Anfang und Ende zu signieren, sondern das Gespräch in seinem Verlauf zu dokumentieren und kontinuierlich zu signieren. Erreicht werden Dokumentation und kontinuierliche Signaturen durch das Verketten von Hashes und Signaturen der gesendeten und empfangenen Datenpakete des Gesprächs.
Technologische Grundlage dieses Systems wird die Android Plattform sein, ebenso sind SIP und RTP Schlüsseltechnologien für VoIP-basierte Kommunikation. Basis der beigelegten Implementierung wird der später beschriebene Algorithmus sein. In einem Fallbeispiel wird die entstandene Bibliothek exemplarisch integriert.

1.1 Thematik

Meiner eigenen Motivation voranstellen möchte ich einige Ergebnisse aus aktuellen Marktforschungen zum Thema Voice over IP (VoIP) und Internet Telefonie im Allgemeinen.
Laut dem Jahresbericht der Bundesnetzagentur (siehe Bundesnetzagentur 2010, Seite 82 ff.) aus dem Jahre 2010 waren 2005 kaum mehr als 1% der verwendeten Vermittlungstechnologien IP-basierte Netze, 2006 waren es immerhin schon 5% und bis 2010 stieg der Anteil auf 22%. Was auch recht interessant anzusehen ist: Während die Deutsche Telekom AG noch eher moderat im Einsatz mit IP-basierten Verbindungen war und zum größten Teil klassische Telefonsysteme verwendete, so waren es die kleinen Anbieter und Wettbewerber die mit ca. 13% Marktzuwachs hauptsächlich auf DSL und VoIP setzten. Schaut man sich dann den Jahresbericht der Bundesnetzagentur von 2011 (siehe Bundesnetzagentur 2011, Seite 66 ff.) an, bemerkt man, dass der Anteil der Deutschen Telekom AG merklich unter den der Wettbewerber gefallen ist. Dazu ist auch der Gesamtanteil der VoIP Zugänge gestiegen. Rund 23% der Gesprächsminuten wurden diesem Bericht zu Folge über IP-basierte Netze geführt. Dies ist ein Bild, das sich bereits zur Jahrtausendwende abgezeichnet hat. Glaubt man den Marktforschern von Infonetics [1], dann stieg in den USA der Anteil der IP-Telefonie von 2010 bis 2011 um 33% an und für 2012 wird ein noch größerer Anstieg erwartet. Dies dürfte wohl damit zusammenhängen, dass der Ausbau von Breitbandverbindungen weltweit wichtiger wird.
Viel interessanter als private Zugänge der Haushalte sind jedoch die geschäftlichen Möglichkeiten. Die IP-basierte Telefonie ist beherrschend im geschäftlichen Bereich und sie hat auch immense Vorteile was die Verwaltung und Ressourcenausnutzung betrifft, ein typischer Vorteil den alle paketbasierten Systeme gegenüber den klassischen Techniken haben.
Typische Bedingungen die gerade in der geschäftlichen Nutzung von großer Bedeutung sind:

- Vertraulichkeit: Die verbundenen Partner sollten davon ausgehen können, dass sie in ihrem Gespräch unbehelligt von Dritten sind und das Gespräch somit nur gewünschten Parteien zugänglich ist. FhI-SIT (2005)
- Integrität: Die Teilnehmer eines Gesprächs sollen in der Lage sein zu beweisen, dass die unter Umständen aufgezeichneten Gespräche exakt so vorliegen wie sie erzeugt wurden.FhI-SIT (2005)
- Authentizität: “Authentizität der Daten besagt, dass die Daten tatsächlich von dem vermeintlichen Kommunikationspartner stammen.” - FhI-SIT (2005). “Authentizität des Kommunikationspartners besagt, dass der Partner derjenige ist, der er vorgibt zu sein” - FhI-SIT (2005)

[1] http://www.infonetics.com/pr/2012/VoIP-UC-Services-Market-Forecast-and-SIP-Trunking-Survey-Highlights.asp

- Verbindlichkeit: Hier kommt die Nichtabstreitbarkeit (siehe Glossar) ins Spiel, denn man will möglichst beweiskräftige Dokumente mit einem aufgezeichneten Gespräch erzeugen und somit eine ganz neue Ebene an Verbindlichkeit und Flexibilität in die Geschäftsprozesse bringen. VoIPSEC (2005)

Diese vier Punkte sind Ziele eines rechtlich Verbindlichen Gesprächs über VoIP mit kryptografischen Methoden.
Diese Ziele finden nicht nur Anwendung in Telefonie zwischen festen Telefonen in Haushalten oder Unternehmen, sondern auch - und in den letzten Jahren besonders - in Smartphones. IP Telefonie hat hier z.B. mit eigenen SIP-Servern große Vorteile: Man könnte seinen Mitarbeitern Smartphones mit mobiler Internetanbindung, bzw. WLAN Verbindung für Hausinterne mobile Geräte zur Verfügung stellen und so ein sehr effizientes Netzwerk für Kommunikation aufbauen. Dazu aber mehr in den möglichen Szenarios.
Zu den Zielen sei jedoch gesagt: Sie sind nicht nur, viel mehr nicht allein, auf kryptografischer Ebene durch Algorithmen zu lösen. Betrachtet man den Punkt "Integrität", so muss man das System als ganzes beobachten. Angriffe auf Server und Proxys, die z.B. die Echtheit des Gesprächspartners in seiner Person oder Position sichern sollen, sind immer möglich und stellen ein Grundrisiko dar. Diese Grundrisiken beginnen schon bei trivialen Dingen, wie verlorene oder gestohlene Passwörter: Passiert dies ohne Wissen des Opfers, ist das gesamte VoIP-System korrumpiert. Für einen Angreifer bieten sich über solche Sicherheitslücken dann zahlreiche Optionen für Angriffe:

- Er gibt sich als vermeintlicher Empfänger sensibler Daten aus.
- Er ändert den Status eines Endgerätes und verursacht so eine permanentes Besetztzeichen.
- Er blockiert spezifische Gespräche mit einem Auflegen-Signal.
- Je nach System wären auch Replay Attacken denkbar, die dann sogar auf Sprachebene noch unbemerkt wären.

Der Punkt Vertraulichkeit ist mit Sicherheit eines der größten Ziele der VoIP Telefonie und gleichzeitig auch ein großer Kritikpunkt, der die Systemsicherheit betrifft. In klassischen Telefonanlagen müsste sich ein Angreifer, bevor er irgend einen Schaden anrichten kann, zunächst physisch Zugang verschaffen VoIPSEC (2005), nicht aber in der VoIP Telefonie. Was in den oben genannten Zielen nicht genannt wurde, ist allerdings essenziell für verschiedene Situationen: Verfügbarkeit. Sowohl Gesprächsquallität als auch die Verfügbarkeit des Telefondienstes selbst sind zu jedem Zeitpunkt erforderlich. Dies gilt für private, öffentliche, dienstliche und geschäftliche Systeme.
Diese Arbeit wird sich nicht mit Infrastrukturellen Maßnahmen befassen, sondern

auf kryptografische Methoden vertrauen und zeigen, welche Aspekte dadurch erfüllt oder gewährleistet werden können. Insbesondere ist das Ziel, Gespräche so zu signieren, dass sie am Ende ein beweiskräftiges Dokument ergeben und damit Nichtabstreitbarkeit gewährleisten.

1.2 Motivation

Die Nachfrage was beweiskräftige Dokumente im VoIP Bereich betrifft ist groß. Betrachten wir zunächst drei der typischsten Szenarios, die ich hoffe mit mobilen Endgeräten abdecken zu können:

- Privat zu Business: Häufig werden in oder während Telefonaten mehr oder weniger verbindliche Zusagen getroffen, die jedoch aus genannten Gründen rechtlich angezweifelt werden können. Würden hier die Klienten gegenseitig recht einfach und ohne Behinderungen ein Gespräch signieren und aufzeichnen können, so hätten beide Seiten die Möglichkeit rechtssichere Aufzeichnungen im Falle eines Streits vorweisen zu können.

- Business intern: Ob Telefonie im Haus selbst oder Verbindungen mit außerhalb arbeitenden Mitarbeitern, geschäftliche Kommunikation ist der Schlüssel in jedem Unternehmen. Es gibt hier beispielsweise den Anwendungsfall, dass hausintern Smartphones zur Kommunikation innerhalb des eigenen Netzes stattfinden sollen. Das sichert zwar Angriffe von Extern gut ab (LAN/W-LAN), bietet aber dennoch keine Möglichkeit verbindliche Telefonzusagen zu treffen, welche hinterher nicht angezweifelt werden können. Der zweite Fall ist, dass Mitarbeiter extern oder über verschiedene Standorte verteilt kommunizieren. Je nach dem wie gut die Vernetzung ist, bieten sich VoIP Clients auf Smartphones sehr gut an, um Firmengespräche zu führen (geringe Koste, gute Nutzung und im besten Fall gleicher Service wie über reguläre Mobilfunknetze). Auch hier ist teilweise gesetzlich verlangt, dass bestimmte Telefonate, wenn nicht sogar alle, beweisbar aufgezeichnet werden müssen. Beispielsweise passiert es im Bankengeschäft sehr häufig, dass wichtige Abwicklungen schnell geschehen müssen. Das Telefon ist dabei das Mittel der Wahl.

- Ein dritter Punkt ist gerade in Deutschland sehr aktuell: Digitalfunk. Beispielhaft gesehen, kann das nachfolgend erläuterte Verfahren in Abschnitt 2.4 auf verschiedene technologische Grundlagen übertragen werden. Von Behörden genutzer Digitalfunk muss aufgezeichnet werden [1]. Die Behörden in Deutschland (Polizei, Feuerwehr, Bundeswehr, ...) kommunizieren als eine große Ausnahme der Europäischen Länder noch fast ausschließlich mit analogen Funkgeräten.

[1] `http://www.polizei.schleswig-holstein.de/cae/servlet/contentblob/579216/publicationFile/leistungsmerkmale-digitalfunk.pdf`

> Grund hierfür ist unter anderem auch ein mangelhaftes Konzept für die Forderung nach sicheren und gerichtlich verwendbaren Aufzeichnungen der Kommunikation.

Genau hier setzt das Fraunhofer SIT mit einem Algorithmus an, denn die folgende Portierung auf ein Smartphone OS wie Android stellt nicht das volle Ausmaß der Anwendungsmöglichkeiten des Prinzips dar.

1.3 Organisation

Kapitel 1, welches Sie gerade lesen, befasst sich lediglich mit der Einführung in das Thema VoIP, einigen Aspekten der Sicherheit und Anwendungsbeispielen der entstehenden Arbeit. Hinzu kommt noch eine Einordnung der vorhandenen Technologien und deren mögliche Verwendung in dieser Arbeit.
Kapitel 2 befasst sich mit den verwendeten Technologien, die hinter dem großen Begriff VoIP stehen, sowie mit der Funktionsweise des in dieser Arbeit umgesetzten Algorithmus.
In Kapitel 3 werden Konzepte für eine Implementierung des zuvor beschriebenen Algorithmus vorgestellt. Anschließend folgt in Kapitel 4 ein Fallbespiel, in dem gezeigt wird, inwiefern sich die ausgearbeiteten Konzepte in einem "echten" Client nachträglich einfügen lassen.

1.4 Stand der Technik

Bisher gibt es nur wenige Anbieter, die eine Lösung der Problematik dieser Arbeit anbieten. Einer davon ist Andtek. *Andtek Phone Recorder*: Diese Lösung ist zwar nicht ganz dem entsprechend, was in dieser Arbeit versucht wird, jedoch klingen einige Schlagwörter sehr interessant. Andtek wirbt und schreibt in Artikeln und Berichten [1] [2]

> München/Hallbergmoos, 24. September 2007 – Mit „AND Phone Recorder" bietet der Unified-Communications-Spezialist Andtek eine Softwarelösung an, mit der sich IP-basierende Telefonate (Voice over IP, VoIP) aufzeichnen lassen. Verschlüsselt und durch eine Signatur vor Manipulation geschützt sind die aufgezeichneten Gespräche als Original verifizierbar und haben zum Beispiel bei Drohanrufen Beweiskraft vor Gericht.

Wer allerdings nach vergleichbaren Urteilen sucht, die diese Technologie bestätigen, der sucht vergebens. Auch habe ich keinen konkreten Anwendungsfall gesehen, abgesehen von einer eher schwammigen Zeichnung die in [2] abgebildet war:

[1] `http://www.andtek.com/de/communications-company-news-125_pressrelease_0407.html`
[2] `http://www.andtek.com/get-datenblatt_andphone_booklet-460.pdf`

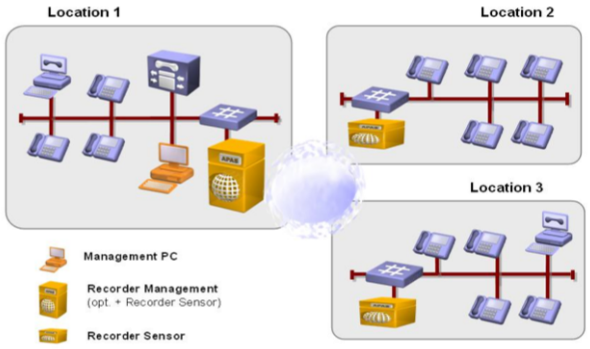

Abbildung 1.1: Aufbau von Andtek's Phone Recorder.

Dabei fällt beim weiteren Lesen der Dokumente schnell auf, dass dieses System darauf basiert, dass alle gewünschten Gespräche innerhalb einer Infrastruktur aufgezeichnet und damit mitgehört werden. Das Signieren passiert an einer der Recording Stationen und nicht auf der Seite des Clients. Dieses System bietet nach meinem Informationsstand nicht die Flexibilität, die wir gewährleisten wollen. Andtek nimmt hier massiv Einfluss auf die Geräte-Infrastruktur. Wir, also ich und das Fraunhofer SIT, wollen lediglich eine Applikation modifizieren um damit auf möglichst beliebigen Infrastrukturen arbeiten zu können. Außerdem beschreibt Andtek, dass ihr System eine maximale Anzahl an Gesprächen hat, die gleichzeitig ablaufen können. Dies ist im Hinblick auf ihre Architektur nicht verwunderlich: Das serverartige Gebilde wird vermutlich an seine Leistungsgrenzen stoßen. Hinzu kommt der nette Zusatz, dass lediglich Cisco Telefone unterstützt werden. Von mobilen Geräten ist dort nie die Rede.
Dies ist bisher der einzige Konkurrent, der einen kleinen Einblick geboten und dennoch nicht wirklich überzeugt hat. Ob und wo das System im Einsatz ist, konnte ich nicht herausfinden. Jedoch sind die Kundenkreise, ähnlich wie in der Motivation dargestellt, Unternehmen (vorzugsweise Banken), die strenge Auflagen haben, was die Qualität und Rechtssicherheit ihrer Aufzeichnungen angeht.

2 Grundlagen zum Thema

Dieses Kapitel wird im Grunde genau den Beginn meiner Arbeit an diesem Projekt beschreiben: Ich habe alles an Schlagworten gesammelt was mir bei der Bearbeitung und Implementierung begegnen würde und diese dann in den entsprechenden Artikeln, RFC's und Fachliteraturen, sofern das denn nötig war, nachgelesen und Informationen zusammengetragen. So sollte eine solide Basis für die weiteren Arbeiten geschaffen werden.

2.1 Voice over Internet Protocol - VoIP

VoIP, Voice over IP, unterscheidet sich deutlich von herkömmlicher Telefon- und Verbindungstechnologie, die noch immer vorherrschend ist. VoIPSEC (2005)
Wie der Name schon sagt, basiert die Technologie auf der Sprachübertragung über das Internet Protokoll. Hier zeigt sich auch schon der drastischste Unterschied: Ruft man sich das Schichtenmodell [1] wieder ins Gedächtnis, dann kann man gut nachvollziehen, dass traditionelle Telefonie auf Schicht 2 (Data Link Layer oder Übertragungsschicht) mit fester Bandbreite arbeitet. IP Telefonie hingegen arbeitet verbindungslos und paketorientiert auf Schicht 3. Damit gehen auch typische Probleme, wie der Verlust von Paketen, die Verzögerung im Empfang der Pakete (Delay) und unterschiedliche Verzögerungszeiten der einzelnen Pakete (Jitter) einher. Ähnlichkeiten bestehen darin, dass die Signalisierung (siehe SIP) der Gespräche sowohl bei traditionellen Systemen (z.B. ISDN) getrennt vom eigentlichen Gespräch stattfinden (Out-of-Band). Jedoch hatte man herkömmlich einen festen Sprachkanal auf dem dann das Gespräch stattfand, das ist bei VoIP nicht so. Ein VoIP Gespräch basiert rein auf Zieladressen des Gesprächspartners und hat damit keinen bestimmten Kanal oder Übertragungsweg. Aus diesem Konzept ergibt sich, dass einzelne VoIP Pakete immer auch eine Zieladresse enthalten müssen und daher auch prinzipiell über verschiedene "Wege" in einem Netz ihr Ziel erreichen. Außerdem ergibt sich für IP Telefonie zwangsweise ein Overhead der verwendeten Protokolle, z.B. UDP (User Datagram Protocol) in der Transportschicht sowie RTP (Real-Time Protocol) und SIP (Session Initiation Protokol) die bis in die Anwendungsschichten reichen. IP selbst, als Teil der Schicht 3, macht aber schon klar, dass es hier einen Overhead gegenüber den herkömmlichen Systemen aus Schicht 2 geben muss.
Sicherheitstechnisch ergeben sich auch starke Unterschiede, gerade was die Bedrohungen angeht. Herkömmliche Telefonanlagen sind physikalisch adressierbar, ihre Adresse, also der Endpunkt, ist fix und ändert sich nicht. Der Teilnehmer ist somit identifizierbar. Bei IP-baiserten Netzen ist dies nicht so, die Endpunkte sind flexibel und müssen explizit, bevor ein Gespräch stattfinden kann, authentifiziert werden.

[1] `http://en.wikipedia.org/wiki/OSI_model`

Man sollte auch immer beachten, dass ISDN Endgeräte üblicherweise relativ primitiv gestaltet sind. VoIP Endgeräte und deren Software basieren häufig auf komplexeren Betriebssystemen, hier Android. Dies bietet, wie jede verwendete Software, ein erhöhtes Risikopotenzial an, aber eben auch die gewünschte Flexibilität für gezielte Erweiterungen im Bereich der Sicherheitsmechanismen. VoIPSEC (2005)

2.2 SIP - Session Initiation Protocol

Das Session Initiation Protocol ist ein Signalisierungsprotokoll, das heute im Bereich der IP Telefonie große Bedeutung hat. Signalisierung ist hierbei die Hauptaufgabe. In unserem Falle wäre dies konkret das Steuern des Telefonats: Ankündigen eines Gesprächswunschs, die Annahme des Gesprächs und das Beenden des Gesprächs zwischen, der Einfachheit halber, zwei Parteien. SIP wurde erstmals 1996 von Henning Schulzrinne und Mark Handley definiert und ist im heutigen RFC3261 in der aktuellsten Version definiert. SIP ist ein Textbasiertes Protokoll des Application Layers und kann somit auf verschiedenen Transportprotokollen implementiert werden. Typischerweise sind dies UDP und TCP (Transmission Control Protocol). Zu bemerken ist, dass SIP durch seinen simplen Aufbau im Vergleich zu H.323 [2] Anwendung in verschiedensten, auf Kommunikation basierten, Systemen hat. Dazu zählen VoIP Geräte, Instant Messenger, Videokonferenz-Tools oder auch verteilte Computerspiele und Applikationen mit ähnlichen Szenarios. SIP Adressen ähneln Email-Adressen: hagen.lauer@sipserver. Der SIP Body enthält im Grunde alle Informationen, die für einen Austausch eines Medienstroms wichtig sind (IP, Port). Auf einige Spezialfälle dieser "Kontaktinformationen" gehe ich später noch ein. Vereinfacht dargestellt besteht die SIP Infrastruktur aus Clients und Servern. Ein Client spiegelt dabei einen Account auf einem Server wider. Ein VoIP Gerät meldet sich nun mit einer Register-Anfrage, seiner Adresse und einem Passwort auf diesem Server an. Der Server hat nun die Kontatkinformationen des Gerätes und weiß mit welchem Account dieses arbeitet. Möchte das Gerät/der Nutzer nun ein anderes Gerät/einen anderen Nutzer anrufen, so geschieht dies über eine Invite Nachricht von A (A ruft B an) an den Server, dieser ermittelt ob B erreichbar ist (B muss ebenfalls registriert und angemeldet sein). Ist B registriert und gerade angemeldet sendet der SIP Server B einen Anfrage für ein eingehendes Gespräch. Nimmt B an, erhalten A und B die Adressen (IP) ihres Gegenübers und können darauf basiert einen Datenstrom austauschen (üblicherweise Audio und Video ströme). Beendet und abgebaut wird das Gespräch auch wieder von einer Seite über den Server zur anderen. Das ist eine stark vereinfachte Darstellung eines Telefonats mit SIP zur Signalisierung. Etwas detaillierter ist Abbildung 2.1:

[2] `http://www.itu.int/rec/T-REC-H.323`

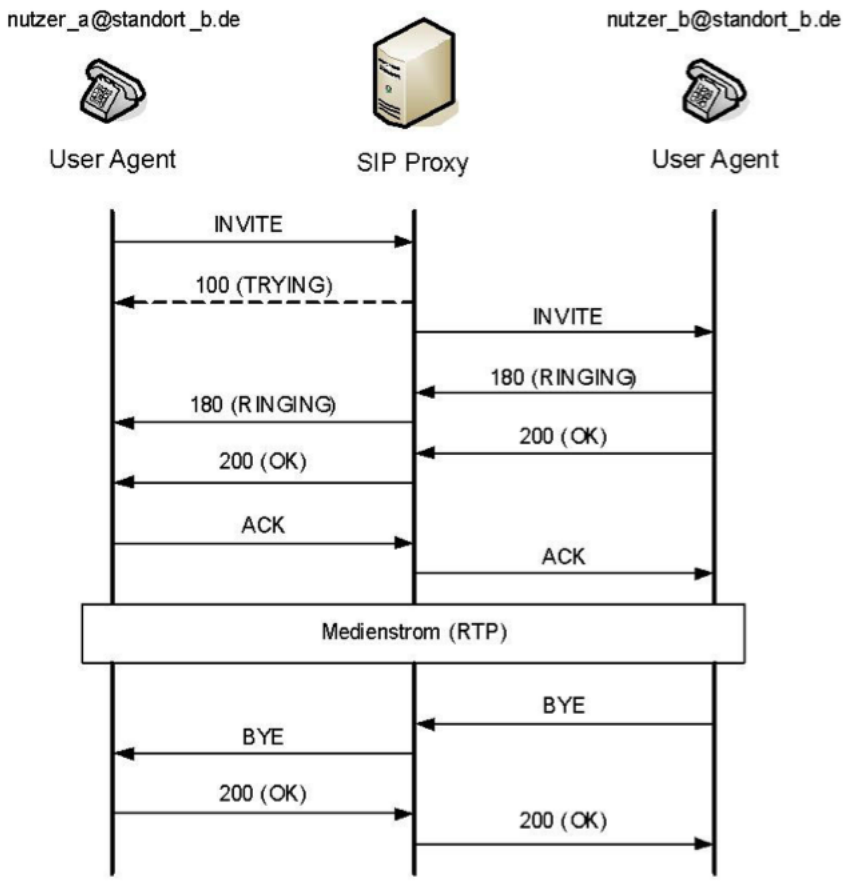

Abbildung 2.1: “Sip-Call” Beispiel, Quelle: VoIPSec - BSI, 2005

Schön zu sehen ist in Abbildung 2.1, dass SIP grundsätzlich aus Request und Response Nachrichten besteht. Die Request und Response Typen ähneln denen von HTTP dabei sehr stark. Für mich war dieses Protokoll sehr leicht verständlich und es ist auch im Gebrauch und in der Implementierung sehr durchsichtig. Was man an dieser Stelle bemerken sollte ist, dass es etwas wie End-to-End Sicherheit in SIP nicht gibt. Man kann nur durch TLS (Transport Layer Security) zwischen den einzelnen Netwerkknoten eine sichere Übertragung erreichen, nicht aber von einem SIP Client zum anderen. Hier ist eine Liste der SIP Request Nachrichten:

2.2.1 SIP Requests

Request	Beschreibung	Definitionsstelle
INVITE	Ein Client soll zu einer Sitzung, üblicherweise ein Anruf, eingeladen werden	RFC 3261
ACK	Bestätigung des Empfangs eines einer Antwort auf ein gestellte INVITE Anfrage	RFC 3261
BYE	Beendet die Sitzung kann von z.B. Anrufer oder Angerufenem gesendet werden	RFC 3261
CANCEL	Schließt eine zuvor gestellte Anfrage	RFC 3261
REGISTER	Soll den Server dazu veranlassen die im Header mitgesendete Adresse zu registrieren	RFC 3261
PRACK	Provisorisches ACK mit dem Unterschied zu ACK, dass es zum PRACK ein spezielle SIP Response gibt	RFC 3262
SUBSCRIBE	Fordert eine Registrierung für den Empfang von Events an	RFC 3265
NOTIFY	Benachrichtigt einen Registrierten Client von einem Event	RFC 3265
PUBLISH	Sendet ein Event zum Server	RFC 3903
INFO	Sendet weitere Informationen während der Sitzung, die diese aber nicht in ihrem Status verändern	RFC 6086
REFER	Ermöglicht die Rufweiterleitung (A und B telefonieren, B muss aber mit C sprechen, nur A kennt C, dann kann A mit einem REFER B den Anruf zu C ermöglichen)	RFC 3515
MESSAGE	Versendet eine Nachricht (Instant Message) via SIP	RFC 3428
UPDATE	Modifiziert die Sitzungsinformationen, eingesetzt wird dies z.B. in der Phase während ein INVITE gesendet aber noch nicht akzeptiert wurde	RFC 3311

2.2.2 SIP Respones

SIP Responses sind nach RFC 3261 etwas, was es so nicht gibt, genau wie SIP Fehlercodes oder sonstige Bezeichnungen. Der korrekte Begriff ist SIP Status Codes Rosenberg et al. (2002). Natürlich treten die Status Codes üblicherweise als Folgeereignisse auf Requests auf und daher werde ich sie auch als Antworten bezeichnen.
Wie dem auch sei, es gibt viele dieser Status Codes, sehr viele. Ich werde hier nur

die Klassifizierung der einzelnen Codes vorstellen und Beispiele dazu. Jedoch kann man diesen Satz an Codes jederzeit erweitern, um genaueren Ansprüchen gerecht zu werden, ohne das Protokoll selbst zu stören. Wer dort tiefer einsteigen will, der sollte sich generell an die RFC's halten:

- Vorläufige/Informative Antworten: Diese Status Codes werden versendet, bevor eine Sitzung mit dem möglichen Gesprächspartner aufgebaut werden konnte und helfen dem Client seinen Vermittlungsstatus nachzuvollziehen:
  - 100 Trying: Die Suche nach einem Kontakt dauert lange, jedenfalls lange genug, sodass der Server diesen Status Code den Client wissen lässt.
  - 180 Rining: Der Kontakt ist gefunden und hat das INVITE erhalten, jetzt steht nur noch die Antwort darauf aus.
- Antworten, die erfolgreich bearbeiteten Requests folgen: Wie der Name schon sagt, werden diese Status Codes zum Signalisieren eines erfolgten Sitzungsaufbaus verwendet.
  - 200 OK: Welche Anfrage auch gesendet wurde, sie war erfolgreich.
  - 202 Accepted: Request wird bearbeitet, eine Antwort steht aber noch aus.
  - 204 No Notification: Request wurde erfolgreich bearbeitet, jedoch wird die entsprechende Antwort darauf nicht mehr kommen.
- Weiterleitungs Antworten: Diese richtig genannten Redirect Responses folgen üblicherweise auf INVITE Requests und deuten dem Client an, dass hier mehr für den Verbindungsaufbau getan werden muss:
  - 300 Multiple Choices: Die angegebene Adresse im INVITE kann zu mehreren Endpunkten aufgelöst werden, es wird empfohlen, dass der Client einen bevorzugten Endpunkt spezifiziert und seine Anfrage dorthin erneut stellt. Üblicherweise sollten also Antworten mit diesem Code eine Auswahl an möglichen Endpunkten bieten, aus denen der Client dann entweder automatisch wählt oder den Benutzer entscheiden lässt.
  - 301 Moved Permanently: Der angefragte Nutzer der Adresse ist unter dieser nicht mehr zu finden. Die neue Adresse sollte dem Client geliefert werden, woraufhin er dann seine Anfrage erneut senden kann. Außerdem sollten die Kontaktinformationen auf Client Seite aktualisiert werden.
- Client Fehler Antworten: Hier gibt es ein breites Spektrum an Codes, die aber alle auf Fehler in der Kommunikation zwischen Server und Client oder Client und Client zurück zu führen sind:
  - 400 Bad Request: Fehlerhafte SIP Anfrage, hier ist das Spektrum an Fehlern sehr groß, ähnlich wie bei HTTP.

  - 401 Unauthorised: Autorisierung ist fehlerhaft, Client ist nicht befugt.
  - 402 Payment Request: Für folgende Aktionen sind Zahlungen nötig, es sind Zahlungen ausstehend.
  - 404 Not Found: Was auch immer angefragt wurde konnte vom Server nicht aufgelöst werden.
  - 410 Gone: Teilnehmer ist nicht mehr erreichbar, das sollte dem Client vom Server mitgeteilt werden, wenn das Ende gegenüber nicht mehr erreicht werden kann. Es wird keine BYE Meldung geben, die dann zum Rufabbau genutzt werden kann.
  - 484 SIP Address Incomplete: Klassischer Fehler, der auf Client Seite entsteht und der dann vom Server signalisiert wird.

  Im Fall dieser Fehler sind fast alle der Codes von 400-499 bereits spezifiziert und damit auch schon gut ausgeschöpft, Ich erspare hier die weiteren Aufzählungen der Fehlercodes im Bereich 4xx.

- 500 Server Fehler: Diese Fehlercodes signalisieren grundsätzlich Fehler auf Seite des Servers, der Client wird hier nur bedingt reagieren können.
  - 500 Interner Fehler: Hier kann ein Client nichts ausrichten, der Server hat das Problem.
  - 501 Not Implemented: Die empfangene SIP Anfrage ist vermutlich formal korrekt, kann aber dennoch nicht vom Server bearbeitet werden, da deren Behandlung nicht implementiert ist.
  - 504 Timeout: Timeout eines Servers.
  - 505 Version Not Supported: Der Server unterstützt nicht die angegebene Version des Protokolls dem die SIP Nachricht entspricht. Da SIP abwärtskompatibel aufgebaut ist, dürfte das auf veraltete Versionen auf dem Server hinweisen.

  Die Fehlercodes des Servers sind insgesamt recht übersichtlich, man kann also davon ausgehen, dass hier je nach Server noch spezielle Codes hinzu kommen.

- 600 Globale Fehler: Diese umfassen verschiedenste Fehler, ich werde hier die im RFC 3261 Definierten darstellen:
  - 600 Busy Everywhere: Der gewünschte Teilnehmer ist an allen Endgeräten nicht erreichbar, alle Endgeräte des Teilnehmers sind belegt.
  - 603 Declined: Der gewünschte Teilnehmer hat den Kontaktversuch abgelehnt.
  - 604 Does Not Exist Anywhere: Benutzer, man sollte Account sagen, ist nicht auffindbar / nicht existent.

– 606 Unacceptable: SIP Anfrage ist nicht akzeptierbar, bzw. unzulässig.

In Abbildung 2.2 kann man eine Beispielhafte SIP Nachricht, in diesem Fall ein klassisches INVITE, sehen, samt allen üblichen Parametern einer solchen Nachricht. Der Body dieser Nachricht enthält in diesem Beispiel praktischerweise eine SDP (Session Description Protocol) Nachricht, worauf ich gleich noch zu sprechen kommen werde.

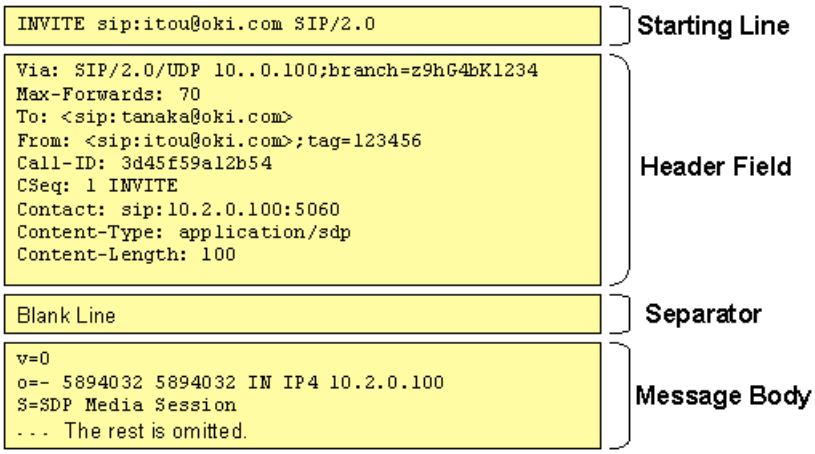

Abbildung 2.2: SIP INVITE Nachricht

2.2.3 Session Description Protocol

SDP ist ein Protokoll um Sitzungen genauer zu beschreiben, als dies in den SIP Headern möglich ist. Oft sind diese SDP Nachrichten der Message Body einer SIP Nachricht, die im Grunde auch nur aus einem Header mit leerem Body bestehen könnte. Angedeutet wird der enthaltene Body in den Header Feldern “Content Type“ und “Content Length“. Hier sind die Verwendungen wieder sehr breit gefächert, Microsoft nutzt diese Message Bodys z.B. um für ihre Instant Messenger Textnachrichten zu übertragen. SDP ist dabei sehr simpel aufgebaut: Es ist rein Textbasiert. Es besteht aus <key>=<value> Paaren. Jedes dieser Paare wird durch ein *newline* getrennt. Eine SDP Nachricht hat mindestens 1 aber bis zu 3 Teilen, jedenfalls nach RFC4566. Der erste Teil sind Sitzungsinformationen, Session Descriptions, der zweite sind üblicherweise Zeitangaben, Time Descriptions, und der dritte Teil besteht aus Medien Informationen, Media Descriptions. Beim Parsen erkennt man die Teile oder Abschnitte allerdings nur an den entsprechenden <key>-Tags und kann dementsprechend die Information der SDP Nachricht klassifizieren. Wie auch SIP kann SDP sehr leicht erweitert werden: Wollten wir z.B. einen Hash von was auch immer, vielleicht der einer Signatur, unserer SDP Nachricht anhängen, so könnten wir einerseits ein bereits definiertes <k> Feld für einen verwendeten Schlüssel verwenden, oder das Protokoll um ein <h> in der Nachricht erweitern um dort

<h>=*unser Hash* anzuhängen. Dies würde andere Parser nicht weiter stören, sofern sie wiederum nicht selbst Parameter unter <h> vermuten. Die Gefahr, von einzelnen Implementierungen der Parser "missverstanden" zu werden, besteht aber immer, jedenfalls immer dann, wenn man sich außerhalb der Standards bewegt. Hier sei schon bemerkt, dass sich SDP grundsätzlich dazu eignen würde, Schlüssel unter den Parteien auszutauschen. Diese Schlüssel werden später zur Signatur durch den Algorithmus beitragen. Typische Felder, in RFC 4566 definiert, sind

- **v** - die Version des verwendeten SDP
- **s** - ein Sitzungsname
- **i** - Sitzungsinformationen, eine lesbare (human readable) Beschreibung, z.B. Inhalt, der Sitzung in Klartext
- **e** - Email Addresse des Senders, j.doe@example.com (Jane Doe), als vorgeschlagene Formatierung
- **p** - Telefonnummer , +49 01234-445566, als Standard für Telefonnummern nach ITU-T E.164
- **c** - Verbindungsinformationen im Format <nettype> <addrtype> <connection-address>, IN (für Internet) IP4 227.5.667.2
- **k** - Schlüssel zum Dechiffrieren, <method>:<encryption key>
- **a** - Attribute, hier sollte SDP eigentlich erweitert werden: a=<attribute1>=<value1>, ..., <attributeN>=<valueN>, oder einfach a=<attr1>, ..., <attrN>
- **m** - Beschreibung des gewünschten Medienstroms, m=<media> <port>/<number of ports> <proto> <fmt>, z.B. m=video 49170/2 RTP/AVP 31. Dies bedeutet Video-Daten, auf Port 49170, 49171 des einen RTP/RTCP Paars und 49172,49173 des anderen RTP/RTCP Paars, RTP/AVP als Transport-Protokoll, 31 ist in diesem Falle der Payload Typ des RTP Protokolls.[3]

[3] `http://tools.ietf.org/html/rfc4566` Seite 34

```
.....................
v=0
o=root 4369 4369 IN IP4 192.168.1.200
s=session
c=IN IP4 192.168.1.200
t=0 0
m=audio 35302 RTP/AVP 18 3 97 8 0 101
a=rtpmap:18 G729/8000
a=fmtp:18 annexb=no
.....................
```

Abbildung 2.3: Beispielhafter Auszug aus einer SIP Nachricht mit SDP Nachricht im Body. Quelle: Oracle

2.3 RTP - Real-Time Transport Protocol

RTP habe ich im Verlauf des Dokuments schon häufiger erwähnt ohne es genauer zu erläutern: RTP steht für Real-Time Transport Protocol und ist ein sehr weit verbreitetes Protokoll wenn es um den paketbasierten Austausch von Datenströmen geht. Real-Time, also Echtzeit, ist etwas irreführend, wenn man Betriebssystem-Aspekte damit in Verbindung bringt. Real-Time heißt es nur, da es ermöglicht einen Datenstrom zu versenden und ihn entsprechend der echten zeitlichen Reihenfolge des Stroms wieder zu rekonstruieren (Schulzrinne et al. (2003)). Bei paketbasierten Systemen ist das ein gewisses Problem, da nicht garantiert ist, dass wenn Paket a vor Paket b versendet wurde, a überhaupt und wenn dann auch vor b ankommt. Für einen Audio- oder Videostrom ist dies allerdings wichtig. Wollte man den Aufbau von RTP in einem Satz beschreiben, so hieße er bestimmt: RTP basiert auf der Idee eines Paketes mit informativem Header, welche die im Paket selbst enthaltene Fracht (Payload) beschreibt. Und genau so funktionieren RTP und RTP-Pakete, die über IP Netze versendet werden. RTP kann sowohl ober TCP (Transmission Control Protocol) als auch über UDP (User Datagram Protocol) übertragen werden, allerdings empfiehlt sich für Streams UDP. TCP ist ein Protokoll, dessen Augenmerk auf der Zuverlässigkeit der Datenübertragung liegt, Paketverluste werden abgefangen etc.. Dies geht auf Kosten der Geschwindigkeit der Übertragung. UDP hingegen ist verbindungslos und kümmert sich nicht um Paketverlust oder sonstiges, UDP ist schnell, aber eher unzuverlässig. Im Audio Video Bereich hat man sich auf UDP eingestellt, man rechnet mit Paketverlusten und Verzögerungen, z.B. sind kleinere Paketverluste im Audio Strom kaum spürbar und sollten sie es doch werden, greifen Codecs ein und versuchen diese Lücken im Strom dynamisch passend zu füllen. Alle diese Strapazen lohnen sich am Ende, da eine höhere Geschwindigkeit der Übertragung via UDP gegeben ist, auch bei geringeren Bandbreiten. Detaillierter zeigt Abbildung 2.4 den Header und Payload.
Weitere Erläuterungen zu Abbildung 2.4, alle Beschreibungen entsprechen den Definitionen aus RFC 3550:

RTP packet header

bit offset	0-1	2	3	4-7	8	9-15	16-31
0	Version	P	X	CC	M	PT	Sequence Number
32	Timestamp						
64	SSRC identifier						
96	CSRC identifiers ...						
96+32×CC	Profile-specific extension header ID						Extension header length
128+32×CC	Extension header ...						

Abbildung 2.4: Elemente, Formatierung und Größen eines RTP Headers gemäß RFC 3550

- **Version** Zeigt die Versionsnummer des Protokolls an, die aktuelle Version ist 2.
- **P Padding** Dieses Bit wird dazu genutzt um anzuzeigen, dass denm eigentlichen Payload noch Daten angehängt sind, die nicht Teil des Payloads sind (z.B. Verschlüsseliungsinformationen)
- **X Extension** Das Extension Bit zeigt an, dass ein zusätzlicher Header zwischen Header und Payload stehet.
- **CC (CSRC Count)** Hier wird die Anzahl der CSRC Identifier angegeben, CSRC ist weiter unten definiert.
- **M Marker** Das Makerbit ist nicht genauer definiert. Es wird anwendungsspezifisch genutzt um z.B. um dem Paket besondere Aufmerksamkeit zuzusprechen.
- **PT Payload Type** Der Payload Type beschreibt den Inhalt / das Format des Payloads mittels Codes / Nummern. Beispielsweise hat der Audio Codec PCMU die Nummer 0, GSM 3 und JPEG 26. Der Zahlenbreich kleiner 96 ist größtenteils Standardisiert, Codes zwischen 96 und 127 werden dabei dynamisch, also anwendugsspezifisch, verwendet.
- **Sequence Number** Die Sequenznummer wird mit jedem Paket hochgezählt, sie dient hauptsächlich zur Rekonstruktion einer Reihenfolge und dem Erkennen von Paketverlusten. Die Anwendung der Sequenznummer oder deren Verwendung ist jedoch nicht standardisiert und wird oft anwendungsspezifisch verwendet.

- **Timestamp** Der Timestamp dient der Wiedergabe eines Stroms mit entsprechenden Intervallen. Ein Audio Strom hat eine gewisse Abtastrate, nehmen wir mal 125μs also 8kHz an. Das ist eine gängige Abtastrate in der Digitalen Telefonie, genutzt u.a. im Codec PCMU. Der Zeitstempel dient zur Rekonstruktion eines solchen Stroms, dabei können zwei Pakete den gleichen Timestamp tragen, wenn sie zur selben Abtastung gehören (z.B. bei Video-Übertragungen).

- **SSRC** Synchronization source identifier dient zur Identifikation der Quelle des Stroms. Über eine Verbindung, können mehrere Ströme existieren, die dann mittels verschiedener Quell-Informationen (Source) unterschieden werden können.

- **CSRC** Contributing source ID zählt die Quellen auf, aus denen ein Strom besteht. Die Anzahl der Quellen wird im CC Feld angegeben.

- **Extension Header** Dieser Header ist optional. Die RFCs definieren ihn nicht genauer in seinem Aufbau, jedoch dient er generell der Erweiterung der Payload-Daten und deren Nutzung, die zusätzliche Informationen benötigt.

2.4 Der Algorithmus

Hier werde ich jetzt den benutzten Algorithmus beschreiben, als Referenz hierfür dient mir das vom Fraunhofer SIT veröffentlichte Dokument Kuntze (2005). Grundsätzlich behandelt der Algorithmus ein Thema, das so alt wie die Telefonie selbst ist: Verträge telefonisch zu schließen. Zusammen mit der paketbasierten Telefonie und kryptografischen Methoden, digitalen Signaturen, kann dies nun umgesetzt werden.

2.4.1 Vereinfachtes Prinzip und Szenario

Ich breche hier den Algorithmus auf ein Level herunter, das veranschaulichen wird, was der Algorithmus technisch leisten kann und muss. Dazu greife ich das Szenario “Bank kontaktiert Kunden” auf und erkläre die Abläufe:

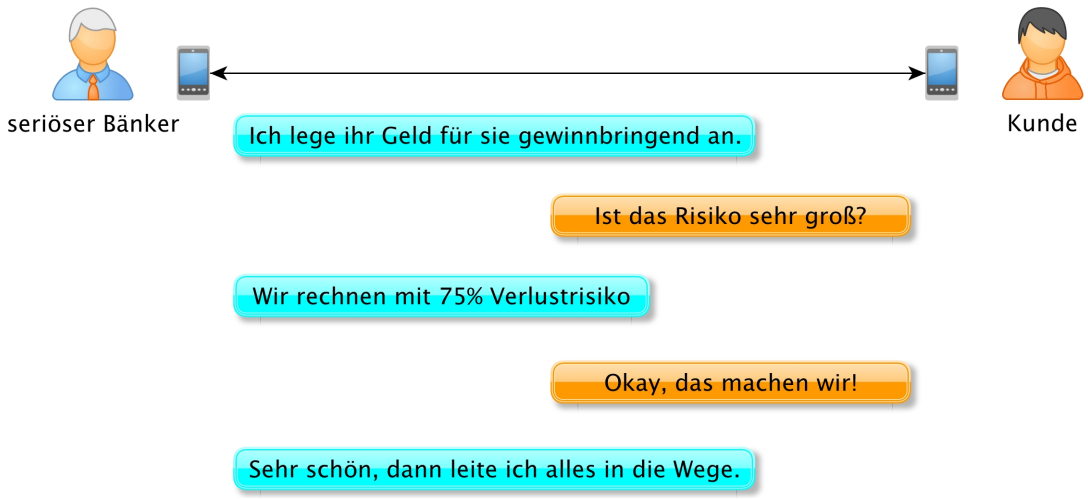

Abbildung 2.5: Alltägliches Szenario im geschäftlichen Umfeld

Wie in 2.5 zu sehen ist, wird hier telefonisch eine risikobehaftete Anlage von Geld mehr oder weniger besprochen. Gerade so kamen viele Banken in letzter Zeit in den Medien etwas in Verruf. Das Geld wird nun angelegt und der Verlustfall tritt ein. Siehe Abb. 2.6.

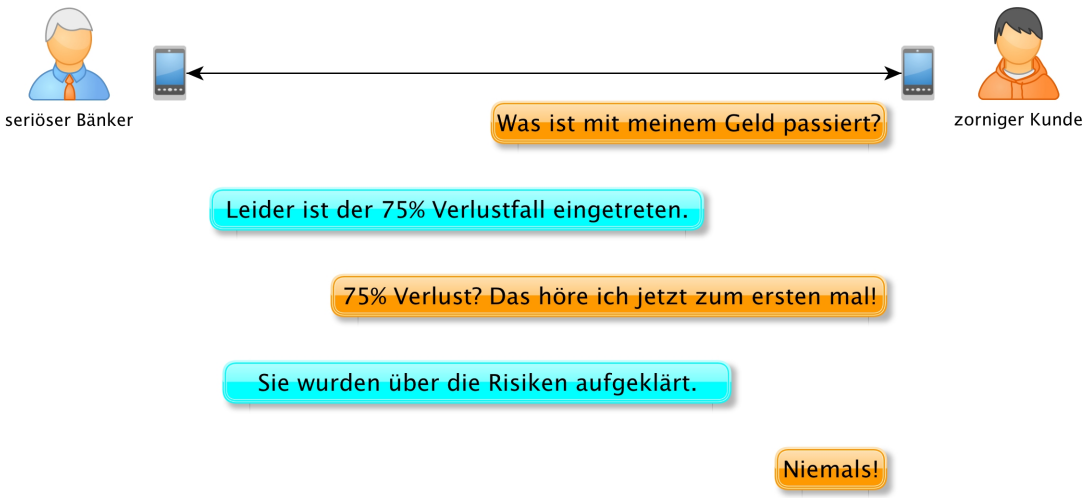

Abbildung 2.6: Problem der telefonisch geschlossenen Verträge

Wollte man nun dieses Gespräch aufzeichnen und gleichzeitig feststellen was die beiden Parteien genau verstanden haben, so kann man einen alten Grundsatz aus dem Bereich des analogen Funkverkehrs nutzen: Wiederhole was du verstanden hast. Zugegeben, dieser Vergleich hinkt, da die Wiederholung im Funk der Fehlerkorrektur dient. Jedoch bleibt der erste Schritt zur Fehlerkorrektur die Abfrage des Verstan-

denen. Ein erster logischer Schritt ist wohl, dass eine Seite, ich wähle die Bank, das Gespräch unterteilt in “gesagt” und “gehört”. So kann man ganz gezielt abfragen was das Gegenüber vom Gesagten verstanden hat und was vom Verstandenen tatsächlich gesagt wurde. (Figur 2.7)

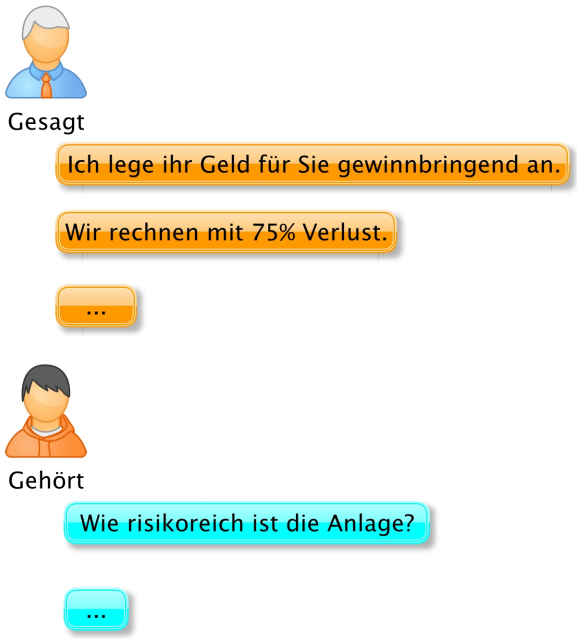

Abbildung 2.7: Aufteilung des Gesprächs in der Praxis

Anschließend muss man sich entscheiden, welche der Seiten nun die archivierende Seite ist, wer also von beiden Parteien im Gespräch selbiges aufzeichnet und abspeichert. Ich wähle hier genau wieder die Bank, da dieser Fall im Grunde Kern der Anwendungsbereiche ist.
Nun kann eine Seite damit beginnen abzufragen, was das Gegenüber gehört und gesagt hat. (Figur 2.8)

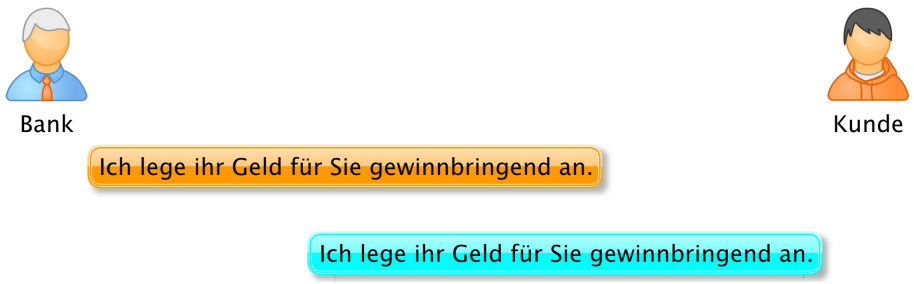

Abbildung 2.8: Wiederholen was gesagt wurde

Das sieht noch recht praktikabel aus, es wird aber völlig absurd, wenn man nun versucht das Verständnis in die andere Richtung zu überprüfen. (Figur 2.9)

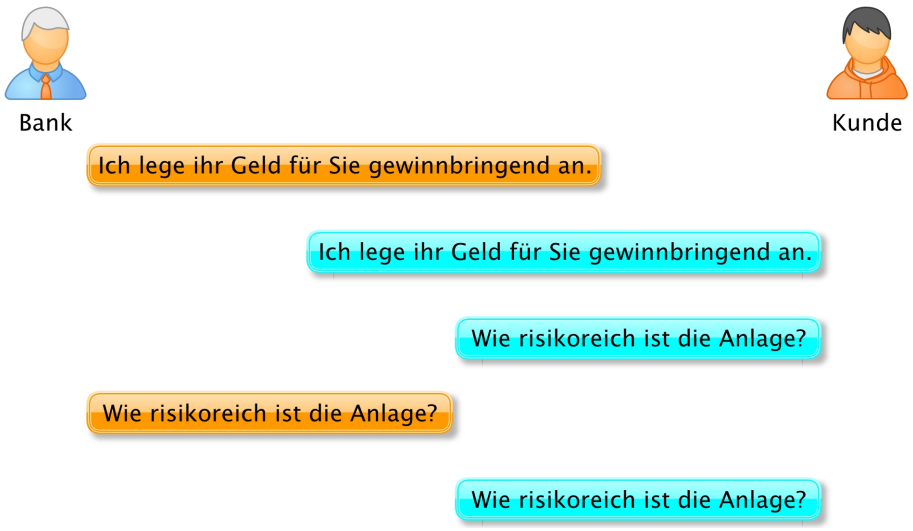

Abbildung 2.9: Wiederholen was gesagt wurde

Hier ist dann vielleicht auch der Richtige Punkt, um einen recht wichtigen Begriff in den kommenden Erläuterungen einzuführen: Die Richtung. Stellt man sich vor man hätte nun während des Gesprächs immer ein Blatt Papier und einen Stift zur Hand und würde immer das aktuell Gesagte und Gehörte mitschreiben. Wir gehen hier davon aus, dass jeder den anderen ausreden lässt. Man teilt das Papier nun auf in eingehend (incoming) und ausgehend (outgoing). Dann fungiert das Papier nun als sehr einfacher Buffer (Abb. 2.7). Nun wäre also die Richtung zu dem was wir vom Gegenüber verstanden haben “incoming” und daher die des selbst Gesagten “outgoing”.
Jetzt stellt sich die Frage, was haben wir nun davon? Im Grunde hätten wir, analog gedacht, nur eine Tonbandaufnahme in der eine Partei (Bank) die andere Partei ständig auffordert zu wiederholen was sie selbst sagt.(Entschuldigen sie bitte, dass ich im folgenden “Ich” öfters groß schreibe, es soll etwas helfen die Parteien besser zu unterscheiden.) Nehmen wir an, ich sei die Bank so geschieht dies immer im selben Rythmus: Ich spreche, Sie wiederholen - Sie sprechen, Ich sage Ihnen was Ich von Ihnen gehört habe, Sie wiederholen was Ich gesagt habe. So prüfe ich also meinen Buffer permanent ab: Erst Outgoing, dann Incoming.
So wie hier gerade beschrieben, bringt der Algorithmus allerdings so gut wie keinen Vorteil. Auf der Ebene des Gesprächs macht er Kommunikation so gut wie unmöglich. Bringt man nun allerdings die paketbasierte Telefonie mit ein, so kommt man schnell auf die Idee sämtliche Wiederholungen, die auf analogem Wege nicht nur umständlich sondern auch noch unnütz dazu sind, in einem anderen “Kanal” zu

führen. Diesen Kanal werde ich von hier ab als “Signier-Kanal” bezeichnen. Der offensichtliche Vorteil ist, dass man das Gespräch selbst völlig unbeeinflusst führen kann, während im Hintergrund die Abfragen wie oben beschrieben laufen können. In den kommenden Abschnitten werde ich das hier Beschriebene kontinuierlich mit technischen Deteils und somit mit Realität befüllen, wodurch sich ein gutes Gesamtbild des Algorithmus, seinen Vorraussetzungen und vor allem seiner Funktionalität ergibt.

2.4.2 RTP Strom und Intervalle

Jetzt werde ich ganz langsam auf die technische Grundlage des Algorithmus eingehen und etwas genauer veranschaulichen, was ich oben mit “gesagt” und “gehört” nur sehr ungenau umschrieben habe. In Abb. 2.10 sieht man, wie der RTP Strom benutzt wird. Essenziell sind dabei die Paketnummern der RTP Pakete. Je nach Codec werden diese Pakete in bestimmten Zeitabständen gesendet und bestenfalls auch empfangen. Für den Audio Codec PCMU wäre das ein RTP Paket je 20 Sekunden, wobei der Codec selbst mit einer Abtastrate von 8000 Hz arbeitet. Die geschätzte Bandbreite des Codecs ergibt sich aus 8000 Hz x 8 bit = 64 kbit/s, dazu kommt natürlich noch der Overhead durch die benutzten Protokollschichten. Dies soll aber nur etwas verdeutlichen auf welcher Ebene und mit welchen Daten RTP hier verwendet wird.

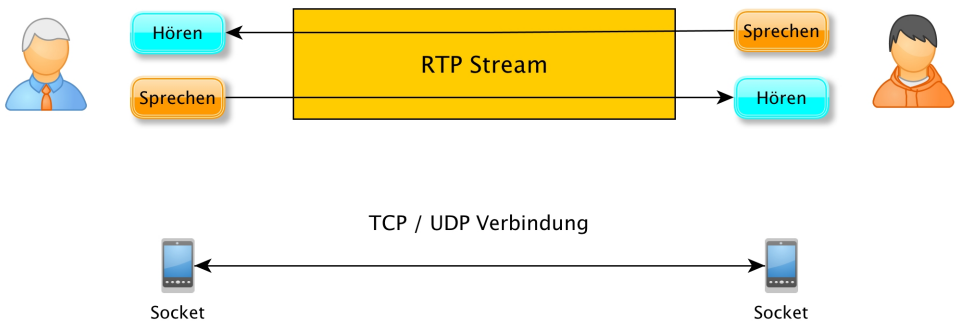

Abbildung 2.10: Ein RTP Strom in der Praxis

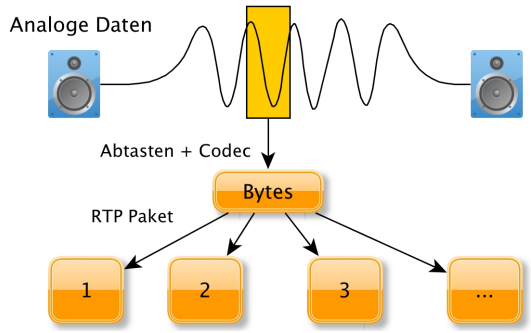

Abbildung 2.11: Wie muss man sich die Funktion eines Codecs vorstellen?

Aus RTP und dem verwendeten Codec ergibt sich nun ein Bild, das einem Strom gleicht. Beide Seiten der Kommunikation erhalten nun fortlaufend Pakete vom Gegenüber. Diese Pakete können, wie im Kapitel 2.4.1 beschrieben, in einem separaten "Signier-Kanal" benutzt werden und somit den Audio-Strom, also das Gespräch, unberührt lassen. Man kann sich nun vorstellen, dass das vom Kunden gegebene "okay" aus den RTP Paketen Nummer 756 - 770 besteht. (Was angesichts der Tatsache, dass bei der Verwendung von PCMU ca. 50 RTP Pakete pro Sekunde benötigt werden, gar nicht so unrealistisch ist.) Wir bleiben dabei, die Bank ist in unserem Szenario der Archivierer. Betrachten wir das "okay" aus dem Gespräch und behandeln es entsprechend im neuen Signier-Kanal:

- Die Bank nimmt die empfrangenen Pakete 756 - 770 und "schnürt" sie zu einer Nachricht zusammen. (Wie das genau passiert werde ich später noch erläutern)
- Nachricht wird an Kunde im separaten Kanal gesendet.
- Kunde erhält Nachricht.
- Kunde vergleicht nun, ob und wenn ja, welches Paket zwischen 756 und 770 er tatsächlich gesendet hat und bildet eine Antwort, die **ausschließlich** die Pakete enthält die er auch tatsächlich gesendet hat. Überprüfen kann er das z.B. durch einen Buffer, in dem alle gesendeten Pakete vorerst aufgehoben werden (das wäre unser Papier im Kapitel 2.4.1).
- Bank erhält diese Antwort und hat nun nebenläufig zum Gespräch erfahren, was von ihren eingegangenen Audio-Daten auch vom Gegenüber gesendet wurde. Man müsste hier denken: Was soll das Abfragen? Die Bank kann im schlimmsten Fall nur weniger verstehen, als der Kunde gesagt hat, aber nie mehr. Das stimmt im Prinzip, allerdings ist die Überprüfung der eingehenden Richtung (Incoming) für die "Beweisführung" genauso wichtig wie die Überprüfung der ausgehenden Richtung (Outgoing). Außerdem sollte man immer

im Hinterkopf behalten, dass jemand versuchen könnte Pakete in den Strom zu schmuggeln.

Um hier vollständig zu bleiben, kommt nun das Wiederholen in der umgekehrten Richtung. Die Bank, der Archivierer, fragt nun seine Gegenseite, was er verstanden hat (Outgoing). Nehmen wir auch dazu an, dass der letzte Satz des Bänkers in den RTP Paketen 640 - 712 übertragen wurde.

- Die Bank bildet auch für die Pakete 640 - 712 eine Nachricht und schickt sie über den Signier-Kanal an den Kunden.

- Der Kunde erhält diese Nachricht und schaut nun in seinem Buffer unter "incoming" nach. Jedes der Pakete aus der Nachricht, das sich auch tatsächlich im Buffer befindet, wird nun in die Antwort geschrieben und dann versendet.

- Die Bank hat mit dieser Antwort nun die Bestätigung, ob und welche Pakete auf der Gegenseite angekommen sind und dementsprechend auch welcher Teil des Gesprächs zu hören war.

Damit wären nun beide Richtungen in denen der Algorithmus arbeitet auf der Ebene der Pakete abgehandelt. Es ist leider sehr unrealistisch einen Algorithmus zu verwenden, der immer ganze Sätze, also deren RTP Pakete, validiert. Technisch wäre das entweder nur schwer oder unsauber zu lösen. (Man könnte hier auf Gesprächspausen warten und dann davon ausgehen, dass dies ein Satz oder wenigstens ein Teil eines Satzes war.) Ebenfalls nachteilig wäre der Ansatz kontinuierlich jedes einzelne Paket auf diese Weise zu validieren, hier entsteht ein wirklich unverhältnissmäßig großer Overhead. Das Verfahren, das sowohl von mir als auch von Herrn Hett verwendet wird, basiert auf der Bildung von Intervallen über den RTP Strom. Zu dieser Intervallbildung gibt es grundsätzlich zwei verschiedene Ansätze:

- Ein Intervall wird auf Grund einer bestimmten Menge an Paketen gebildet, die im jeweiligen Buffer liegen. (Erreicht einer der Buffer eine bestimmte Größe, nehmen wir mal an 50 Pakete, wird der Algorithmus handeln und die Pakete entsprechend mit der Gegenseite abgleichen)

- Ein Intervall wird in zeitlichen bedingten Abständen gebildet, z.B. jede Sekunde wird der Buffer mit der Gegenseite abgeglichen.

Vorstellen kann man sich das Bilden der Intervalle ungefähr so (Abb 2.12)

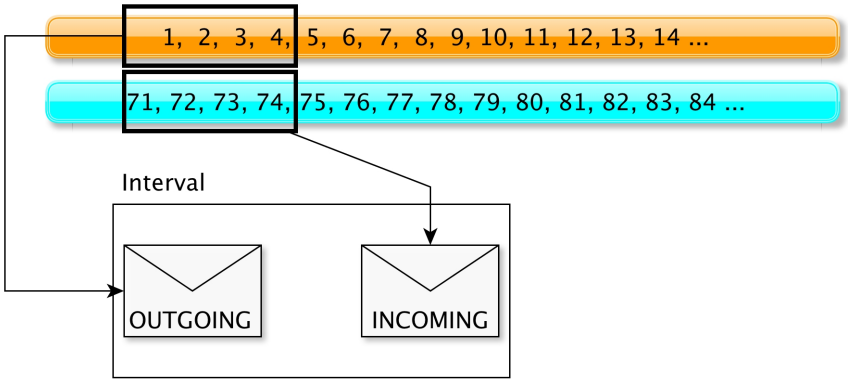

Abbildung 2.12: So werden Intervalle im RTP Strom gebildet

Diese Intervalle werden kontinuierlich über dem RTP Strom gebildet und sollten dazu führen, dass ein Gespräch, während es geführt wird, validiert wird. Das bis hier Beschriebene ist jedoch nur eine Grundlage und kann keine "Nichtabstreitbarkeit" garantieren. Im Moment ist das offensichtlichste Problem, dass der Archivierer oder ein Angreifer die Daten des Signierers, also z.B. seine Antworten über die tatsächlich empfangenen Pakete, munter verändern könnte - ohne, dass es eine Möglichkeit gäbe, diese Manipulation festzustellen. Der Signierer kann also die komplette "Aufzeichnung" als Fälschung bezeichnen und anfechten. Allerdings haben wir auch noch nicht unser zweites Ass ausgespielt: Die Kryptografie, genauer die digitalen Signaturen.

2.4.3 Signaturen und Verkettung der Intervalle

Im vorigen Abschnitt hatte ich die mögliche Manipulation der Daten angesprochen, darauf möchte ich zunächst etwas genauer eingehen: Manipulation zu Verhindern ist nicht das Ziel, sondern diese Manipulation zu erkennen, bzw. Ziel ist es, eine Abwesenheit von Manipulationen an den Gesprächsdaten zu garantieren. Wir wollen also die **Integrität** der Daten sicherstellen. Die Idee ist hierbei, dass der Signierer in einer Antwort die Liste der tatsächlich empfangenen oder gesendeten Pakete (je nach Anfrage) inkludiert und dazu **nur** die Hashes der jeweilig tatsächlich gesendeten oder empfangen Pakete in die Antwort inkludiert. So hat der Archivierer nun alle gesendeten und empfangenen Pakete für ein Intervall und dazu hat er als "Checksumme" die Liste dessen, was sein Gegenüber davon tatsächlich verstanden und gesendet hat. Wollte der Archivierer nun nachweisen, dass ein Paket XY "echt" ist, so würde er einen Hash davon bilden und ihn mit dem aus der Antwort des Signierers vergleichen. Ist das Paket selbst vorhanden und gleichen sich dazu der Hash aus der Antwort und Hash des lokal beim Archivierer gespeicherten Pakets, so kann man von Integrität der Daten sprechen.
Das ist aber nur ein Schritt zum Erreichen der Integrität. Berechtigt ist hier die Kritik, dass der "böse" Archivierer nach Belieben neue Hashes schreiben könnte und

somit wieder unbemerkt die Daten maipulieren kann. Gelöst wird dies allerdings wenn über die Antwort ebenfalls ein Hash gebildet wird. Sodass die Antwort nun im Grunde wie folgt zusammengesetzt ist:

- Liste der Paketnummern
- Hashes der einzelenen Pakete
- Hash über die Liste der Paketnummern + Hashes der einzelnen Pakete

Um jetzt endgültig sicherzustellen, dass dies nicht manipuliert werden kann, wird der letzte Hash mit dem private Key eines Keypaares, z.B. eines Zertifikates, verschlüsselt. Gehen wir zunächst auf die Integrität ein. Hier spielt die Signatur eine Rolle. Der signierte Hash am Ende der Antwort ist unsere "Checksumme". Würde der Archivierer hier nun Daten manipulieren, so würde man wie folgt die Antwort auf ihre Integrität prüfen: Hash der Anwort wie vorgegeben bilden (das hängt von der Implementierung ab, natürlich sollten der "Prüfer" und der zu "Prüfende" hier gleich vorgehen), anschließend wird nun der public Key auf die Signatur angewendet wodurch wir den "orignialen" Hash bekommen. Jetzt **müssen** der eben selbst gebildete Hash und der signierte Hash übereinstimmen, sonst sind die Daten vermutlich manipuliert, auf jeden Fall aber nicht gleich dem, was vom Signierer als Antwort gesendet wurde.
An dieser Stelle passt es gut, sich ein weiteres Prinzip anzusehen, das im Wesentlichen auf die **Kohäsion**, der bis jetzt nur individuell integeren Daten/Antworten/-Intervalle abzielt. Da unser Gespräch aus n-Intervallen bestehen kann, ist es sehr wichtig für eine nachweisbare Rekonstruktion des Gespräches die Zusammenhänge der einzelnen Intervalle zu sichern. Beispielsweise könnte, wenn wir wieder davon ausgehen, dass ein Intervall einem gesagten Satz entspricht, ein Angreifer die Intervalle beliebig austauschen. Denn wir hätten zur Rekonstruktion des Gespräches lediglich z.B. Sendungsnummern in den Intervall-Nachrichten. Durch die Verkettung der Intervalle können wir sowohl die Rekonstruktion sicherer gestalten, als auch die Kohesion der Intervalle untereinander erhöhen. Ein weitere Eigenschaft der Verkettung ist, dass die Nachrichten immer in abwechselnder Richtung verkettet werden: Incoming 1 verkettet Outgoing 1 verkettet Incoming 2 verkettet Outgoing 2 ... und so weiter. Und die Umsetzung ist bestechend einfach: Da wir zu jeder Intervall-Nachricht eine Signatur haben, setzen wir diese mit in die nächste Nachricht: (Abb. 2.13)

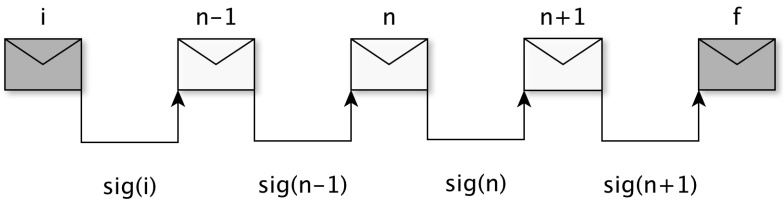

Abbildung 2.13: Verkettung der Nachrichten mit Hilfe der Signaturen

Zur Abbildung 2.13 muss man noch erklären, was mit den grau unterlegten Nachrichten **i** und **f** gemeint ist: Das sind i für Initial-Intervall und f für Final-Intervall. Beide sind extrem wichtig, da sie Anfang und Ende des Gesprächs markieren und unbedingt mit in den Signier- und Archivierprozess gehören. Genaueres zu diesen zwei besonderen Intervallen ist im Abschnitt zur Implementierung zu finden. Einige Eigenschaften dieser Intervalle sind aber mit Sicherheit:

- i Intervall - Initial-Inteval:
  - Zeitpunkt
  - Gesprächspartner A (aus SIP und SDP Daten)
  - Gesprächspartner B (aus SIP und SDP Daten)
  - Informationen über u.a. den benutzten Codec
- f Intervall - Final-Inteval:
  - Zeitpunkt
  - “Call End Reason” - Stammt aus der SIP Nachricht, die das Gesprächsende signalisiert. Mögliche Werte sind “Hang Up” oder z.B. “Error”
  - Weitere SIP und SDP Daten, kann beliebig erweitert werden

Während eine Intervall-Nachricht zum Signieren der Daten aus dem jeweiligen Strom, die Audio-Daten, wie folgt zusammengesetzt sind:

- Zeitpunkt
- Signatur der vorherigen Nachricht (Chaining!)
- Richtung (Richtung, in der signiert werden soll: A→B, B→A)
- Liste der Paketnummern
- Hashes zu den Paketen
- Neue Signatur

Außerdem muss für alle drei Nachrichtentypen eine entsprechende Anfrage formuliert werden, die dann entsprechend beantwortet werden kann. Diese Anfragen, deren Format, hängen eher von der tatsächlichen Implementierung ab als von der Konzeption des Algorithmus.
Ich habe bereits von Signaturen gesprochen und möchte hier auf einen weiteren wichtigen Punkt eingehen: **Authentizität**. Im Grunde ist auch die Integrität der Daten nutzlos, wenn die Urheber nicht bekannt sind. Daher werden Zertifikate benutzt, um durch einen vertrauenswürdigen Dritten bestätigte "Personalien" auszutauschen. Der Archivierer erhält mit dem Zertifikat, dass er vom Signierer bekommt, nicht nur den oft genannten Public-Key, sondern eben auch die Informationen des Ausstellers sowie natürlich die Informationen des Zertifikatinhabers. Der Zertifikatsinhaber, also der Signierer, kann so, über die mit seinem Private-Key verschlüsselten Hashes am Ende einer Antwort, ständig seine digitale Unterschrift geben.

2.4.4 Paketverluste im Signierverfahren

Im in der Einleitung beschriebenen Artikel (Kuntze 2005) ist der Behandlung der Paketverluste auch ein Abschnitt gewidmet, daher möchte ich ihn hier auch nicht auslassen. Paketverluste sind je nach Verbindungstyp und Qualität der Verbindung ein ständiges Thema in der Datenübertragung. Daher muss ein entsprechendes Protokoll zur Kommunikation auch auf Paketverluste vorbereitet sein. Nutzen wir TCP als Transportprotokoll, so ist der Paketverlust bereits durch das Protokoll behandelt, nutzen wir jedoch UDP, so müssen wir uns selbst um mögliche Verluste und deren Behandlung kümmern:
Wie beschrieben besteht die Kommunikation im "Signier-Kanal" aus Anfragen und Antworten. Beide können bei einem verbindungslosen Protokoll wie UDP verloren gehen und daher muss eine Implementierung auch "resending" von Paketen erlauben und behandeln. Wie erkennt man üblicherweise Paketverluste? Eine Anfrage wird gesendet, nach einer Wartezeit t folgt keine Antwort → Anfrage ist verloren gegangen. Die Wartezeit t muss dabei "angemessen" sein. Sowohl maximale Latenz durch die Verbindung, als auch die Zeit, die es braucht um die Antwort zu generieren, muss beachtet werden. Die Folge der verlorenen Antwort ist das erneute Senden der Anfrage. Hier macht es nur bedingt Sinn auf ein ACK, eine Empfangsbestätigung der Anfrage zu warten. Nämlich nur dann, wenn das generieren der Antwort "lange" dauert und dadurch die Wartezeit t überschritten wird und so fälschlich ein Paketverlust angenommen wird. Ebenso diskutabel ist, ob man ein ACK senden sollte um dem Sender der Antwort zu signalisieren, dass diese angekommen ist. Veranschaulicht werden diese expliziten Empfangsbestätigungen zu einer erhaltenen Antwort in den Abbildungen 2.14 und 2.15:

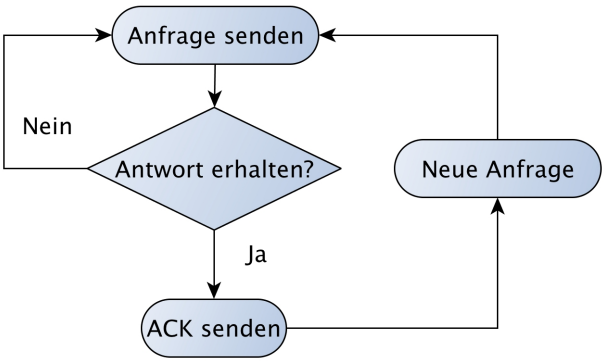

Abbildung 2.14: Ablauf der Antwort und Anfrage Kommunikation mit explititer Empfangsbestätigung ACK aus sicht des Archivierers.

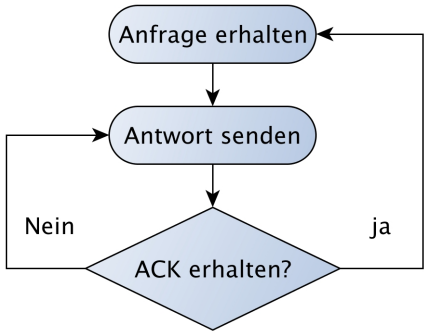

Abbildung 2.15: Passend zur Abb. 2.14 die Sicht des Signierers der das ACK unbedingt benötigt.

Im Artikel (Kuntze 2005) unterscheidet man zwischen impliziten und exliziten Empfangsbestätigungen (ACK). Die beiden kommenden Abbildungen 2.16 und 2.16 zeigen eine Möglichkeit, wie man ganz ohne ACK's auskommen kann. Prinzipiell spart man sich dadurch das ACK, was selbst auch verloren gehen könnte und nur bedingt von Vorteil ist, und handelt sich das oben genannte Problem ein: Die Bearbeitung dauert auf Signierseite so lange, dass der Archvierer über die Wartezeit t hinaus wartet und einen Paketverlust annimmt und unnötigerweise die Anfrage neu sendet, woraufhin der Signierer die Antwort auch unnötig neu sendet. Ein Problem stellt dies nicht dar, da man z.B. durch Sendungsnummern die Anfragen und Antworten identifizieren kann und die letzte Anfrage/Antwort, ohne sie erst generieren zu müssen, erneut senden kann oder im Falle der unnötigen Antwort diese ignorieren kann.

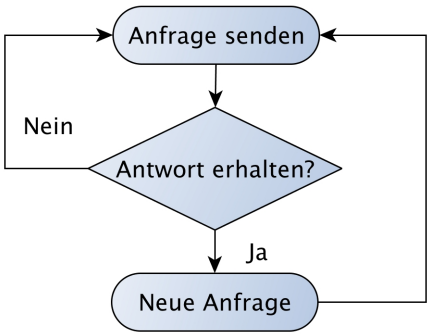

Abbildung 2.16: Der Ablauf im Archivierer ohne explizite Bestätigungen, die Bestätigung erfolgt implizit mit einer neuen Anfrage.

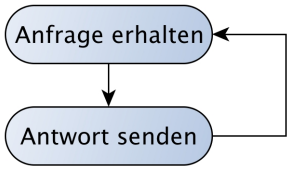

Abbildung 2.17: Passend zur Abb. 2.14 die Sicht des Signierers, der schlicht und einfach auf die Anfragen reagiert.

Es ist nun eine reine Design-Frage, ob man die ACK's, also explizite Empfangsbestätigungen, haben möchte, oder in der schlankeren Version ohne sie auskommt. Entscheident ist nur: Explizite Empfangsbestätigungen führen dazu, dass beide Seiten der Kommunikation "Resending", also eine Fehlerbehandlung, implementieren müssen, während die implizite Version eine Fehlerbehandlung und Erkennung nur auf einer Seite benötigt (hier der Archivierer).

2.4.5 Folge

Zusammenfassend können wir dem Algorithmus nun folgende Eigenschaften zusprechen:

- **Integrität**: Der Signierer überträgt Hashes zu den empfangen und gesendeten Paketen. Eine Rekonstruktion des Gesprächs kann nur mit Paketen stattfinden, zu denen ein Hash passt. Veränderungen der Pakete, z.B. dem RTP Payload, würden zu Fehlern bei der Überprüfung führen. Durch das Hinzufügen eines mit dem Private-Key verschlüsselten Hashes (Signatur) zu jeder Antwort, also einem Intervall, und druch die Verkettung dieser Antworten durch das inkludieren der Signatur der vorangegangen Antwort entsteht eine Kette,

deren Konsistenz im Wesentlichen davon abhängt, dass die verketteten Daten, sowie die Kette selbst, nicht verändert werden.

- **Paketverlust im Audio-Kanal**: Damit ist nicht der Paketverlust auf Ebene der Signaturen gemeint, sondern die Behandlung der Paketverluste im VoIP Kanal. Gemäß "What You Sign Is What You See" Landrock (1998) wird nur signiert, was sich auch wirklich im eigenen Buffer befindet. Dadurch werden Paketverluste aufgedeckt und nachweisbar.

- **Rekonstruierbarkeit** Der Archivierer speichert alle Pakete die er gesendet und empfangen hat. Das ist die Basis der Rekonstruktion. Hinzu kommt jetzt das "Gesprächsprotokoll" des Algorithmus. Dadurch, dass in diesem Protokoll mit den signierten Intervallen alles genau festgehalten ist was auf Seite des Signierers angekommen ist, kann der Archivierer nun das Gespräch anhand der Hashes und Paketnummern der jeweiligen Intervalle genau so rekonstruieren wie es für den Signierer stattgefunden hat bzw. so wie es der Signierer auch "unterschrieben" hat.

Die drei obrigen Punkte können nochmals zusammengefasst werden: **Übereinstimmung / Kongruenz**. Genau das muss der Algorithmus in erster Linie leisten, er muss eine Übereinstimmung zwischen dem liefern, was theoretisch (in einer Perfekten Umgebung, ohne Paketverluste, Übertragungsfehler etc.) gesendet und empfangen wurde und dem was tatsächlich davon angekommen ist. In unserem Szenario ist das typischerweise eine Übereinstimmung zwischen dem was der Archivierer speichert und dem was der Signierer davon gesendet und empfangen hat.
Ein weiteres Merkmal des Algorithmus ist **Kohäsion / Zusammengehörigkeit**: Zu diesem Punkt gehören die Eigenschaften, die dazu notwendig sind, um ein Gespräch vom Beginn bis zum Ende aufzuzeichnen. Wichtig ist dafür ein Paket, ich habe es oberhalb schon mal Inital-Intervall genannt, dass den Beginn des Gesprächs (Zeitpunkt) markiert. Darin enthalten sind sowohl der Gesprächsbeginn als auch die Authentifizierungsdaten der SIP Parteien sowie das Zertifikat des Signierers. Realisieren könnte man den Gesprächsbeginn mit einem Timestamp-Service, der dieses Initial-Intervall signiert.
Alle folgenden Pakete werden nun beginnend beim Inital-Intervall verkettet. Dabei werden die Signaturen der Intervalle immer wieder auf das Zertifikat des Signierers zurückzuführen sein. Das ist dann auch gleich ein weiterer Punkt, den der Algorithmus bietet: Eine kontinuierliche Authentifizierung. Zusammen mit den kontinuierlich gebildeten Intervallen ist es so möglich, parallel zum Gespräch, permanent nicht nur die gesendeten und empfangen Pakete zu überprüfen, sondern eben auch die "Echtheit" des Gegenübers. So wirkt man möglichen Angriffen auf das Gespräch und das Signaturverfahren schon entgegen, während es läuft.

3 Der Algorithmus als Bibliothek

Die Idee, den Algorithmus als eine Art Bibliothek zu implementieren, kam recht früh ins Gespräch, da ich zum Zeitpunkt als ich mich mit dem Algorithmus befasst hatte, noch nicht genau wusste welchen Client ich erweitern würde. Aus dieser Not heraus stammen folgende Überlegungen und Konstruktionen die dazu geführt haben, dass nun aus dieser Bibliothek sowohl Sgnierer als auch Archivierer benutzt werden können ohne dabei in möglichen Zielumgebungen große Integrationsmaßnahmen durchführen zu müssen.

3.1 Abgrenzung zur Arbeit von Christan Hett

Herr Hett hat sich mit dem Thema Non-repudiation in seiner Diplomarbeit "Security and Non-Repudiation for Voice-over-IP conversations" bereits 2006 beschäftigt und eine mögliche Implementierung vorgelegt. Da erscheint es schon fast sinnlos sich dem Thema und vorallem der Implementierung nochmals zu widmen, wenn man allerdings den technologischen Fortschritt seit 2006 betrachtet, dann war es an der Zeit das Thema nochmals zu untersuchen. Bei dieser Untersuchung liegt der Fokus auf der Umsetzung dieses Algorithmus basierend auf Smartphones, genauer Android basierte Smartphones.

3.1.1 Zusammenfassung

Grundsätzlich kann man sagen, dass Herr Hett und ich am gleichen Thema arbeiten. Ich verwende auch einige seiner Ideen weiter und überprüfe ihre Umsetzung erneut. Allerdings besteht ein wesentlicher Unterschied und damit auch das Ziel meiner Arbeit darin, ein Szenario zu bearbeiten, dass Herr Hett nicht bearbeitet hat.
Herr Hett hat sich in seiner Implementierung damit auseinandergesetzt ein Szenario zu beschreiben, das er "Self-Signed Archive"Hett (2006) nennt. (Siehe Abb. 3.1)

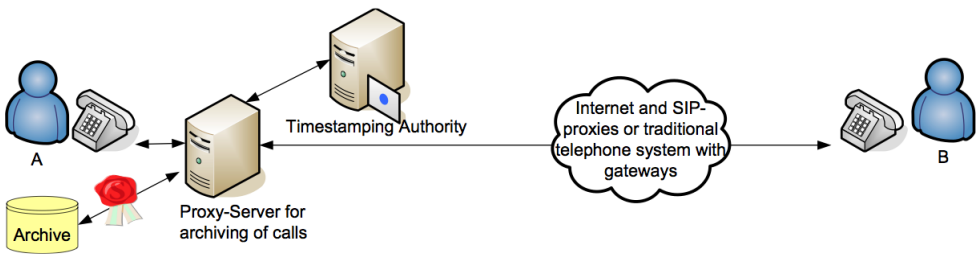

Abbildung 3.1: Self-Signed Archiver, Quelle: Hett Hett (2006)

Was dabei schnell auffällt ist, dass es wohl nur um das Archivieren geht nicht aber um das beideseitge Unterzeichnen der Gesprächsdaten. Herr Hett schreibt auch deutlich, dass die Signierfunktion in diesem Szenario ganz allein bei A liegt. A signiert was später archiviert wird, B ist dabei "nur" Gesprächspartner und nicht Teil des Signaturprozesses. Von einer Nachweisbarkeit im Sinne des Algorithmus aus Kapitel 2.4 kann also nicht die Rede sein. Der eingesetzte Proxy erledigt hierbei das Signieren und Archivieren, der Timestampserver dient dabei als vertrauenswürdiger Dritter, der den exakten Beginn und das Ende des Gesprächs datiert.
Ein anderes Szenario hat er jedoch als "Signing interactive, duplex voice conversations between two parties" bezeichnet, das ist im Grunde auch Ziel meiner Implementierung:

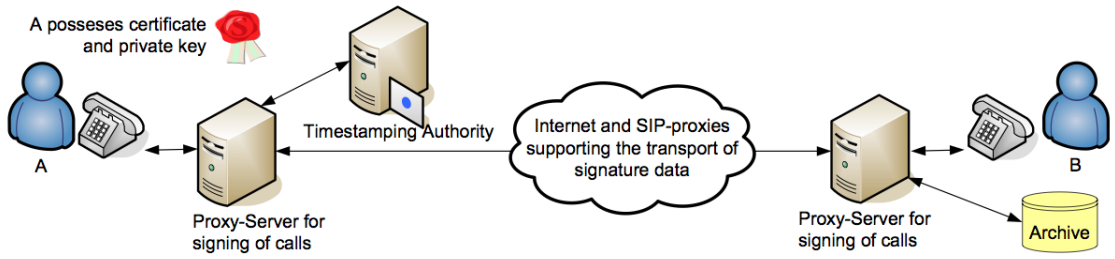

Abbildung 3.2: "Signing interactive, duplex voice conversations between two parties" - Signierprozess zwischen zwei Parteien, Quelle: Hett Hett (2006)

Denn hier agieren die von Herrn Hett mit "Intelligenz" versehenen Proxys, vor den IP-Telefonen als Signierpartner, bzw. Achrivierer und Sender Paar. In Abb. 3.2 sieht man, dass das Archiv auf Seite von B liegt, das Archiv könnte aber auf jweils beiden Seiten liegen. Dieses Szenario erfüllt den im Kapitel 2.4 Algorithmus, ist aber nicht Teil von Herrn Hetts Implementierung gewesen.

3.1.2 Szenario

Neben den Verschiedenen Szenarien, die wir bearbeitet haben, ist der wohl grundlegende Unterschied zwischen meiner Arbeit und der von Herrn Hett die technologische Basis. Herr Hett ist damals noch von "dummen" IP-Telefonen ausgegangen, bzw. hat sich offensichtlich dagegen entschieden ein Softphone [1] zu erweitern. Die Gründe dafür waren vermutlich, dass solche Clients entweder nicht sinnvoll erweiterbar oder erst gar nicht quelloffen verfügbar waren. Ich hätte mir jedenfalls gut vorstellen können, beide Szenarien in die Clients zu Integrieren, wobei das Szenario aus Abb. 3.1 in einem Proxy System gut aufgehoben ist.

[1] Softphone: Softwarebasierte Emulation eines IP Telefon's, sehr verbreitet ist X-Lite oder Skype

3.1.3 Softwarearchitektur

Zur Softwarearchitektur würde ich gerne mehr schreiben, jedoch ist der Code, den man mir von Herrn Hett überlassen hat, größtenteils undokumentiert und so müsste ich einiges an Zeit darauf verwenden mich einzulesen und daraus eine Architektur abzuleiten. Ich kann jedoch mit großer Sicherheit sagen, dass sich unsere Entwürfe nicht nur bedingt durch das Szenario grundlegend unterscheiden sondern auch durch die Implementierung- des Signier und Archivierprozesses. Hinzu kommt, dass Herr Hett bei den Zertifikaten explizit auf Windows Software eingeht, was bei meiner Implementierung nicht der Fall ist, sondern komplett dem "Integrator" des Zielsystems überlassen wird. Grundsätzlich sollte meine Architektur eine einfache Erweiterbarkeit und Integration bieten. Herr Hett scheint hier einen Ansatz verfolgt zu haben, der sehr maßgeschneidert auf sein bearbeitetes Szenario ist. Ich kann jedenfalls nicht viel aus seiner Implementierung gewinnen, viel weniger als ich mir zu Beginn des Projektes vorgestellt hatte.
Zusammenfassend kann man sagen: Die beiden Szenarien sind bedingt durch ihre Funktionen nicht wirklich in einem gemeinsamen Entwurf unter zu bringen.

3.2 Vorstellungen und Umgebung

Hier kläre ich mal ein paar grundsätzliche Ideen und Konzepte die zur Implementierung der Bibliothek geführt haben:
Zunächst wäre da der Überblick auf das Szenario, das im Wesentlichen genau dem aus der Erklärung des Algorithmus entspricht. Die Idee sind also zwei Clienten die in verschiedenen Rollen im Szenario auftreten: Archivierer und Signierer.

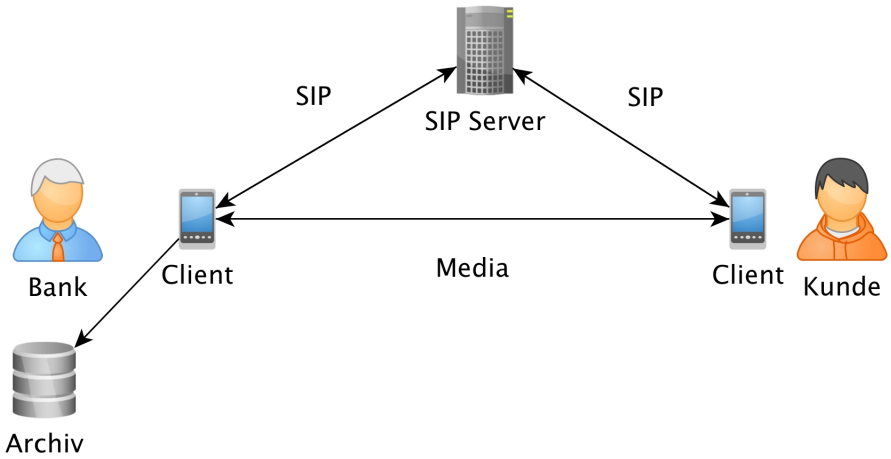

Abbildung 3.3: Die Infrastruktur zum neuen Szenario, Bank archiviert, Kunde signiert

Die untere Verbindung in Abb. 3.3, “Media”, besteht dabei aus einem RTP Strom (Abb. 2.10) basierend auf einer TCP / UDP Verbindung. In der Erklärung des Algorithmus wird dieser Aufbau beispielhaft für einen RTP Strom dargestellt und genau das ist auch im Szenario der Fall. Warum spreche ich es dann überhaupt erneut an? An dieser Stelle wird dieser Aufbau ausgenutzt.
Geht man etwas mehr in die Tiefe, dann kann man sich leicht den Ursprung dieses Stroms vorstellen: Eine Quelle für gesendete und eine Quelle für empfangene Pakete auf beiden Seiten. In Verbindung mit der Anwendung von Audio Codecs entstehen dann die jeweils Typischen RTP Pakete mit ihren Payloads. Die Idee ist dabei die Pakete dupliziert abzulegen, bevor oder nachdem sie gesendet werden und nachdem sie empfangen werden. An genau diesen Stellen werden die Pakete aus dem “echten” Strom abgegriffen und in die Bibliothek eingespeist. Diesen Zuführmechanismus nenne ich von hier an Provider. (Abb. 3.4)

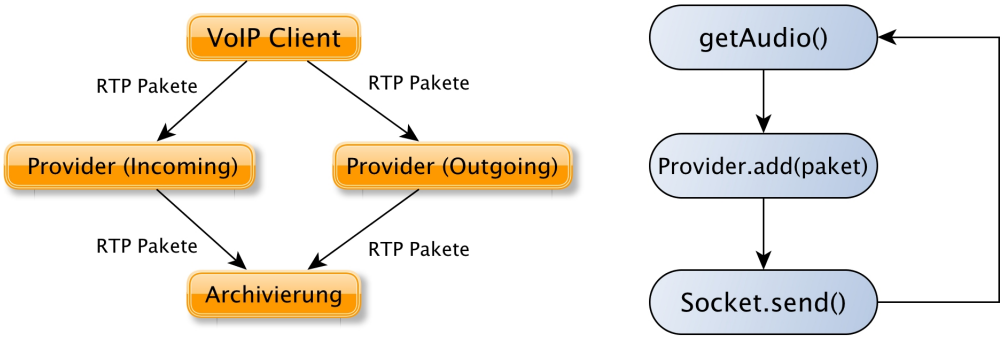

Abbildung 3.4: Funktion des Providers im VoIP Client, rechts die Idee für die Architektur, links die Vorstellung wo genau im Code das einspeisen passieren muss.

Dieser Provider ist aus softwaretechnischer Sicht nur ein Interface, das im Wesentlichen *add()* und *get()* als Methoden vorschreibt. Somit kann ein Provider nach belieben implementiert und damit auch sehr flexibel ausgetauscht und ersetzt werden.

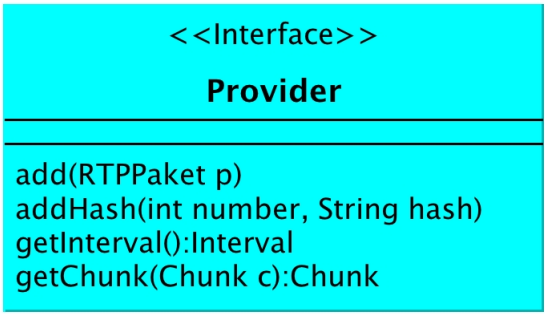

Abbildung 3.5: Das Interface "Provider" in der Software-Architektur.

Warum man diese Methoden unter Umständen austauschen möchte, ist zu erklären: Der Algorithmus basiert im Grunde auf Hashes und Hashfunktionen, diese sollten daher möglichst performant sein. Es gibt aber mehr als diesen Ansatz der performanten Implementierung von Hashfunktionen, vielleicht hilft es auch schon die Hashbildung in einem anderen Thread zu betreiben. Genau für diesen Zweck gibt es zwei verschiedene *add()* Methoden. Die eine fügt Pakete "roh" ein, so wie sie vom System erzeugt worden sind, die andere berechnet bereits deren Hashes. Warum ich von Threads spreche erklärt sich darin, dass typischerweise die beiden Audio-Ströme für Sprechen und Hören getrennt als Threads realisiert sind. Unter Umständen kann es also sein, dass man durch die Hashberechnung in diesen Threads die Audioübertragung unnötig verzögert. Dann muss man allerdings akzeptieren, dass die Hashes im Thread der Signierfunktionen berechnet werden muss und diese dadurch verzögert werden.
Eine weitere wichtige Schnittstelle ist der sogenannte "SessionDataHold", in diesem Objekt, das die Bibliothek als Schnittstelle anbietet, werden sämtliche für den Signierprozess zur Verfügung stehenden Daten eingefügt und intern von der Bibliothek genutzt. Wie erklärt bietet SIP, bzw. eine SIP Nachricht, die Möglichkeit im Anhang (SDP) zusätzliche Sitzungsinformationen zu senden. Diese Informationen werden vermutlich von jedem VoIP Client geparst oder sind parsebar. Die Idee ist, dass ein Integrator nun nur an den Stellen an denen er SDP Nachrichten empfängt, diese parst und die Informationen entsprechend in den "SessionDataHold" überführt. Wichtige Elemente für den Algorithmus sind dabei: User A, User B, Zeitpunkt des Gesprächs, verwendeter Codec, Grund warum der Anruf beendet wurde (CallEndReason), Zeitpunkt des Anrufendes ... . Diese Informationen einzutragen, sobald sie verfügbar sind, bietet keine Gefahren im Sinne von gleichzeitigem Zugriff oder sonstigen Synchronisierungsproblemen, da man der Bibliothek den "start()" Aufruf geben sollte, nachdem die Daten vorhanden sind. Im Grunde soll die Bibliothek gesteuert werden wenn SIP Request eintreffen, z.B. ein *INVITE*. Da dieses Eintreffen eines Invites auch irgendwie dem Benutzer kenntlich gemacht werden soll, kann man also davon ausgehen, dass es dafür spezielle Codeabschnitte im Client gibt. Opti-

malerweise sind diese Stellen sauber definierte CallBack Methoden zu den einzelnen SIP Requests / SIP Events. Hier wäre dann die Integration sehr elegant, da man sowohl Informationen, als auch nötige Aktionen an der gleichen Stelle im Code hat:

function onCallIncoming(String Message)
parse(Message)
//include new actions
addToDataHold(Value1, Value2, ...)
launchArchive()
//continue with normal execution
end function

Ein weiteres Merkmal der Bibliothek ist, dass sie permanent in eigenen Threads läuft. Bis auf die Funktionen zur Signalisierung des Anfangs und Endes eines Gesprächs und den oben genannten Schnittstellen, bleibt sogar die Prozessarchitektur des verwendeten Clients unangetastet. Hier zu argumentieren, dass durch eigene Threads die Laufzeit nicht beeinträchtigt würde, wäre falsch. Die Laufzeit wird so oder so beeinträchtigt, jedoch kann diese Beeinträchtigung durch neue Threads mit den Signierfunktionen im Optimalfall (genügend Systemleistung, mehrere CPU's) unbedeutend gering sein. Mehr dazu in den folgenden Abschnitten.

3.3 Softwaredesign - Ansätze, Ideen, Lösungen

Einer der wichtigsten Ansätze für das Design der Architektur der Bibliothek ist wohl der Gedanke, dass hier ein Protokoll implementiert werden muss. Der Algorithmus kann gut in ein Protokoll übersetzt werden mit folgenden Grundzuständen:

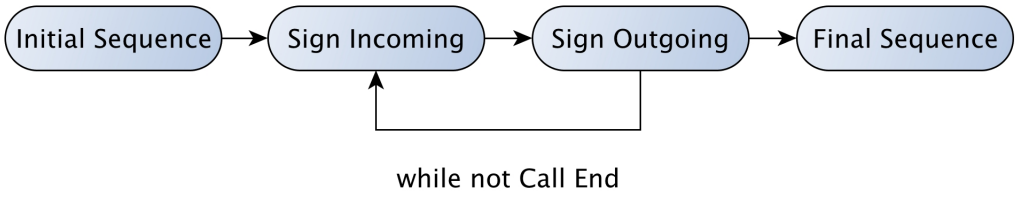

Abbildung 3.6: Die Übersicht über die Protokollzustände (stark vereinfacht).

Die Darstellung in Abb. 3.6 ist stark vereinfacht und stellt nicht die komplette Übersicht der Protokollzustände dar, die konkreten Zustände und Konzepte werden in den Abschnitten 3.5 und 3.6 erklärt. Diese hier noch sehr überschaubare Menge an Zuständen würde ich vermutlich aus reiner Bequemlichkeit mit einfachen "Switch-Case" Anweisungen formulieren und unterscheiden. Diese Idee bereitet im Grunde keine Probleme und erfüllt voll und ganz was implementiert werden muss:

```
1 int state = 0;
2 while(RUNNING){
```

```
3     switch(state) {
4       case 0:
5            sendInitials();
6            state = 1;
7            break;
8       case 1:
9            sendIncomig();
10           state = 2;
11           break;
12      case 2:
13           sendOutgoing();
14           if(callNotEnd){
15             state = 1;
16           }else{
17             state = 3;
18           }
19           break;
20      case 3:
21           sendFinals();
22           state = -1;
23           break;
24     default:
25           RUNNING = FALSE;
26           break;
27    }
28  }
```

Quelltext 3.1: Zustände realisiert mit switch-case Anweisungen und einfachen Verzweigungen

Diese Art ein paar wenige Zustände zu implementieren funktioniert gut, aber eben nur für wenige Zustände. Wird das System, also die Anzahl der Zustände und deren Übergänge, komplexer, dann wird man früher oder später mit dieser Implementierung an die Grenzen der Benutzbarkeit stoßen. Auch die Wartbarkeit des Codes nimmt mit größerem Funktionsumfang der Zustände deutlich ab. Wollte man jetzt zum Beispiel für den Archivierer asynchrones Anfragen und Empfangen von Sequenzen, nennen wir sie mal Intervallquittungen, ermöglichen, dann wäre man mit einer solch primitiven Konstruktion schnell an einem Punkt wo der Code sehr obscur und unübersichtlich wird.

> Drei Dinge müssen bei jedem Bau berücksichtigt werden und zwar der Nutzen, die Dauerhaftigkeit und die Schönheit. - Andrea Palladio

Nimmt man dieses Zitat als Maxime für seine Architektur, dann liegt die Schönheit zwar im Auge des Betrachters jedoch ist der Nutzen einer einfachen Konstruktion nicht sonderlich beeinträchtigt. Die Dauerhaftigkeit ist es jedoch. Ergeben sich Änderungen im Protokoll oder Zustandsmodell, fällt schnell auf wie unelegant und unschön eine solche Konstruktion ist.
Einen anderen, schöneren, Ansatz für diese Zustände bietet das Statepattern Gamma et al. (1996). Das Statepattern nutzt außerdem eine sehr wichtige Eigenschaft von Java aus, die in der "Switch-Case"-Variante nur primitiv genutzt würde: Objektorientierung. Das Statepattern sieht dabei in UML Sicht so aus:

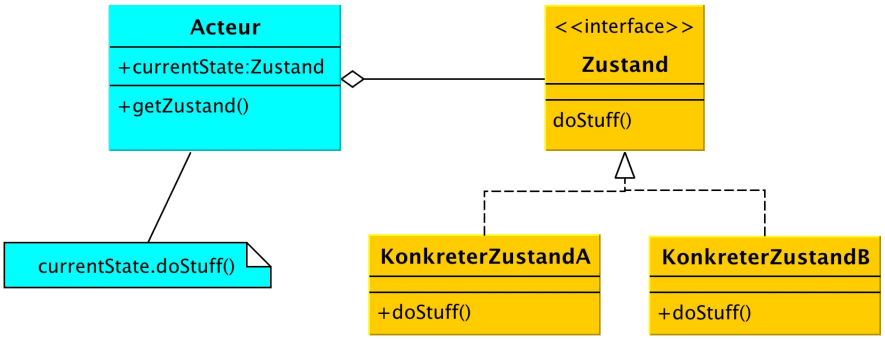

Abbildung 3.7: Das Statepattern in UML

Die Implementierung der States und des Acteurs kann dabei denkbar einfach sein:

```
1  public class Acteur extends Thread{
2
3    ...
4
5    void run(){
6      while(RUNNING}
7          currentState.doStuff();
8      }
9    }
10 }
```

Quelltext 3.2: Der Acteur bei der Ausführung der Zustände, wir gehen hier davon aus, dass die Zustände selbstständig Übergänge realisieren

Während durch die Kapselung eines Zustandes in einem Object das Interface Zustand implementiert dann so aussieht:

```
1  public class ZustandA implements Zustand{
2
3    ...
4
5    void doStuff(){
6      //do my work here
7
8      ...
9
10     if(condition){
11       myActor.currentState = new ZustandB();
12     }else{
13       myActor.currentState = new ZustandC();
14     }
15   }
16 }
```

Quelltext 3.3: Implementierung eines Zustandes, er erledigt seine Arbeit und legt anschließend den Folgezustand ein

Das Design mit Statepattern ist in meinen Augen sehr gut wartbar und die States sind alle recht einfach austausch- und erweiterbar. Der Nutzen muss natürlich auch gegeben sein, für mich ist dieser Ansatz sogar noch nützlicher als der Erstgenannte:

Das Protokoll, ist so wie es hier implementiert ist, zum ersten mal definiert worden, es werden sich Änderungen ergeben. Eine Architektur, die eine schnelle Änderung zulässt ist dafür unerlässlich. Wenn das Protokoll einmal ausgereift und ausreichend definiert ist, könnte man vom Statepattern abkommen und einen simpleren und dadurch vielleicht schnelleren Ansatz bei der Implementierung verfolgen (zweifelhaft inwiefern das heute noch Sinn hat). Genrell kann ich nur sagen, es lohnt sich hier die softwaretechnisch aufwändigere Implementierung zu wagen.
Das Statepattern ist also das Mittel der Wahl zur Implementierung des Protokolls. Ich habe breits davon gesprochen, dass ich das Anfragen und Empfangen von Intervallquittungen gleichzeitig, also nebenläufig, gestalten möchte, ich möchte wenigstens die Möglichkeit dazu offen lassen. Das hat einige gravierende Folgen für die Architektur: Wir brauchen mindestens zwei Threads, zwei Zustandsmaschinen und wir müssen die Kommunikation zwischen den Prozessen regeln. Mit dem Statepattern, implementiert wie oben gezeigt, wird das etwas schwierig. Angenommen wir sind der Archivierer (Abschnitt 3.6) und wollen eine Quittung anfragen, Thread T1 ist für das Senden der Anfragen zuständig, Thread T2 für das empfangen:

- T1 sendet die Anfrage an den Signierer
- T1 signalisiert “Anfrage gesendet” an T2
- T2 muss in den Zustand “Warte auf Antwort” versetzt werden
- T1 muss in einen Wartezustand versetzt werden
- T1 wartet bis Anfrage beantwortet wurde oder sendet nach verstrichener Zeit t neu
- T2 empfängt eine Antwort und signalisiert “Antwort erhalten” an T1
- T1 muss eine neue Quittung vorbereiten
- ...

Man muss für jeden der Schritte einen neuen Zustand defnieren, genauer gesagt: Jeder Schritt wird als eigene Klasse abgebildet. Jeder Thread kann jetzt die verschiedenen Zustände benutzen, die man, wie im Beispiel eben, miteinander verketten könnte (Zustand1 legt Zustand2 ein, Zustand2 legt Zustand3 ein, ...). So gäbe in der Sammlung der Zustände zwei “Ketten”, die eine würde von T1 und die andere von T2 entlang gelaufen. Das löst aber nicht die Abhängigkeiten der Zustände untereinander. Man könnte hier relativ aufwändige Mechanismen zur Synchronisation verwenden. Aufwändig in der Implementierung sind Mechanismen wie Mutex, Semaphoren und Locks nicht, Java bietet viele Mechanismen zur Synchronisation an. Das Aufwändige daran ist viel mehr, diese Mechanismen in den Zuständen umzusetzen als sie zu implementieren. Grundsätzlich war mein Gedanke die starke Bindung

zwischen den Zuständen etwas aufzubrechen und so die feste Kettenbildung zu vermeiden. So kann erst zur Laufzeit ermittelt werden, welcher Zustand tatsächlich der Folgezustand wird. Mein Grundgedanke dabei war ein Stack. Der Stack enthält Zustände, die in der Implementierung *Actions* genannt werden.

```
1  public class ArchiveReceiver{
2    ...
3    while (running) {
4
5      current = actionstack.getLast();
6      if (current == null) {
7        current = new Sleep(this);
8      }
9      current.execute();
10   }
11    ...
12 }
```

Quelltext 3.4: Implementierung des Empfängers auf Archiv-Seite, das oberste Element des Stacks wird genommen und dessen execute() Methode ausgeführt

Die Idee dahinter ist jetzt: Actions legen nicht unmittelbar selbst "currentAction", also den nächsten auszuführenden Zustand, ein. Sie legen den Zustand, in den sie übergehen, auf den Stack. In einigen Fällen ist es gar nicht erwünscht, dass Actions currentAction oder den Stack überhaupt verändern. Das bietet Vorteile: Man kann theoretisch ganze Protokollabschnitte, also mehrere Zustände, auf den Stack legen und dann ausführen lassen. Die Zustände sind entkoppelt, der Stack kann von "aussen" manipuliert werden. Warum will man das überhaupt? Ein Grundproblem mit mehreren Threads die sich gegenseitig beeinflussen, ist Synchronisierung. Ein Thread kann den Stack des anderen manipulieren. Stellt man nun noch über einen simplen Mechanismus in Java sicher, dass der Stack nur jeweils von einem Thread gleichzeitig bearbeitet werden kann, dann kann man durch "geschicktes Einfügen" grundlegende Probleme vermeiden. Ein Beispiel: Wir senden eine Anfrage, wollen dann auf Antwort warten. Klingt relativ einfach, bietet jedoch viele Probleme. Was brauchen wir? Einen Zustand der sendet, einen der wartet, einen der empfängt, einen der als Bestätigung dient. Wer tut was? Bleiben wir bei T1 für den Sender und T2 für den Empfänger. T1 bekommt nun Senden und Warten auf den Stack geworfen. Währenddessen wartet T2 schon im Empfangen Zustand auf die Antwort des Signierers. Wenn T2 die Antwort empfängt, legt er die Bestätigung auf den Stack von T1. Wenn man die Regel beachtet **Aktionen die zwingend hintereinanderausgeführt werden müssen, müssen "am Stück" auf den Stack geworfen werden** und **die Threads pushen sich nur Bestätigungen auf den Stack des anderen** dann bleibt der Stack konsistent. Weitere Synchronisationsmechanismen sind damit unnötig. Beachtet man dies nicht, sieht der Stack von T1 vielleicht so aus: Senden, Bestätigung, Warten. Da im Acteur immer das letzte Element ausgeführt wird, warten wir bis zum Timeout. Beachtet man allerdings diese Regeln, sieht der Stack noch vor dem Senden(!), also bevor eine Bestätigung gepusht

werden kann, so aus: Senden, Warten. Anschließend kommt die Bestätigung “oben” auf den Stack und wird somit als nächstes ausgeführt. Die Bestätigungen nenne ich Kontroll-Sequenzen (im Code ControlSequence). Typischerweise sind diese dazu gedacht den Stack zu säubern und den nächsten Protokollabschnitt einzuleiten. Diese Seqenzen werden von außerhalb gepusht, aber eben vom Thread selbst ausgeführt. Jeder Thread legt die Zustände, die unbedingt aufeinander folgen müssen (kritische Reihenfolgen), nur selbst auf seinen Stack.

3.4 Nachrichten, Anfragen, Antworten

Nachrichten: Die Komplette Kommunikation zwischen Archivierer und Signierer basiert auf XML im Payload eines RTP Pakets. Warum RTP? Weil es ein nützlichen Header bietet. Die Sequenznummer im RTP Header dient zur Identifikation von breits gesendeten oder verlorenen Paketen/Nachrichten. Nutzt man das RTP Format, besteht grundsätzlich die Möglichkeit im Audio-Kanal weitere RTP Pakete mit neuer SSRC Nummer zu übertragen, so wären Sprache und Signaturen im gleichen Kanal möglich. Ein anderer Ansatz ist einen weiteren Kanal im RTP Protokoll zu betreiben und eine andere, eine eigene, Payload-Nummer zu benutzen, um so den Signaturkanal zu deklarieren. Grundsätzlich ist der RTP Overhead zu verkraften, auch im Signaturkanal wo es überhaupt nicht auf die Streaming-Eigenschaften von RTP ankommt. Warum XML? XML hat für mich den großen Vorteil, dass es schnell und einfach zu definieren ist. Es ist im Debugging gut lesbar und kann mit einfachen Mitteln geschrieben und geparst werden.
Anfragen: Anfragen sind unvollständige Antworten. Sie repräsentieren ein teilweise ausgefülltes Formular: Einige Dinge sind Vorgegeben, andere müssen vom Signierer ausgefüllt werden, wieder andere werden überschrieben. Die Vorgaben dienen dazu, die Interessen und Feststellungen beider Parteien zu sichern, z.B. wann das Gespräch begonnen hat oder wer die Gesprächspartner sind. Andere Dinge, wie z.B. die Signierten Hashes und die Liste der Pakete, sind vom Singierer auszufüllen (Hashes) und die Liste der Pakete ist zu überschreiben (es wird nur signiert was auch gesagt/gehört wurde).
Antworten: Ganz so wie es sich gehört, beginnen wir bei der Begrüßung. Die Begrüßung (Abb. 9) besteht aus Time t, Caller a, Callee b, Codec c, dem Public Key k, und der Signatur s. Der Key ist dabei nicht notwendigerweise der Public-Key aus dem gewünschten Zertifikat zur Authentifizierung, sondern ein wesentlich kleinerer (maximal 512bit), für die Sitzung generierter Key aus einem asymetrischen Keypaar. Dieser Key ist kleiner, schneller und für die Dauer der Sitzung völlig ausreichend groß.

```
1 <g>
2  <t>Time</t>
3  <a>Party A</a>
4  <b>Party B</b>
5  <c>Codec<c>
```

```
6   <k>Session-Key</k>
7   <s>Signature</s>
8  </g>
```

Quelltext 3.5: Greeting, vom Signierer ausgefüllt und an den Archivierer zurück gesendet.

Ein Chunk (Abb. 3.6) entspricht einem Intervall für eine Gesrpächsrichtung. Ein Chunk besteht aus den Elementen Time t, Previous Signature (Chaining!) p, Direction d, Numbers of Packets n, Hashes h und Signature s. Der Signierer benutzt d und n um die Pakete in seinen Buffern zu suchen und entfernt nicht gefundene Pakete aus n und fügt für jedes Paket aus n den Hash in h ein. Anschließend wird über t,p,d,n und h ein neuer Hash gebildet und verschlüsselt. Dies ergibt verschlüsselt die Signatur. Greeting, vom Signierer ausgefüllt und an den Archivierer gesendet.

```
1  <c>
2   <t>Time</t>
3   <d>dir</d>
4   <n>Paketnummern</n>
5   <h>Hashes der Pakete</h>
6   <s>Signature</s>
7  </c>
```

Quelltext 3.6: Ein Chunk mit einem Paket, Paketnummer 1, mit der Gesprächsrichtung “outgoing” und dem Hash zum Paket

Und zu letzt noch die Abschiedsnachricht (Abb. 8): Sie besteht aus Time t, Previous Signature p, Reason r, Cerfificate c, SessionKey k und Singature s. Time und Reason kommen vorausgefüllt beim Signierer an, der dann sein Zertifikat anfügt und das ganze dann signiert.

```
1  <f>
2   <t>Time</t>
3   <r>CallEnd-Reason</r>
4   <c>Certificate / Zertifikat</c>
5   <k>Session-Key</k>
6   <s>Signature</s>
7  </f>
```

Quelltext 3.7: Finals, mit “HangUp” als Grund für das Gesprächsende und dem Zertifikat c

3.5 Rollen: Der Signierer

Die erste Rolle die ich besprechen möchte ist dier Signierer. Der Signierer ist der reaktive Teil des Systems, während der Archivierer maßgeblich steuert. Was muss der Signierer im Sinne des Algorithmus leisten? (Abb 3.8)

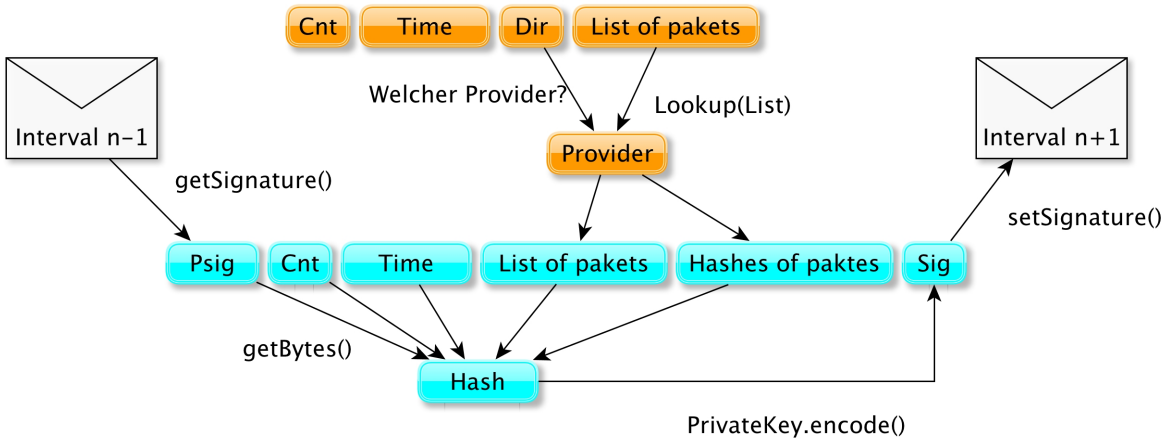

Abbildung 3.8: Die Hauptverantwortlichkeit des Signierers ist Anfragen zu beantworten. (Orange: Inhalt der Anfrage, Blau: Inhalt der Antwort)

Der Signierer beantwortet Anfragen, er reagiert also ausschließlich auf die 3 im Kapitel 3.4 genannten Anfragetypen. Dabei braucht er im Grunde nur den RTP Header zu untersuchen, ob die Sequenznummer “neu” ist, also größer als die letzte und anschließend die Nachricht zu parsen. Ist die Anfrage nicht neu, also die Sequenznummer gleich der letzten wird ein “resend” des letzten Pakets durchgeführt, so wie im Algorithmus unter “Paketverlusten” beschrieben. der Ablauf ist also recht einfach: (Abb. 3.9)

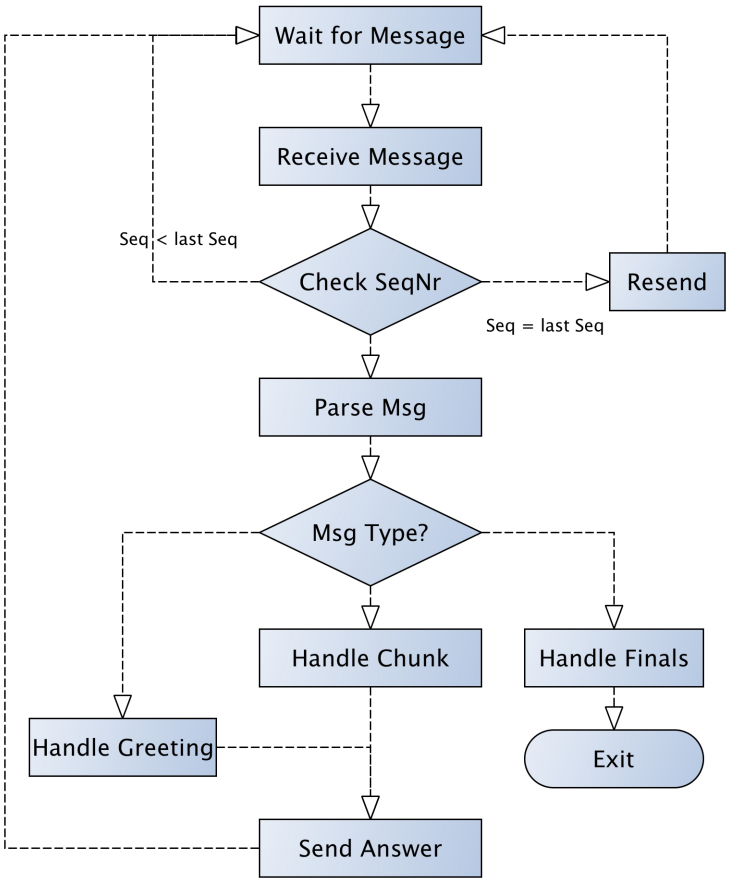

Abbildung 3.9: Ein Ablaufdiagramm das die Vorgänge im Signierer beschreibt.

Der Signierer ist im übrigen mit Switch-Case Anweisungen realisiert. Da es hier nur drei unterschiedliche Möglichkeiten zur Verarbeitung einer neuen Nachricht gibt: Greeting, Chunk und Finals. Das Statepattern kann aber auch hier ohne Probleme realisiert werden, aus Mangel an Zeit und der Bequemlichekeit bei so wenigen Zuständen, habe ich mich jedoch dagegen entschieden, den Signierer weiter auszuarbeiten. Eine grundlegende Idee kann man aber ruhig ansprechen: Der Signierer sollte so simpel wie möglich gehalten werden. Nehmen wir an eine Bank würde eine Telefon-App mit integrierter Signierfunktion verteilen, kann man unmöglich abschätzen wie Leistungsstark die Geräte der Kunden sind. Man müsste schon genaue Angaben über die unterstützten Geräte machen.

3.6 Rollen: Der Archivierer

Der Archivierer steuert das Protokoll. Ich habe im Abschnitt 3.3 bereits von mehreren Threads gesprochen, dies wird hier umgesetzt. Zu erst will ich allerdings auf die zuständigkeiten des Archivierers im Sinne des Algorithmus eingehen:

- Steuert das Protokoll: Gibt Ablauf vor, kümmert sich um Paketverluste und validiert die Signaturen kontinuierlich. (Abb. 3.6)
- Bildet Intervalle (Chunks).
- Versendet Begrüßung.
- Versendet Chunks für die beiden Gesprächsrichtungen.
- Versendet die Schlussnachricht Finals.
- Bietet Schnittstelle zum speichern der signierten Chunks sowie der nicht signierten Daten (zur Rekostruktion/zum Abspielen der Aufzeichnung).

Zum Ablauf des Protokolls komme ich später, denn dies ist mit dem Thema Threads, Statepattern und Stacks verbunden. Das nächste auf der Liste wäre das Bilden von Intervallen. Wie, wann und wo Intervalle gebildet werden ist bereits im Kapitel zum Algorithmus beschrieben. Abbildung 3.10 zeigt die Elemente die für den Signierer in Abb. 3.8 wichtig sind.

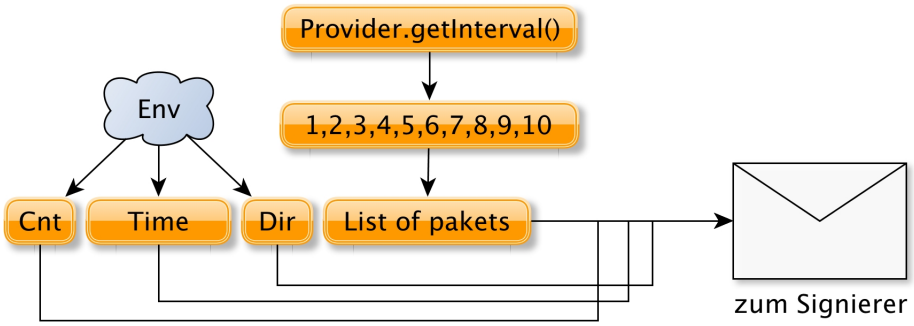

Abbildung 3.10: Das bilden eines Chucks auf Basis der Interval-Idee des Algorithmus.

Hier ist zu sehen, welche Elemente aus der Bibliothek bzw. nicht von außen kommen (Env) und welche Elemente durch die Schnittstelle Provider in eine solche Nachricht fließen. Count Cnt ist die fortlaufende Nummer der Nachrichten und Direction dir ist abhängig davon ob das Protokoll im Zustand “SendOutoing” oder “SendIncoming” ist. Entsprechend der gewünschten Gesprächsrichtung wird auch der Provider ausgewählt. Nach der Antwort des Signierers (Abb. 3.8) haben wir dann folgendes: Die Antwort der Gegenseite und die Daten aus den eigenen Providern (Paketnummern, RTP Pakete) müssen zusammen gespeichert werden. Die Antwort “Chunk” und die Daten “Interval” werden im Code als eine Klasse, mit den Eigenschaften und Attributen beider Klassen, zusammengefasst mit dem Namen: “ChunkInterval”. Ein Objekt dieser Klasse kann anschließend serialisiert (XML) und in eine Datei geschrieben werden. Wesentlich einfacher sind Anfragen vom Typ Greeting und Finals.

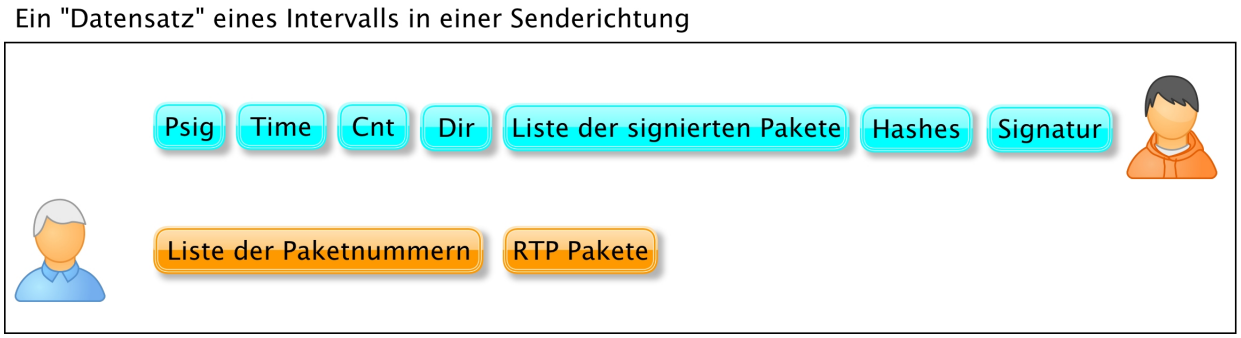

Abbildung 3.11: Blaue Elemente stammen aus der Antwort, orange Elemente sind die Daten aus den Providern des Archivieres.

Deren Objektrepräsentation kann, unmittelbar nach Erhalt, vom Archivierer in eine Datei geschrieben werden.
Die Abläufe im Signierer sind relativ trivial, im Archivierer haben allerdings Sender und Empfänger verschiedene Zustandsstacks mit denen sie arbeiten: (Figur 3.12)

- **Send Greeting** Hier wird die Begrüßung versendet. Das Objekt wird einmalig auf den Stack gelegt und seine "execute()" - Methode n-mal aufrufen. n ist variabel und momentan mit der globalen Variable *maxretry*=3 eingestellt. So wird eine maximale Anzahl an Versuchen gewährleistet, um evtl. Verbindungsfehler zu erkennen. Wichtig ist, dass auf solche "Send" Zustände immer auch "Idle" Zustände folgen, jedenfalls dann, wenn UDP (verlustbehaftet) benutzt wird.

- **Idle** - Zustände lösen was UDP nicht bietet: Eine Erkennung ob Pakete verloren gegangen sind und das erneute Senden des verlorenen Paketes. Der Idle Zustand wird auf den Stack gelegt, sobald etwas gesendet wurde. Der Idle Zustand beinhaltet nichts weiter als einen *Thread.sleep(MillisToSleep)* Aufruf und eine Stackoperation, mit der er sich selbst vom Stack entfernt. Es wird in kleinen Abständen (5ms) n-mal gewartet:

```
1  public int execute() {
2
3    counter++;
4
5    if (counter < MAX_IDLES) {
6
7      try {
8        if (DEBUG)
9            System.out.println(actionName + "␣sleeping");
10
11       Thread.sleep(MILLISTOSLEEP);
12     } catch (InterruptedException e) {
13       e.printStackTrace();
14     }
15
```

```
16        // recusive approach? - better not.
17        // thread.actionstack.stack.lastElement().execute();
18        // so this is either a controll sequence or this object again
19
20        return 0;
21      } else { // initiate resending the packet.
22        System.out.println("Need to resend the package");
23        thread.removeSelf(this); // remove myself from stack, so that object
              under me can be executed.
24        return 1;
25      }
26
27  }
```

Quelltext 3.8: Implementierung der execute() Methode eines Idle-Zustands. Der Zustand kann MAX_IDLES mal aufgerufen werden. So entstehen z.B. die 50ms Wartezeit.

nun wird im besten Fall während des Millisekundenschlafs eine **CrtlSeq** auf den Stack gelegt, die dann an Stelle des Idle Zustandes ausgeführt wird. Passiert dies nicht, entfernt sich der Idle Zustand vom Stack, wodurch nun der Send Zustand wieder zum Vorschein kommt. Es wird erneut gesendet.

- **CrtlSeq** Control-Sequences, Kontroll-Sequenzen. Sie steuern im wesentlichen den Ablauf des Protokolls und sollten gemäß den beschriebenen Vorstellungen von einem kontrollierendem Thread auf den Stack des kontrollierten Threads gelegt werden. Die *execute()*-Methode des Zustands wird dann vom kontrollierten Thread ausgeführt. Die kritischen Stack Operationen werden dann in der eigenen Laufzeit des jeweiligen Threads ausgeführt. Im wesentlichen beschränken sich die Fuktionen der CrtlSequences auf das Säubern des Stacks und das neue befüllen mit Zuständen. Es gibt zu jedem Protokollschritt spezielle CrtlSeq's.

- **NewInterval** leert die Provider, speichert ihren Inhalt in Interval-Objekten zwischen. Hier geschieht das Bilden der Intervalle über die gesendeten und empfangenen Pakete. Dieser Zustand ist wichtig für das fast gleichzeitige Bilden der Intervalle beider Ströme. Anschließend werden beide Intervalle (in,out) in Nachrichten verpackt, diese fertigen Nachrichten können dann versendet werden.

- **SendIncoming** Funktioniert genau wie SendGreeting, nur eben mit der von NewInterval vorbereiteten Nachricht für empfangene Pakete im Payload des UDP Datagramms. Es folgt ein Idle und darauf entweder eine CrtlSeq und damit der nächste Protokollschritt oder eben das erneute Senden, falls Idle nicht länger warten kann und das erneute Senden noch zulässig ist.

- **SendOutgoing** Funktioniert genau wie SendGreeting, nur eben mit der von NewInterval vorbereiteten Nachricht für gesendete Pakete im Payload des UDP

Datagramms. Es folgt ein Idle und darauf entweder eine CrtlSeq und damit der nächste Protokollschritt oder eben das erneute Senden, falls Idle nicht länger warten kann und das erneute Senden noch zulässig ist. Der nächste Protokollschritt wäre "NewIntervall" oder das Senden der finalen Daten (nicht dargestellt, aber im Prinzip gleich den anderen Send-Zuständen).

- **ReceiveGreeting** Hier wird auf die zuvor gesendete Begrüßung gewartet. In diesen Zuständen sollte man nur empfangen und die Antwort an den Sender weiterleiten, da die Zustände für das Empfangen relativ zeitkritisch sind und nicht verzögert werden sollten. Die Idee ist, den CrtlSeq's die empfangene Antwort zu übergeben, und selbst in den Folgezustand (z.B. ReceiveIncoming) zu wechseln. So wird das Überprüfen der Nachricht in den Sender verlagert, ist die Nachricht korrupt (erwarteter Inhalt wurde ungültig modifiziert, Signatur passt nicht zum Schlüssel, Daten passen nicht zum Hash) wird das Signieren und Archivieren sofort gestoppt, da von einem Angriff ausgegangen werden muss. Den Sender zu verzögern ist nicht kritisch und der Empfänger ist so schon bereit zum Empfangen bevor etwas gesendet werden kann.

- **ReceiveIncoming** Gleicht im Prinzip ReceiveGreeting, es werden hier die Intervallquittungen für empfangene Pakete erwartet. Folgezustand ist ReceiveOutgoing.

- **ReceiveOutgoing** Gleicht ReceiveOutgoing, es werden Intervallquittungen für gesendete Pakete erwartet, es folgt HandleCallState.

- **HandleCallState** Diese Action klappert nur die möglichen Variablen ab, die makieren, dass das Gespräch beendet wurde. Man könnte hier auch einbauen, dass z.B. eine Stummschaltung erkannt und korrekt behandelt wird. Es folgt ein ReceiveIncoming oder ein Receive-Zustand um die finalen Daten zu empfangen (nicht dargestellt, aber im Prinzip gleich dem bisherigen Vorgehen mit Greeting, Incoming's und Outgoing's).

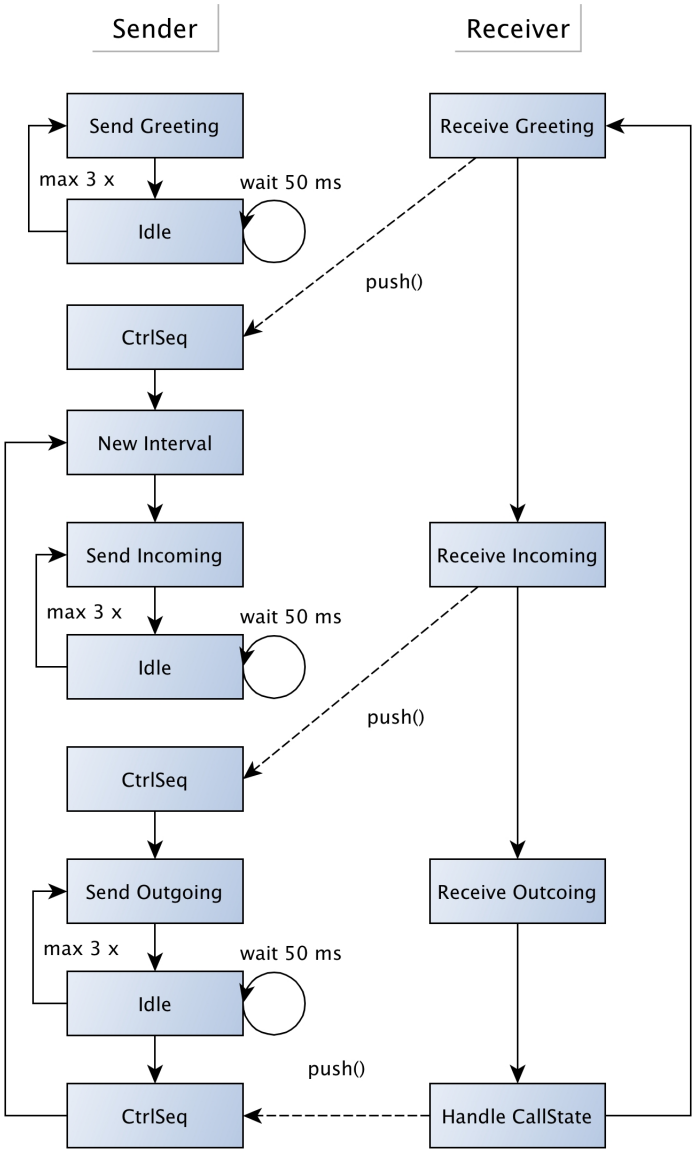

Abbildung 3.12: Hier sieht man den Ablauf des “normalen” Archiviervorgangs mit Fokus auf die einzelnen Zustände, die dazu implementiert wurden.

4 Fallbeispiel: SipDroid

4.1 Anforderungen an den Client

Die grundsätzliche Idee meiner Arbeit war es, einen bereits existierenden Android Client für VoIP Kommunikation zu erweitern, um mir gewisse Arbeiten zu ersparen. Grundsätzlich kam es dabei auf die Folgenden Punkte an:

- **Open Source** ist ein fast selbsterklärendes Kriterium, leider sind viele gängige Clients auf dem Market Teil der Softwarepakete von Softwarefirmen und sind daher aus gutem Grund nicht quelloffen.

- **Funktionalität** ist sehr wichtig, wobei ich mit einem eher eingeschränkten Funktionsumfang schon zufrieden bin. Die Testumgebung wird ein WLAN Netzwerk sein, alle Geräte sind im gleichen Netz. Dieses Szenario lässt dabei infrastrukturelle Probleme außenvor und ermöglicht mir, mich auf die wesentlichen Punkte zu konzentrieren:
  - Implementierung des UI, damit will ich mich nicht beschäftigen müssen. Die App sollte aber geeignete Schnittstellen für den Benutzer haben, damit meine ich nicht nur eine Ansicht um die Verbindungseinstellungen zu editieren sondern auch eine Benachrichtigung für Anrufe und entsprechende Konzepte für das kontinuierliche warten auf eingehende SIP-Nachrichten.
  - Funktionierender SIP Stack. SIP "komplett" zu implementieren wird kaum möglich sein und selbst die typischen Funktionen von SIP zu implementieren dürfte recht aufwendig sein.
  - Funktionierender RTP Stack, genau wie SIP muss auch RTP eigens implementiert werden. RTP ist dabei aber wesentlich kleiner was den Funktionsumfang betrifft und muss in unserem Fall nur die Audio-Übertragung implementieren. RTP reicht sein Payload an einen Codec weiter, somit wären wir mit der Wiedergabe und Abtastung nicht weiter beschäftigt.

- **Architektur/Erweiterbarkeit** muss natürlich gegeben sein. Ich habe mir dabei vorgestellt, nur an bestimmten Stellen der Applikationen ansetzen zu müssen. Ziel ist es für mich, dass jeder Client relativ einfach um das neue Sicherheitskonzept erweitert werden kann. Ich hatte in meinen Vorstellungen zum Algorithmus bereits von CallBack-Methoden gesprochen, ein Client, der dieses Prinzip für Ereignisse nutzt, wäre perfekt geeignet. Hinzu kommt noch, dass für den RTP Strom möglichst zugängliche Implementierung genutzt würden: Der Provider muss sich integrieren lassen.

- **Benutzbarkeit** Die App sollte grundsätzlich ohne mein Beitragen schon einfach und gut benutzbar sein und möglichst ohne große Designschnitzer, die ich dann bearbeiten müsste. Das ist ein Punkt, den man an der App-Bewertung im "PlayStore" erkennen kann. Wirklich bewerten kann ich dies nicht.

- **Dokumentation** ist im Laufe meiner Arbeit das Wichtigste geworden. Das ist mir leider etwas spät aufgefallen. Sollte der Code auch noch so lesbar und die Konzepte durchsichtig sein, so ist bei fehlender Dokumentation der Arbeitsaufwand Code zu nutzen oder ihn selbst zu schreiben mindestens gleich groß.

4.2 Sipdroid

Sipdroid ist ein guter Bekannter unter den Android VoIP Clients, wurde auch oft erweitert und wird noch gepflegt. Der Code wird auf Goole Code gehostet [1]. Dort ist auch ein aktives Forum zu Bugs und häufigen Fragen zu finden. Per SVN checkout bekommt man die aktuelle Version des Codes. Wer eine Dokumentation sucht, der sucht vergebens. Auch die Oberflächen sind eher durchschnittlich, aber durchaus benutzbar.

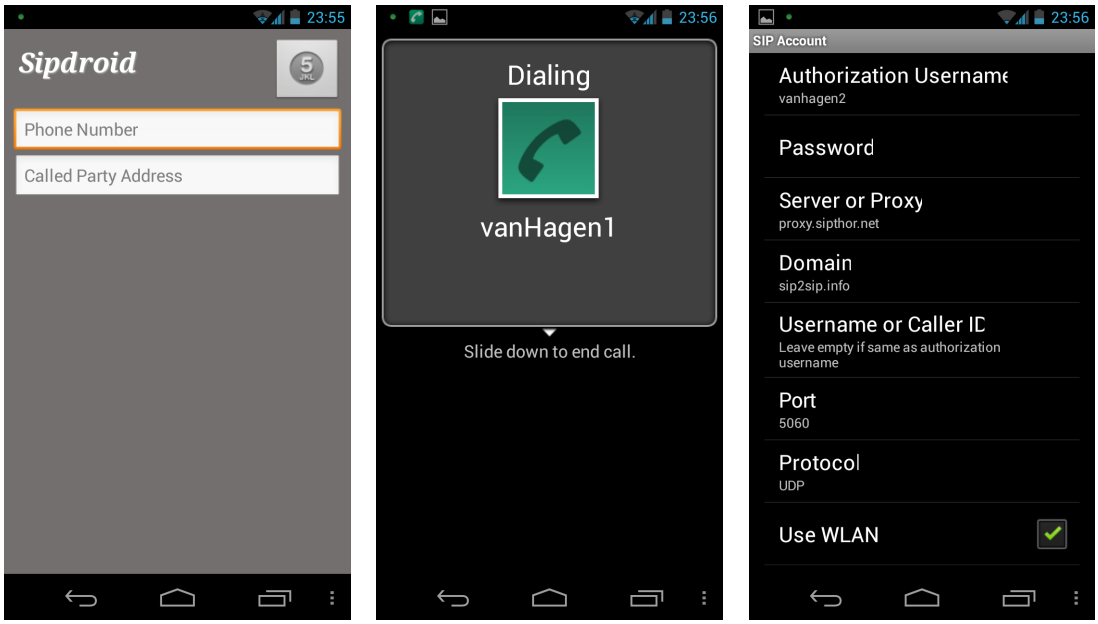

Abbildung 4.1: Von links nach rechts: 1. Wahlmenü für Telefonnummern und SIP Adressen, 2. Das Telefon-Interface mit einem Slide zum Auflegen und Abnehmen und 3. das Settings-Menü

[1] `http://code.google.com/p/sipdroid/`

Was in Abbildung 4.2 auffällt ist, dass die App im ersten Moment nicht sehr selbsterklärend ist und dazu viele Schalter bietet, die ein durchschnittlicher Benutzer sicher nicht sinnvoll zu nutzen weiß. Generell hilft hier das eigene Wiki, das aber leider mehr für Benutzer als für Entwickler ausgelegt ist. Was dort an Fragen und Antworten zu finden ist, betrifft eher Bedienprobleme des Benutzers.
Was die Aversion gegen Dokumentation betrifft, so kann man jedoch als Pluspunkt die benutzten Bibliotheken werten. Diese sind im Gegensatz zum Projekt SipDroid selbst sehr aufgeräumt und auf ihren eignen Seiten gut dokumentiert. Benutzt wird **Mjsip** als Bibliothek für den SIP Stack, diese Bibliothek eignet sich sehr gut für unser Vorhaben.

4.3 Erweiterung

4.3.1 Architektur

Das wirklich Schöne an der Erweiterung bei SipDroid ist, Mjsip. Mjsip bietet einen kompletten SIP Stack an, der beliebig austauschbar und erweiterbar ist. Der Stack ist in vier Ebenen unterteilt:

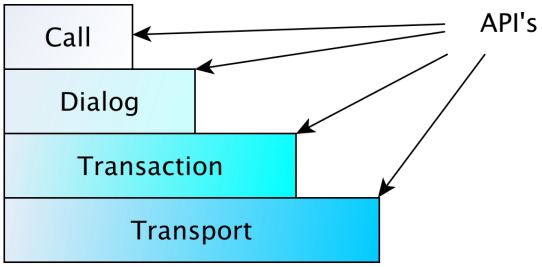

Abbildung 4.2: Die Architektur der Bibliothek: 4 Schichten, 4 API's die vier Level des Stacks repäsentieren.

- **Transport - Layer** ist der tiefste Layer und verantwortlich für den Transport der SIP Nachrichten. Seine Aufgabe ist das Kapseln der Transportprotokolle (TCP/UDP) und das Bereitstellen einer Schnittstelle zum Senden und Empfangen von SIP Nachrichten.

- **Transaction - Layer** sendet und empfängt Nachrichten über den Transport-Layer. Er kümmert sich um Resends, Timeouts und passende Nachritenformate sowie passende Antworten auf Anfragen (Response, Request).

- **Dialog - Layer** ist der Layer, der nun auf die Parteien der SIP-Kommunikation eingeht. Hier werden die verschiedenen Sitzungsteilnehmer im Detail berücksichtigt. Zu jeder Sitzung die gerade Aktiv ist gehört mindestens ein Dialog (ein

Element aus diesem Teil der API). Jede Verbindung einer Sitzung hat einen eignen Dialog, so werden die unterschiedlichen Teilnehmer berücksichtigt.

- **Call - Layer** ist der oberste und abstrakteste Layer. Auf dieser Ebene werden nur noch Ereignisse die in direkter Verbindung mit einem Anruf stehen beantwortet: Wählen, Besetzt Zeichen, Auflegen, Abnehmen, nicht erreichbar etc. . Eigentlich sollte man ihn "CallControlLayer" nennen, da er mit seinen Objekten und Methoden zur Steuerung des Anrufs dient.

Glücklicherweise hat sich das SipDroid Team dazu entschieden die Bibliothek bis in die oberste Schicht zu übernehmen und zu benutzen. Die Kommunikation der einzelnen Ebenen funktioniert über Listener:

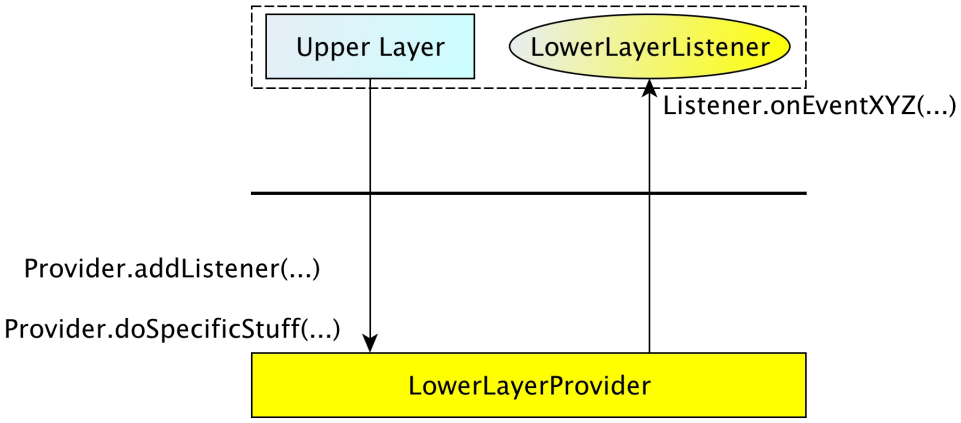

Abbildung 4.3: Modellierung der API's und deren Schnittstellen zur jeweils unteren Schicht.

Beschäftigen wir uns mit der höchsten Ebene der API: Call. In Abb. 4.3 sieht man das Prinzip, also den Aufbau der einzelnen Schichten in Verbindung zu einander. Wenn man nun auf der Ebene "Call" arbeiten möchte, muss man sich die Implementierung des Listeners ansehen. Der Listener erwartet die implementierung folgender Funktionen:

- **onCallIncoming** (Call call, NameAddress caller, String sdp, Message invite), wird aufgerufen, wenn ein INVITE eintrifft.
- **onCallRinging**(Call call, Message resp), wird aufgerufen, wenn ein INVITE zum Gesprächspartner durchgeleitet wurde und dieser vom eingehenden Anruf benachrichtigt worden ist. Siehe "180 Trying".
- **onCallAccepted**(Call call, String sdp, Message resp), wird aufgerufen, wenn ein Anruf angenommen wird (call accepted 2xx).

- **onCallClosed**(Call call, Message resp), wird aufgerufen wenn, ein 200 OK nach einem gesendeten BYE Request eintrifft (call closed).

- **onCallClosing**(Call call, Message bye), wird aufgerufen wenn, eine BYE Anfrage eintrifft (close request).

Call bietet noch weiter Methoden an, die die Interaktion der Bibliothek nach außen erleichtern sollen. Zu diesen Methoden gehören, **call()** (einen neuen Anruf tätigen), **accept()** (einen Anruf annehmen) und **hangup()** (Anruf beenden). Die obene genannen Methoden des Listeners sind keine vollständige Liste der Methoden, die implementiert werden müssen, sie geben jedoch genau die Methoden wieder die wir für unsere Erweiterungen benötigen.
Die Implementierung des RTP-Stroms ist eine Eigenentwicklung des SipDroid Teams. Senden und Empfangen laufen je in einem eigenen Thread, die Kommunikation erfolgt über Java Sockets. Mit kleinen Modifikationen kann man beim erstellen der Threads unsere Provider mit einschleusen und so mit sehr subtilen Änderungen integrieren.

4.3.2 Integration - Signierkanal

Zum erstellen des Signierkanals gibt es drei verschiedene Ansätze:

- **hardwired** Dieser Ansatz ist der am leichtesten Umsetzbare. Er verzichtet auf die Integration des zusätzlichen Medienstroms (neben dem Audio Strom) im SIP und SDP Umfeld. IP's, Ports, Protokolle und Codecs werden für die Entwicklungsphase fest vorgegeben, dies schaltet eine erhebliche Nummer an Fehlern der Infrastruktur aus.

- **Signaturen im Audio-Kanal** Hier ist die Idee die RTP-Pakete mit den Signaturdaten einfach mit einer anderen Quellnummer (SSRC) als die des Audio-Kanals zu versehen. Die Klasse "RTPPaket" in der Implementierung basiert darauf, dass diese Nummer zufällig gewürfelt wird. Man könnte sie auch fest zuweisen, so wüsste man immer, was nun Audio (zufällige SSRC) und was Signatur-Anteil (feste SSRC) ist. Signierer und Archiverer bekämen dann einfach das Socket-Object des RTP Stroms übergeben und würden auch über dieses Kommunizieren. Man kann so die Signierdaten aus RTP-Strom zum Empfangen, nach dem "socket.receive()"-Aufruf, in die Bibliothek einzuspeisen. Diese Lösung lässt die SIP und SDP ebenfalls unbehelligt. Grundsätzlich können hier viele Probleme mit der Infrastruktur auftreten, beispielsweise dann, wenn der RTP Strom über einen Server läuft und dieser die Datenmenge entsprechend des Codecs überprüft. In dem Fall würde schnell auffallen, dass die Signierdaten weder zum Codec des Stroms noch zum ungefähren Daten und Paketvolumen des Stroms passen.

- **Signaturen in neuem Kanal** Das ist die Idee, die am schönsten klingt. Leider ist es auch die am schwierigsten umsetzbare Idee, da sie viele Variablen hat. Eine Variable ist der Client.
  Im Client muss, wenn man die Anfrage erstellt, einen weiteren Kanal in die SDP Nachricht einfügen. Das geht mit Mjsip recht leicht, da man hier über eine Funktion *addMediaDescriptor()* einer SDP Nachricht leicht weitere solcher Media-Deskriptoren hinzufügen kann. Im Falle, dass man eine SIP-Anfrage beantwortet, überprüft man die dort angegebenen Media-Deskriptoren und fügt für die Antwort auf die Anfrage die jeweils eigenen hinzu (z.B. Audio + Applikation). Man sollte sich nur noch Gedanken über die Portauswahl machen, ich finde "default-audio-port+1", also 21001, recht praktisch für den Signierkanal.
  Die zweite Variable sind die Server. Installiert man sich einen eigenen SIP-Server, eine komplette Telefonanlage, z.B. Asterisk, so kann und muss man viel einstellen und konfigurieren. Man kann sich hier den Signierkanal, also den zweiten Media-Strom, selbst konfigurieren und ermöglichen. Ganz anders sieht es da bei gehosteten PBXes (Phone Branch System, Telefonanlage) aus. Sie sind sehr restriktiv was Media-Ströme, deren Klassifizierung und Art betreffen. Teilweise werden sie auch erst gar nicht durchgeleitet. Das stellt ein großes Problem dar und ist dazu sehr mühsam zu debuggen und nicht geeignet zur Entwicklung.

4.3.3 Integration - Wichtige Schnittstellen und Provider

Wo müssen die Änderungen im Quelltext von SipDroid gemacht werden? In der Klasse, die den Listener auf "Call" Ebene der Bibliothek implementiert. Nehmen wir für das folgende Beispiel mal an, der Angerufene signiere immer und der Anrufer archiviere (ein mögliches Szenario wie SipDroid eingesetzt werden könnte). Dazu muss in den oben beschriebenen Methoden folgene Änderung gemacht werden:

- **onCallIncoming** (Call call, NameAddress caller, String sdp, Message invite) diesen Aufruf wird der Angerufene ausführen, er hat den Anruf noch nicht angenommen. Wir verlangen hier:
  - Generiere Session-Key (Java's Krypto Library ist hier sehr praktisch, über ein KeyPair Object generieren wir ein Schlüsselpaar)
  - Aus dem Call Object werden die Eigenen Daten, bzw. die eigenen Account Infos bezogen.
  - Aus der Invite Message werden die Daten des Anrufers entnommen

  Die Daten für unser "Greeting" sind ab diesem Punkt vorhanden. Die nächste Aktion des Users ist "annehmen", diese Aktion löst im Code die Ausführung einer "accept()"-Methode und damit den offiziellen Start für den Signierprozess aus.

- **onCallRinging**(Call call, Message resp), diese Methode wird vom Anrufer (Archivierer) ausgeführt, an diesem Punkt ist klar: Das Gegenüber ist erreichbar. Der Archivierer kann die ersten Protokollschritte schon durchführen. Dazu gehört z.B. das Senden der Begrüßung "Greeting", denn die INVITE Nachricht ist angekommen und ein "Call-Rining" wurde als Response empfangen. Beide Seiten haben an diesem Punkt bereits Informationen ausgetauscht.

- **onCallAccepted**(Call call, String sdp, Message resp) Der Anrufer führt diese Methode aus, sobald das "accept()" beim Angerufenen ausgelöst wird. Ab hier beginnt das Gespräch offiziell, so wie es auch bei klassischen Telefonen ist. Der Algorithmus sollte nun im mittleren Teil des Protokolls sein. Es werden die Intervalle gebildet und die einzelnen Chunks ausgetauscht.

- Anschließend bleiben noch die Methoden **hangup()** sowie **onCallClosing()** und **onCall Closed()** als CallBack-Methoden, um das Gespräch und den Signierprozess sauber mit einer "Final"-Message zu beenden.

Das hier beschriebene Szenario mit Signierer und Archivierer ist nur ein Bespiel. Eine nützliche Anwendung kann SipDroid auch in zwei verschiedenen Versionen werden. Die eine Version ist die der Bank, sie archiviert immer. Die andere Version wäre die frei verfügbare, die signierende Version. So könnte die Bank entscheiden, ob sie archivieren möchte oder nicht. Der Kunde wäre permanent bereit zum signieren. Alternativ könnte man sich auch vorstellen, dass die Bank und der Kunde während des Gesprächs entscheiden, dieses zu archivieren. Dazu müssten dann z.B. über eigene SIP-Requests der Signierprozess eingeleitet und gesteuert werden.
Als letzte der Schnittstellen fehlt uns noch der Provider, bzw. die Provider. Sipdroid kapselt den Start von AudioSender und AudioReceiver in einer Methode *launchMediaApp()*. In diesen werden Sender und Receiver für den Audio Strom erzeugt und gestartet. Wie angedeutet erweitern wir nun die Konstruktoren der Threads um das Attribut *Provider p* und benutzen den neuen Konstruktor. Die Idee dabei ist, im Listener die Provider zu erzeugen, sie dann jeweils AudioSender und Receiver "schreibend" zu übergeben und anschließend die beiden Provider auch der Bibliothek "lesend" zu übergeben.
Als letztes fehlt nur noch das Speichern der gesammelten Daten, das ist allerdings dem Integrator überlassen, da das Speichern oft abhänging von der Plattform ist. Möglich und sinnvoll ist das Speichern der Daten in einer der CrtlSeq's, gleich im Anschluss zur Validierung.

5 Ausblick

- **Szenario** Hier gibt es viele Möglichkeiten, den Algorithmus weiter einzusetzen, ich beschränke mich jedoch auf die Möglichkeiten, die der App "SipDroid" noch offen stehen. Nehmen wir an, SipDroid sollte nun eingesetzt werden: SigningSipDroid. Die App sollte über Google's PlayStore verfügbar sein. Was ermöglicht man in der App? Eine denkbare Möglichkeit wäre hier, eine Version der App zu veröffentlichen, die das Signieren zulässt. Praktisch wäre das für Banken, da sie eine private (Archivierer) und eine öffentliche Versionen (Signierer für den Kunden) der App benutzen können.
  Ein anders Szenario für die App wäre: Der Angerufene signiert und der Anrufer archiviert. So könnte man, ohne großen Aufwand, die Rollen verteilen. Eine Version der App könnte also, je nach Situation, beide Rollen übernehmen. Praktisch ist dies z.B. für den "hausinternen" Gebrauch, um z.B. Absprachen zweier Führungspersonen zu archivieren. Denkbar wäre auch die Veröffentlichung dieser Version als eine App, die für Privatleute oder kleine Geschäfte gedacht ist. Es ist nur fraglich, inwiefern diese Technologien dort gebraucht werden.
  Das letzte Szenario, jedenfalls das letze von mir durchdachte, basiert auch auf dem Gedanken "eine Version der App kann beide Rollen übernehmen". Hier würde ich grundsätzlich eine Einstellung in der App empfehlen, welche die Grundhaltung definiert: Will ich als Signierer oder vorzugsweise als Archivierer auftreten. Außerdem sollte man die Möglichkeit haben, zu jedem Zeitpunkt des Gesprächs, die signierenden und archivierenden Funktionen ein und auszuschalten. Der Algorithmus muss nicht zwangsläufig mit dem Gesprächsbeginn gestartet werden, es könnte auch ein bliebiger Zeitpunkt im Gespräch als Beginn der Aufzeichnung angesehen werden. Fraglich ist nur, inwiefern das die Aussagekraft der Aufzeichnung als Beweismittel beeinflusst.

- **Infrastruktur** Hier sollte besonderes Augenmerk darauf liegen, möglichst bald in "echten" Umgebungen zu arbeiten. Momentan arbeite ich mit SipDroid und ausschließtlich "handverdrahtet" im eingenen WLAN, mit provisorischen PBX'es als SIP-Server. Dieser Aufbau eigenet sich zwar, wenn man zeigen möchte, dass der Algorithmus laufzeittechnisch unproblematisch ist. Allerdings ist es, mit Blick auf "echte" mobile Netze, ein eher unrealistischer Aufbau. Mein Vorschlag, an dessen Umsetzung ich arbeite, wäre ein schrittweises Vorgehen: Eigenes WLAN mit Internetzugang und gehosteter SIP-Server, feste Ports, kein Zusatz im SDP. Anschließend: Gleicher Aufbau, jedoch mit Zusatz im SDP (neuer Medien-Strom). Als letztes müsste man dann zeigen, dass die App über alle möglichen Szenarien (Netzwerke) hinweg arbeiten kann. Erst dann, finde ich, kann man die App als ausgereift genug ansehen, um sie

zu verbeiten.

- **Umsetzung "Action-Stack-Controlled" Threads im Signierer** Angesprochen hatte ich dies bereits, als ich den Signierer vorgestellt hatte. Im Grunde besteht hier keine wirkliche Notwengikeit, jedoch sollte dies im Sinne der Vollständigkeit noch geschehen. Es ergeben sich, genau wie beim Archivierer, die Vorteile, dass man während man Anfragen noch bearbeitet, neue Anfragen schon empfängt. Somit wären auch hier Senden und Empfangen entkoppelt und man könnte besser auf Fälle reagieren, die passieren, wenn die Berechnung der Antwort so lange dauert, dass bereits eine "resend"-Anfrage eintrifft. Es ist auch diskutabel, ob dies wirklich notwendig wird, wenn man das Protokoll mit zusätzlichen Empfangsbestätigungen (ACK's) versieht. Im Moment funktioniert es jedenfalls mit dem einfacherem Entwurf ohne Probleme.

- **Wiedergabe & Prüfung - Tools** Herr Hett hat hier einige gute Vorschläge für Tools gemacht und implementiert. Es gibt die Möglichkeit, die Aufnahme zu rekonstruieren, Signaturen zu überprüfen und fehlende RTP-Pakete im Strom grafisch darszustellen, um so Aussagen über die Gesprächsqualität treffen zu können. Denkbar wäre, dass ich versuche, mein Datenvormat (XML) umzuparsen und so die bestehenden Tools weiterzuverwenden. Ich würde diese Tools aber lieber neu, unter Android, implementieren. Ein sog. "Playback"-Tool für Android hätte den Vorteil, dass ich das Backend komplett mit Java entwickeln kann und anschließend nur ein halbwegs bedienbares Android-Frontend dazu basteln muss. Das Java-Backend kann dann auf Deskop-Systemen ebenfalls mit einer angepassten Oberfläche versehen werden.

- **Android & Zertifikate** Zertifikate und insbesondere Zertifikat-Behandlung ist in dieser Arbeit leider nicht wirklich ein Thema geworden. Ein grundlegendes Problem dabei ist, dass Android hier, nicht wie sonst, noch keine praktischen Mittel bietet, um installierte Zertifikate in Apps zu verwenden. Ich vermute, dass dies nachgereicht wird, da Android mehr und mehr auch als Geschäftsplattform heranwächst. Die primitivste Methode, und von diesem Fall bin ich ausgegangen, ist, die Zertifikate als Datei aus einem definierten Ordner beim Start der App einzulesen. Hier bietet Java ein entsprechendes Paket an, dass aus den eingelesenen Bytes ein "Certificate"-Objekt ableitet. Wünschenswert wäre hier jedoch: Eine einheitliche Behandlung von Zertifikaten durch Android selbst. Am Algorithmus ändert dies nichts. Das richtige Verwenden von Zertifikaten ist sehr wichtig, wenn man auf Realismus in den Demonstrationen abziehlt.

Glossar und Abkürzungsverzeichnis

Nicht-Abstreitbarkeit Die Nicht-Abstreitbarkeit unterscheidet man in Nicht-Abstreitbarkeit der Herkunft und Nicht-Abstreitbarkeit des Erhalts:

- Nicht-Abstreitbarkeit der Herkunft bezeichnet die Garantie, dass der Kommunikationspartner nicht abstreiten kann, der Urheber der Daten zu sein.FhI-SIT (2005)
- Nicht-Abstreitbarkeit des Erhalts besagt, dass der Partner nicht abstreiten kann, die Daten erhalten zu haben.FhI-SIT (2005)

Public & Private - Key Public und Private Key's, häufig als PKI (Public Key Infrasturkur) zusammengefasst, basieren auf folgender Idee: Ein Key-Paar besteht aus öffentlichem und privatem Schlüssel. Der öffentliche Schlüssel dient dabei zur Verschlüsselung und der private, geheime, Schlüssel der Entschlüsselung. Aus der Kenntnis des des öffentlichen Schlüssels ist der private Schlüssel nicht zu erschließen. Den öffentlichen Schlüssel kann man also munter weiter geben, ohne fürchten zu müssen, dass der private, geheime, Schlüssel daraus abgeleitet werden kann. Der Nutzen einer PKI ist also, dass mit einem öffentlichen Schlüssel Daten verschlüsselt werden können, die nur vom Besitzer des privaten Schlüssels wieder entschlüsselt werden können. Umgekehrt kann man Daten mit dem privaten Schlüssel verschlüsseln, die dann nur dem öffentlichen Schlüssel entschlüsselt werden können.

Signatur (elektronisch) "Elektronische Signaturverfahren beruhenm wie asymmetrische Verschlüsselungs- verfahrenm auf einem Schlüsselpaar, bestehend aus einem öffentlichen und einem privaten Signaturschlüssel. Während der öffentliche Signaturschlüssel für alle Kommunikationspartner zugänglich gemacht wird, muss der private Signaturschlüssel geheim gehalten werden. Ein Signaturverfahren besteht aus einer Signaturfunktion und einer Verifikationsfunktion. Der Signaturalgorithmus erzeugt aus den gegebenen Daten und dem privaten Signaturschlüssel einen meist kurzen Wert, die so genannte Signatur. Das Verifikationsverfahren dient zum Überprüfen von Signaturen. Es gibt an, ob die Daten und die vorgelegte Signatur zu dem öffentlichen Schlüssel passen, und verifiziert damit die Gültigkeit der Signatur." - FhI-SIT (2005)

Abbildungsverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

Bundesnetzagentur (2010). *Jahresbericht 2010*.

Bundesnetzagentur (2011). *Jahresbericht 2011*.

Campbell, B., J. Rosenberg, H. Schulzrinne, C. Huitema und D. Gurle (2002). *Session Initiation Protocol (SIP) Extension for Instant Messaging*. RFC 3428 (Proposed Standard).

Casner, S. und P. Hoschka (2003). *MIME Type Registration of RTP Payload Formats*. RFC 3555 (Proposed Standard). Obsoleted by RFCs 4855, 4856, updated by RFCs 3625, 4629.

FhI-SIT, Fraunhofer-Institut Sichere Telekooperation, NOVOSEC AG (2005). *Verschlüsselung und Signatur - Grundlagen und Anwendungsaspekte*.

Gamma, Erich, R. Helm, R. Johnson und V. John (1996). *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, Bonn, 1. Aufl. Design Patterns, 1995, Deutsche Übersetzung von Dirk Riehle.

Handley, M., V. Jacobson und C. Perkins (2006). *SDP: Session Description Protocol*. RFC 4566 (Proposed Standard).

Hett, Christian (2006). *Security and Non-Repudiation for Voice-over-IP conversations*.

Holmberg, C., E. Burger und H. Kaplan (2011). *Session Initiation Protocol (SIP) INFO Method and Package Framework*. RFC 6086 (Proposed Standard).

Knuth, Donald E. (1998). *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley, Boston, MA, USA.

Kuntze, Schmidt, Hett (2005). *Non-Repudiation in Internet Telephony*.

Landrock, Petersen (1998). *WYSIWYS? What You See Is What You Sign? Information Security Technical Report*.

Niemi, A. (2004). *Session Initiation Protocol (SIP) Extension for Event State Publication*. RFC 3903 (Proposed Standard).

Rescorla, E. und B. Korver (2003). *Guidelines for Writing RFC Text on Security Considerations*. RFC 3552 (Best Current Practice).

ROACH, A. B. (2002). *Session Initiation Protocol (SIP)-Specific Event Notification*. RFC 3265 (Proposed Standard). Updated by RFCs 5367, 5727.

ROSENBERG, J. (2002). *The Session Initiation Protocol (SIP) UPDATE Method*. RFC 3311 (Proposed Standard).

ROSENBERG, J. und H. SCHULZRINNE (2002). *Reliability of Provisional Responses in Session Initiation Protocol (SIP)*. RFC 3262 (Proposed Standard).

ROSENBERG, J., H. SCHULZRINNE, G. CAMARILLO, A. JOHNSTON, J. PETERSON, R. SPARKS, M. HANDLEY und E. SCHOOLER (2002). *SIP: Session Initiation Protocol*. RFC 3261 (Proposed Standard). Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141.

SCHAAD, J. (2005). *Use of the RSASSA-PSS Signature Algorithm in Cryptographic Message Syntax (CMS)*. RFC 4056 (Proposed Standard).

SCHAAD, J., B. KALISKI und R. HOUSLEY (2005). *Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 4055 (Proposed Standard). Updated by RFC 5756.

SCHULZRINNE, H. und S. CASNER (2003). *RTP Profile for Audio and Video Conferences with Minimal Control*. RFC 3551 (Standard). Updated by RFC 5761.

SCHULZRINNE, H., S. CASNER, R. FREDERICK und V. JACOBSON (2003). *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550 (Standard). Updated by RFCs 5506, 5761, 6051, 6222.

SOMMERVILLE, IAN (2007). *Software Engineering*. Addison-Wesley, München, 8. akutalisierte Aufl.

SPARKS, R. (2003). *The Session Initiation Protocol (SIP) Refer Method*. RFC 3515 (Proposed Standard).

VOIPSEC, BUNDESAMT FÜR SICHERHEIT IN DER INFORMATIONSTECHNIK (2005). *VoIPSEC Studie - Studie zur Sicherheit von Voice over Internet Protocol*.

WEICKER, KARSTEN (2007). *Evolutionäre Algorithmen*. Teubner, Stuttgart, 2. Aufl.

XIE, Q. (2003). *RTP Payload Format for European Telecommunications Standards Institute (ETSI) European Standard ES 201 108 Distributed Speech Recognition Encoding*. RFC 3557 (Proposed Standard).

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 31. Januar 2013 Hagen Lauer