Implementierung einer Experimentsteuerung auf Android Smartphones

Bachelorarbeit

von

Daniel Sathees Elmo

aus

Jaffna

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

Dezember 2013

Betreuer:
Norbert Goebel, M. Sc.

Zusammenfassung

Die vorliegende Bachelorarbeit beschäftigt sich mit der Implementierung einer vorhandenen Experimentsteuerung (*Stateful Experiment Control*) für das *Android*-Betriebssystem. Aktuell lässt sich die Experimentsteuerung mit Messgeräten verwenden, die auf *x86*-Hardware basieren. Durch die Portierung für die *Android*-Plattform lässt sich die Palette an verwendbaren Gerätearchitekturen erweitern, so dass mobile *ARM*-Maschinen an Experimtenten teilnehmen können.

Der in *C++* geschriebene Programmcode der Experimentsteuerung wurde unter Verwendung des *Android Native Development Kit (NDK)* in einer *Android*-Anwendungshülle integeriert, die in *Java* geschrieben ist. Die Interaktion zwischen *C/C++* und *Java*-Code erfolgt dabei durch das *Java Native Interface*.

Die entwickelte *Android*-Anwendungshülle berücksichtigt dabei die Anforderungen der Experimentsteuerung und stellt benötigte Funktionen bereit. Dabei mussten einige Hürden überwunden und Besonderheiten des *Android*-Betriebssytem berücksichtig werden. Um eine plattformunabhängige Weiterentwicklung und Verwendung der Experimentsteurung zu gewährleisten, wurde im Rahmen dieser Arbeit der Quelltext und das Protokoll zur Experimentbeschreibung angepasst.

Ziel bei der entwickelten *Android*-Applikation war die Verwendbarkeit der Experimentsteuerung mit beliebigen *Android*-(Mess-)Anwendungen. Insbesondere sollte die Verwendbarkeit der Experimentsteuerung mit einem bereits für *Android* portierten Messframework (*Rate Measurement Framework*) ermöglicht werden.

Die entwickelte Anwendungshülle erzeugt ein *Overhead* über die eigentliche Experimentsteuerung führt zu einer zusätzlichen Systembelastung und Verzögerung bei der Ausführung von Befehlen. Um sicherzustellen, dass diese Faktoren in einem vernachlässigbaren Bereich liegen, wurden entsprechende Messungen durchgeführt.

Danksagung

Zunächst möchte ich mich ganz herzlich bei meinem Betreuer, Norbert Goebel, für die sehr gute Betreuung und die interessanten, fachlich anregenden Diskussionen in den vergangenen drei Monaten bedanken.

Bei Tobias Amft und Malte Olfen möchte ich mich für die Unterstützung und Beantwortung von Fragen bezüglich ihrer Abschlussarbeiten bedanken.

An meine Familie, Freunde und Bekannte geht ein großer Dank für die Unterstützung während der Bearbeitungszeit!

Ganz herzlich bedanke ich mich bei Sigfried Klemm, Ralph-Gordan Paul und Konstantin Kosin für das Korrekturlesen und Feedback.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

1.1 Motivation

In der heutigen Zeit ist Mobilität ein lebensbestimmender Factor. Besonders das Automobil hat als industrielles Massenprodukt den Alltag der Menschhheit verändert und ist nicht mehr wegzudenken. Mit zunehmender Verbreitung von Kraftfahrzeugen und der damit steigenden Verkehrsdichte, besteht ein erhöhter Bedarf an Sicherheit im Straßenverkehr. Während früher Bestandteile eines Automobils mechanischer Natur waren, finden heutzutage immer mehr elektronische Komponenten ihren Weg in die Fahrzeugtechnik. Moderne Sicherheitssysteme wie das Antiblockiersystem (*ABS*), Eletronisches Stabilitätsprogramm (*ESP*) oder *Airbag* sind elektronisch gesteuert und erhöhen maßgeblich die Sicherheit der Fahrzeuginsassen.

Die Eletrotechnik führte auch zur Entwicklung morderner Kommunikationstechniken, die in den letzen Jahren neue Wege eröffneten um die Effizienz und Sicherheit im Straßenverkehr weiter zu erhöhen und die Zahl an Verkehrunfällen und Toten zu senken. Als Datenübertragungstechnik konkurieren im Forschungsgebiet der Fahrzeug-zu-Fahrzeug Kommunikation (Car2Car Communication) *WLAN 802.11p* auf der einen Seite mit dem klassischen, infrastrukturbasierten Mobilfunk (*GSM, GPRS, UMTS*) auf der anderen Seite. Der Lehrstuhl für Rechnernetze entwickelt zu diesem Zweck ein Verfahren zur Simulation von Mobilfunknetzwerken für die Fahrzeug-zu-Fahrzeug Kommunikation auf

Basis von Messungen.

In einer vorangegangen Arbeit wurde eine Experimentsteuerung (*Stateful Experiment Control*) entwickelt, die eine kontrollierte und reproduzierbare Durchführung von Messexperimenten erlaubt. In einer weiteren Arbeit wurde zur Messung von Paketverlusten, Daten- und Latenzraten das sogenannte *Rate Measurement Framework (RMF)* entwickelt. Das *RMF* erlaubt Messungen in Mobilfunknetzwerken unter Bewegung durchzuführen. Sowohl *SEC*, als auch *RMF* wurden ursprünglichen für *x86*-basierte *Alix*-Geräte entwickelt. Die Verwendung von Smartphones für Messungen wurde zunächst aufgrund der fehlende *GPS Puls-Per-Second*-Unterstützung ausgeschlossen. Es kann keine ausreichend genaue Zeitsynchronisation vorgenommen werden, um Latenzmessungen mit brauchbaren Ergebnissen durchzuführen. Durch Kopplung geeigneter *GPS*-Empfänger an Smartphones, können theoretisch *PPS*-Signale zugeführt werden und auf Software-Ebene verarbeitet werden. Des Weiteren bieten Smartphone-Geräte die Möglichkeit Messungen flexibler und mobiler als mit *Alix*-Geräten durchzuführen. Die gesammelten Messdaten werden für die Simulation von Mobilfunknetzwerken benötigt. Zusammen mit der Aussicht eine breitere Masse an mobilen Endgeräten für Messungen verwenden zu können, wurde im Rahmen einer Arbeit am Lehrstuhl für Rechnernetze und Kommunikation das *Rate Measurement Framework* für *Android*-Geräte portiert.

Mit *RMF* auf *Alix*-Geräten durchgeführte Messreihen werden mit *SEC* ausgeführt, welches kontrollierte und zeitgesteuerte Experimente ermöglicht. Um vergleichbare Messungen mit Smartphone-Geräten durchführen zu können, ist es naheliegend als nächsten logischen Schritt, die *Client*-Komponente von *SEC* ebenfalls für die *Android*-Plattform zu portieren. Das Ziel dieser Arbeit ist daher eine Umsetzung des *SEC-Clienten* als *Android*-Anwendung, die eine kontrolliert gesteuerte Nutzung nahezu beliebiger Software auf *Android* ermöglicht.

1.2 Struktur der Arbeit

Zunächst wird im Kapitel 2 auf die Arbeiten eingegangen, auf denen diese Arbeit aufbaut. Im folgenden Kapitel 3 werden einige Grundlagen zur *Android*-Plattform, dem

Android SDK und dem *Android NDK* erörtert. In Kapitel 4 werden anhand der Anforderungen, die der *SEC-Client* an die Laufzeitumgebung stellt, verschiedene Möglichkeiten zur Umsetzung disktutiert und anschließend Designentscheidungen getroffen. Auf Basis des erstellten Designentwurfes wird in Kapitel 5 der Implementationsprozess beschrieben. Dieser war mit einigen Hürden verbunden, die überwunden werden mussten. Anschließend folgt in Kapitel 6 eine Gegenüberstellung der verwendeten Implementierungstechnik mit einer alternativen Methode zur Erstellung von Programmen, die ebenfalls auf *Android*-Geräte ausgeführt werden können. Messungen und Auswertungen zur Evaluierung der durchgeführten Implementierung werden im Kapitel 7 behandelt. Im letzten Kapitel 8 wird diese Arbeit mit einer Zusammenfassung und Vorschlägen zur Weiterentwicklung der erstellen Anwendung abgeschlossen.

Kapitel 2

Verwandte Arbeiten

Damit die *Android RMF*-Anwendung zusammen mit *SEC* in Experimenten eingesetzt werden kann, muss die *Client*-Komponente von *SEC* ebenfalls als lauffähiges Programm für die *Android*-Plattform umgesetzt werden. Die Möglichkeiten *SEC* in *Android*-Systemen verfügbar zu machen und die anschließende Implementation sind Kernbestandteile dieser Arbeit. Neben dem Einsatz von *SEC* für beliebige *Android*-Anwendungen, steht die Verwendbarkeit zusammen mit der *Android RMF*-Anwendung im Vordergrund.
Die vorliegende Arbeit basiert auf zwei vorangegangene Arbeiten. Wesentliche Aspekte dieser Arbeiten werden im folgenden näher erläutert.

2.1 Stateful Experiment Control

Statefule Experiment Control [Amf12] ist eine Experimentsteuerung, die in erster Linie entwickelt wurde um Mobilfunkmessungen mit mehreren Geräten durchführen und koordinieren zu können. *SEC* besteht dabei aus einer *Server*- und *Client*-Komponente. Auf den Geräten, die an Experimenten teilnehmen, wird der *SEC-Client* ausgeführt. Es wird eine Verbindung mit dem *SEC-Server* aufbaut, um Anweisungen entgegenzunehmen und diese auf dem Zielgerät durchzuführen. Messungen werden dabei nicht vom *SEC* selbst durchgeführt, sondern vielmehr startet der *SEC-Client* zeitgesteuert ein gewünschtes Programm, welches die eigentliche Messung durchführt. Dabei gibt es die

Möglichkeit, während der Ausführung eines Messprogrammes über eine *TCP-Loopback*-Schnittstelle weitere Anweisungen an diesen zu schicken. Dieses Features muss jedoch explizit von der Messsoftware unterstüzt werden.

Die durchzuführenden Experimente werden über *XML*-basierte Dateien beschrieben. Dabei kann ein Experiment in beliebigen Abschnitte unterteilt werden, und die Anzahl der Teilnehmer flexibel geändert werden. Des Weiteren können Zeitpunkte der ausgeführten Befehle beliebig angegeben werden. Das Protokoll zur Experimentsteuerung berücksichtigt den Einsatz von fehleranfälligen Kanälen. Beispielsweise sind beim Mobilfunknetz hohe Latzenraten und Paketverluste häufig auftretende Vorkommnisse.

SEC-Server und *-Client* wurden als *Linux*-Konsolenanwendungen in *C++* geschrieben und verwenden neben der *C++ Standard Bibliothek* die *Linux Socket API* und *POSIX Threads*.

2.2 Rate Measurement Framework für Android

Ein Messprogramm, welches beispielsweise mit *SEC* verwendet werden kann, ist das *Rate Measurement Framework*. Es wurde entwickelt um Ende-zu-Enden Messungen in Mobilfunknetzwerken durchzuführen. Operationen, die zum Versenden und Empfangen von Daten, der Messung von Datenraten, Paketverlusten und Verzögerungszeiten notwendig sind, werden vom *Framework* bereitgestellt. Durch Bereitstellung dieser Funktionalitäten können neue Messalgorithmen auf einfache Art und Weise implementiert werden.

RMF wurde ebenfalls in *C++* für die Ausführung in *Linux*-basierenden Systemen geschrieben und konnte erfolgreich in der Arbeit [Olf13] als Applikation für die *Android*-Plattform portiert werden. Wie in Abschnitt 1.1 beschrieben wurde, können aufgrund der fehlenden *Pulse-Per-Second (PPS)*-Unterstützung, keine verwertbaren Latenzmessungen durchgeführt werden. Durch Verwendung eines *GPS*-Gerätes, welches *PPS*-Signale in *Android*-Gerät zur weiteren Verarbeitung einspeist, ist es theoretisch möglich, dieses Problem zu lösen.

Kapitel 3

Grundlagen

Die Portierung des *SEC-Clienten* erfolgte im Rahmen dieser Arbeit für das mobile Betriebssystem *Android*. Daher werden im folgenden Kapitel zunächst einige Grundlagen dazu behandelt. Zudem wird die verwendete *Android*-Entwicklungsumgebung vorgestellt. Der letzte Abschnitt des Kapitels stellt das *Native Development Kit (NDK)* vor, welches ermöglicht, nativen *C* und *C++ Code* (im Folgenden: *C/C++* oder nativer Code) in *Android* Applikationen einzubinden. Des Weiteren wird auf das von der Open-Source Gemeinschaft weiterentwickelte *CrystaX NDK* eingegangen.

3.1 Android

Als *Android* wird sowohl ein Betriebssystem, als auch eine *Java-Software-Plattform* für mobile Endgeräte bezeichnet. Unter dem offiziellen Projektnamen *Android Open Source Project* wird *Android* von der *Open Handset Alliance (OHA)*[1] [Ope13] entwickelt. Mit dem Ziel, einen offenen Standard für mobile Geräte zu etablieren, wird *Android* seit Oktober 2008 [Dav13] als freie und quelloffene Software angeboten. Diese Merkmale führen zu einer großen Beliebtheit von *Android* bei Hardware-Herstellern, welches kostenlos und nach Belieben anpasst als mobiles Betriebssystem einsetzt werden kann. Inzwischen

[1] Von Google gründetes und geleitetes Konsortium, bestehend aus Firmen aus dem Bereich Software, Mobiltelefon, Mobilfunk, Hardware und Marketing.

ist *Android* weitverbreitet und besaß als Smartphone-Betriebssystem im zweiten Quartal 2013 einen Marktanteil von 79,3 Prozent [Hei13]. Laut Google werden täglich mehr als 1,5 Millionen *Android*-Geräte aktiviert [Prz13]. Unter den verschiedenen *Android*-Versionen ist *Jelly Bean* (Version 4.1.x - 4.3) aktuell am weitesten verbreitet [Goo13]. Auf den Smartphones, die zu Testzwecken in dieser Arbeit zum Einsatz kamen, wurde ebenfalls *Android Jelly Bean* als Betriebssystem verwendet. Nähere Details zu den Testgeräten werden im Abschnitt 7.1 beschrieben.

3.1.1 Systemarchitektur

Die *Android*-Systemarchitektur lässt sich in vier Schichten unterteilen (siehe Abbildung 3.1), die in den nächsten Abschnitten näher beschrieben werden.

3.1.1.1 Kernel

Der Kern des Betriebssystem basiert auf einem *Linux-Kernel*. Anfangs wurde dabei die Version 2.6 des *Linux-Kernels* verwendet. Mit *Android* 4.x (*Ice Cream Sandwich*) basiert das Betriebssystem auf den *Linux-Kernel* der 3.x-Serie. Der *Kernel* wurde dabei modifiziert und optimiert, so dass dieser performant auf mobilen Geräten läuffähig ist. Während der ursprüngliche *Linux-Kernel* auf die freie Implementation der *GNU C-Standard-Bibliothek*[2] *Glibc* setzt, wird beim *Android-Kernel* eine eigene von Google entwickelte Bibliothek namens *Bionic* verwendet. *Bionic* ist vom Funktionsumfang kleiner als *Glibc* und wurde speziell für eingebette Geräte und deren *CPUs*, wie *ARM* Prozessoren angepasst.
Der *monolithische Kernel* bildet die Hardwareabstraktionsschicht und beinhaltet neben einer optimierten Energie-, Speicher- und Prozessverwaltung, die Treiber für die verschiedenen Hardwarekomponenten eines mobilen Gerätes. Ebenfalls im *Kernel* implementiert sind Funktionalitäten zur *Interprozesskommunikation (IPC)*. *IPC* ist ein Vorgang, der ständig im Betriebssystemen stattfindet. Beispielsweise kommunzieren An-

[2] Genormte Programmierbibliothek, die u.a. Funktionen für Ein- und Ausgabe, mathematische Operationen, Verarbeitung von Zeichenketten und Speicherverwaltung bereit hält.

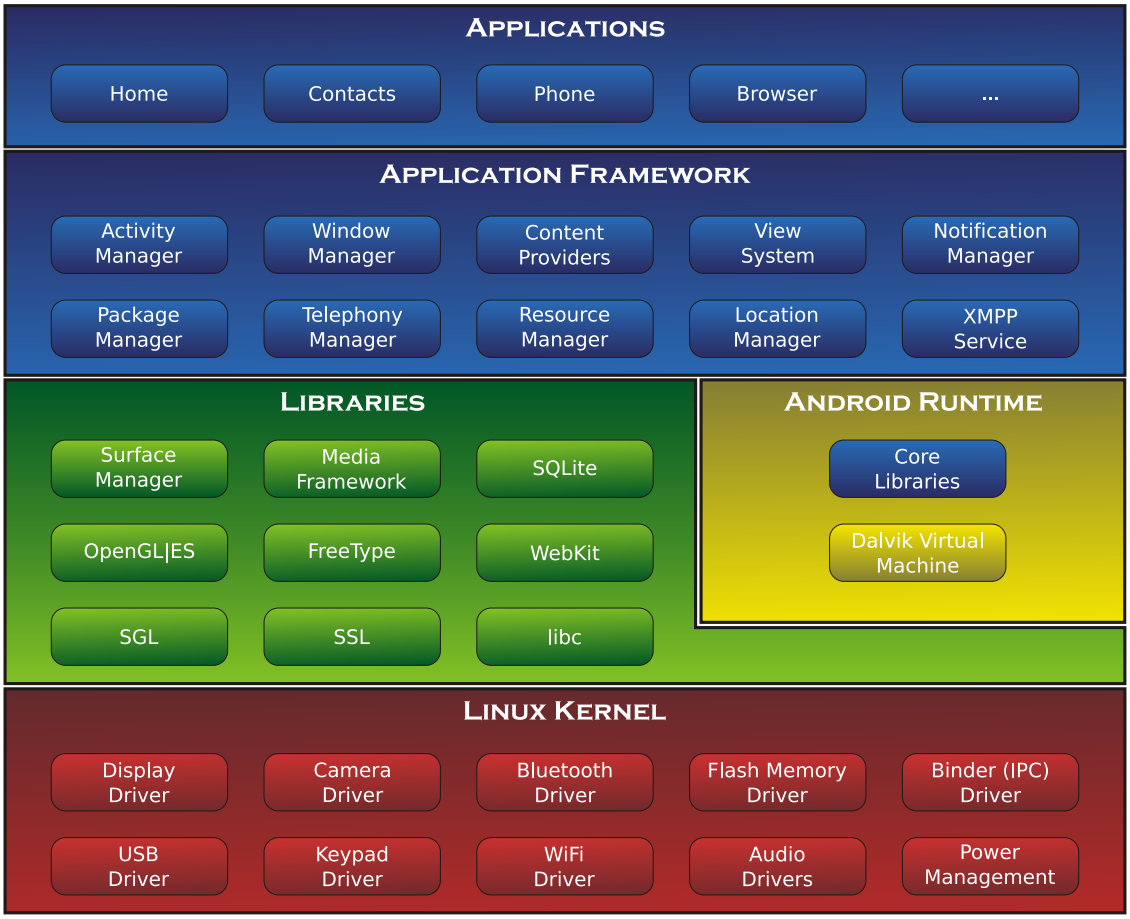

Abbildung 3.1: Die Schichten der Android-Systemarchitektur

wendungen untereinander, mit dem Betriebssystem oder mit verschiedenen Hintergrunddiensten. Obwohl *Android* auf dem *Linux-Kernel* basiert, wird für die Interprozesskommunikation nicht der *System V IPC*[3]-Mechanismus verwendet. Android verwendet zur Interprozesskommunikation ein spezifisches System mit dem Namen *Binder*[4] [Cin12]. Diese Technik wird auch für die Kommunikation innerhalb eines Prozess verwendet, um beispielsweise den Datenaustausch zwischen verschiedenen Komponenten einer Anwendung zu ermöglichen. Entwickler greifen zur Verwendung der *Binder*-Technik nicht direkt auf den *Kernel* zu. Funktionalitäten zur Kommunikation innerhalb und zwischen

[3] Erstmal in den 70er-Jahren für die *Unix*-Variante *Columbus Unix* entwickelte Technik die Kommunikation über Prozessgrenzen hinweg ermöglicht.

[4] Ursprünglich von *Be inc.* für das Betriebssystem *BeOS* entwickelt, später als *Open Source* Modul mit dem Namen *OpenBinder* veröffentlich. Um die *Apache Linzenz* zu erfüllen, wurde das *OpenBinder*-Projekt als *Android Binder* komplett neu geschrieben.

Prozessen werden im *Application Framework* (Abschnitt 3.1.1.3) gekapselt bereit gestellt.

Die Applikation, die im Rahmen dieser Arbeit entwickelt wurde, macht ebenfalls Gebrauch von der *Binder-IPC*-Technik um mit anderen Anwendungen zu kommunizieren. Genaure Details dazu sind im Design-Kapitel 4.2.7 und Implementations-Kapitel 5.2.3 zu finden.

3.1.1.2 Native Bibliotheken und Laufzeitumgebung

Auf dem *Kernel* aufsetzend befinden sich diverse Bibliotheken, die in *C/C++* geschrieben sind um geschwindigkeitsoptimierte Funktionen bereit zu stellen. Es werden unter anderem Bibliotheken zur Grafik (*OpenGL*)-, Webbrowser (*WebKit*)-, Netwerkwerk (u.a. *Secure Sockets Layer (SSL)*)-, Datenbank (*SQLite*)- und Medienwiedergabe (diverse *Codecs*) angeboten. Ebenfalls ist hier die bereits erwähnte *Bionic* C-Bibliothek angesiedelt. In *Java* verfasste *Android*-Anwendungen können nur indirekt über die darüber liegende Schicht auf die nativen Bibliotheken zugreifen. Die *Application-Framework*-Schicht stellt für den Zugriff auf die nativen Bibliotheken eine *Java*-Schnittstelle bereit.

Gleichermaßen über dem *Kernel* befindet sich die *Android* Laufzeitumgebung. Diese enthält die *Core Library* und die virtuelle *Java*-Machine *Dalvik*. Die *Core Library* enthält eine Untermenge der Klassenbibliothek *Apache Harmony*, einer freien *Java*-Implementierung. *Dalvik* ist eine speziell für *Android* entwickelte *virtuelle Java-Machine (JVM)*, die speziellen *Android-Bytecode* ausführt. Der Focus bei der Entwicklung von *Dalvik* war die effiziente Ausführung von Programmcode auf mobilen Endgeräte, die über begrenzte Ressourcen (langsame *CPU*, wenig Arbeitspeicher, kein *Swap-Speicher*) verfügen. Ein besondere Eigenschaft dabei ist, das die *Dalvik-VM* eine *Registermachine* ist. Das ist der Hauptunterschied zur *Java-VM*, die auf einem *Kellerautomaten* basiert. Moderne Prozessoren in Smartphones (*x86*-, *MIPS* und *ARM*-Prozessoren) sind ebenfalls *Registermaschinen*, so dass *Dalvik* eine effiziente Ausführung des *Java-Bytecodes* gewährleistet. Daher unterscheidet sich der *Bytecode* den *Dalvik* verarbeitet wesentlich vom normalen *Java-Bytecode* und muss vorher konvertiert werden (siehe nächster Abschnitt *Android* SDK 3.2). Seit der *Android*-Version 2.2 (*Froyo*) enthält *Dalvik* einen

Just-In-Time-Compiler (JIT), durch den die Ausführungsgeschwindigkeit weiter gesteigert werden kann. Jede *Android*-Anwendung wird als eigener Prozess mit jeweils einer eigener *Dalvik-VM* Instanz ausgeführt. Dadurch wird die Stabilität und Sicherheit des *Android*-Betriebssystems erhöht.

Um den damit gleichzeitig ansteigenden Ressourcenbedarf entgegen zu wirken und gleichzeitig performant zu bleiben, verwendet *Android* den sogenannten *Zygote*-Prozess, um Anwendungen zu starten. Der *Zygote*-Prozess wird beim Systemboot gestartet, initialisiert eine *Dalvik-VM*-Instanz und lädt wichtige Anwendungsbibliotheken. Um eine *Android*-Applikation zu starten wird der *Zygote*-Prozess *geforked*. Dies führt zu einem schnelleren Start der Applikation. Gleichzeitig nutzt der durch *Forking* entstandene Prozess für den Zugriff auf Grundbibliotheken den selben Speicherbereich, den auch der Elternprozess verwendet. Daraus resultiert ein verminderter *Overhead*, der sich ressourcenschonend auf das System auswirkt.

3.1.1.3 Android Application Framework

Das *Application Framework* enthält die Programmierschnittstelle für Entwickler, und abstrahiert die darunter liegenden Schichten. Es ermöglicht das Erstellen von Anwendungen mit grafischer Benutzeröberflächer und vermittelt den Zugriff auf die Gerätehardware (Kamera, WLAN, Sensor, Tastatur). Ein Rechtesystem steuert dabei, was einer Applikation erlaubt ist. Ein Kernkonzept des *Application Framework* ist, dass Anwendungen ihre Funktionen transparent offen legen können und damit für andere Andwendungen verfügbar machen können. Dadurch, dass Anwendungen von Drittanbietern gegenüber den fest integrierten *Android*-Anwendungen gleichgestellt sind, können für verschiedene Basisfunktion wie das Telefonieren, das Versenden einer *Mail* oder *SMS* verschiedene Anwendungen genutzt werden.

3.1.1.4 Anwendungsschicht

Die oberste Schicht bildet die Anwendungsschicht. Hier befinden sich die *Android*-Anwendungen, die vom Benutzer verwendet werden können. Dazu gehören neben den

mitgelieferten Systemanwendungen auch Programme von Drittanbietern. Diese können entweder über *Application Stores* wie dem *Google Play Store*, manuell als *Apk*-Archivdatei oder über *ADB* (Abschnitt 3.2) installiert werden.

3.2 Android SDK

Um *Android*-Anwendungen zu erstellen, wird von Google das *Android Software Development Kit (SDK)* zur Verfügung gestellt. Dieses beinhaltet neben Bibliotheken, APIs, Betriebssystem-Images und USB-Treibern, eine Vielzahl von Entwicklerwerkzeugen. Durch den *Android Virtual Device (AVD) Manager* ist es möglich, verschiedene virtuelle Geräte zu erstellen, die unterschiedliche Hardwarekonfigurationen aufweisen. Diese können im *Android Emulator* mit einem beliebigen *Android*-Betriebssystem-Image gestartet werden, um darauf entwickelte Applikationen zu testen. Der *Dalvik Device Monitor Server (DDMS)* ermöglicht das Debuggen und zeigt alle Aktivitäten einer *Android*-App an.

Die *Android Device Bridge (ADB)* ist ein mächtiges und hilfreiches Tool und erlaubt den Zugriff auf (virtuelle) *Android*-Geräte. Es lassen sich damit eine Vielzahl von Operationen auf dem Gerät durchführen. Beispielsweise können über eine *Linux*-ähnlichen *Shell* Dateioperationen durchgeführt werden, Applikationen installiert, gestartet und beendet werden, das Gerät neugestartet oder gar der *Kernel* geflasht werden. Ferner erlaubt *ADB* den Zugriff auf den *Android*-Systemlogger *Logcat*, welcher stets im Betriebssystem aktiv ist. Jeder Prozess hat die Möglichkeit verschiedene Arten von Meldungen über *Logcat* auszugeben. Ein weiteres Dienstprogramm das über *ADB* aufgerufen werden kann, ist *dumpsys*. Es gibt Informationen zum Systemzustand aus. *ADB* selbst ist als Verbund von *Client/Server*-Komponenten zu verstehen, die auf dem Entwicklungsgerät (z.B. *PC* oder *Mac*) und dem Zielgerät (z.B. Smartphone oder Tablet) ausgeführt werden. Während der Entwicklungsphase stellte sich *ADB* als unverzichtbares und wichtiges Hilfsmittel heraus.

Ein weiteres, im *SDK* enthaltenes Werkzeug ist *Android Lint*. Es in der Lage, *Android*-Projekte zu analysieren und potenzielle *Bugs*, Performance- und Sicherheitsproble-

me aufzuzeigen. Ein wichtiges, nicht direkt sichtbares Tool ist das Programm *dx*. Da die *Dalvik-VM* keinen normalen *Java-Bytecode* ausführen kann, der im *Kellermachinen*-Format vorliegt, muss dieser vorher umgewandelt werden. Diese Aufgabe übernimmt *dx* und konvertiert den vom *Java*-Compiler erstellten *Bytecode* in das sogenannte *dex*-Format um (*Dalvik Executables*). Als letztes sei das *Android Developer Tools (ADT)* erwähnt, ein *Plugin*, um das *Android-SDK* für die Anwendungserstellung in der quelloffenen Entwicklungsumgebung *Eclipse* zu integrieren.

3.3 Android NDK

Das *Android NDK* [Off13b] wird von Google mit dem Zweck entwickelt, die Ausführung von nativen *C/C++* Code auf *Android*-Geräten zu ermöglichen. Im Gegensatz zum *Java-Bytecode*, der von der *Dalvik-VM* interpretiert werden muss, wird der kompilierte *C/C++* Code direkt auf dem Prozessor ausgeführt. Hauptmotivation für die Verwendung des *NDK* ist die Entwicklung von Spielen. Hier bietet es sich performance kritische Operationen wie aufwändige Grafik- oder Physikberechnungen in *C/C++* zu schreiben. Neben der Spieleprogrammierung gibt es weitere sinvolle Einsatzbereiche für das *NDK*. Bereits bestehende, in nativen Code verfasste Programme müssen nicht in *Java* umgeschrieben werden. Für die Portierung des *SEC-Clienten* entscheiden wir uns für den Einsatz des *NDK*. Die Gründe dafür werden im Design-Kapitel (siehe Abschnitt 4.2) dargelegt. Als *NDK*-Distribution nutzen wir das *CrystaX NDK* [Off13a], eine von der Open-Source Gemeinschaft weiterentwickelte Version des Google *NDK*.

Im nächsten Unterabschnitt folgt eine Gegenüberstellung des *Google NDK* mit dem *CrystaX NDK*. Es werden jeweils die Vor- und Nachteile beider Distributionen beschrieben.

3.3.1 Google NDK und CrystaX NDK

Das Google *NDK* ist nicht als einzelnes Tool zu betrachten, sondern vielmehr als Zusammenstellung verschiedenener Werkzeuge. Neben einer Ansammlung von *APIs*, *Cross-*

compilern, *Linkern*, *Debuggern*, *Build-Tools* und einem eigenen auf dem *GNU Make* basierendes Build-Systems, enthält das *NDK* einen Satz an nativen Bibliotheken. Darin enthalten sind unter anderem *C/C++* Grundbibliotheken, sowie Headerfiles zur Grafik-, Audio-, Speicher- und Fensterprogrammierung. Ebenfalls werden Funktionen zum Zugriff auf die verschiedene Sensoreingabemöglichkeiten eines Smartphones bereitgestellt. Anfangs enthielten die ersten Releases des Google *NDK* neben der C-Standardbibliothek, lediglich eine minimale *C++* Bibliothek. So fehlten Bestandteile der *C++ Standardbibliothek*[5] wie *STI*[6] , *RTTI*[7] und die Fähigkeit *C++ Exceptions* auszuwerfen. Zudem gab es keine Ünterstützung für *C++11* Code.

Mittlerweile wurde das Repertoire des Google *NDK* erweitert, so dass inzwischen verschiedene Implementationen der *C++* Standardbibliothek zur Verfügung stehen. Diese setzten zu unterschiedlichen Anteilen den *C++* Standard um. Hervorzuheben sei dabei die *GNU Standard C++ Library*, welche Ausstatungsmerkmale wie STL, RTTI und Exceptions besitzt. Durch die Verwendung neuerer Versionen des *GCC*[8] und *Clang*[9] Compilers, ist es aktuell möglich C++11 spezifischen Code größtenteils unverändert zu verwenden. Eine Liste der unterstützen C++11 Features kann für beide Compiler hier [C++13b] [C++13a] eingesehen werden. Um die aufgezählten Funktionalitäten nutzen zu können, müssen diese beim Google *NDK* explizit in der spezifischen *Android-Makefile* eingetragen werden.

Das CrystaX *NDK* [Off13a] ist eine von der Open-Source Gemeinschaft weiterentwickelte Version des Google *NDK* und stellt einen vollwertigen Ersatz dar. Es verfügt über nützliche Erweiterungen, die im Laufe der Zeit in das Projekt eingeflossen sind. Besonders hervorzuheben sei dabei die Verfügbarkeit einer weitgehend vollständig implementierten Standard *C++*(11) Bibliothek. Darüber hinaus enthalten die CrystaX Releases

[5] Eine standardisierte Programmierbibliothek, die neben den Sprachmitteln unter anderem verschiedene generische Container, Funktionsobjekte, generische Zeichenketten, den Zugriff auf Datenströme zur Verfügung stellt. In ihr ist auch die gesamte C-Standard-Bibliothek enthalten. Ebenfalls in ihr ist die gesamte Standardbibliothek der Programmiersprache C enthalten.

[6] Die Standard Template Library ist fester Bestandteil der Standard *C++* Standardbibliothek mit dem Focus auf generischer Programmierung mit Datenstrukturen und Algorithmen.

[7] Die Runtime Type Information ermöglichst es zur Laufzeit den Typ eines Objektes zu ermitteln.

[8] Vom GCC-Team entwickelte Compilersammlung für die Programmiersprachen *C*, *C++*, *Java*, Objective-C, Fortran, Ada und Go.

[9] Compilerfrontend des LLVM-Team für die Programmiersprachen *C*, *C++*, *Objective-C* und *Objective-C++*

stets die neusten Versionen des GCC und Clang Compilers. Im Gegensatz zum Google *NDK* sind viele Funktionen bereits aktiviert und müssen nicht im *Makefile* ausdrücklich angegeben werden. Ein weiterer Pluspunkt ist der raschere und flexibeler Support, den das CrystaX-Entwicklerteam im Gegensatz zu Google bietet. Da das CrystaX *NDK* mit der Implementierung fehlender Features und Erweiterungen aufwarten kann, die besonders für die *Android*-Version des *RMF* essentiell sind, liegen die Vorteile klar auf der Hand, das *CrystaX NDK* in dieser Arbeit zu verwenden.
Als Nachteil sei aufzuführen, dass die CrystaX Distributionen spezifische Bugs enthalten können. Des Weiteren erscheint ein neuer Release durch die Notwendigkeit der Anpassung erst einige Zeit später als das dazugehörige *Google NDK* Pendant.

3.3.2 Java Native Interface (JNI)

Damit eine *Java*-Anwendungen mit nativen *C/C++* Code zusammenarbeitet kann, wird das *Java Native Interface (JNI)* benötigt. Es stellt eine standardisierte Schnittstelle und Spezifikation seitens Oracle dar, die von virtuellen *Java*-Maschinen implementiert werden müssen. Es ermöglicht die Kommunikation in beide Richtungen. Nativer Code kann aus *Java* aufgerufen werden und *Java* Code kann aus *C++* Code aufgerufen werden. Für Letzteres stellt das JNI eine reflektive API [Rat12] zur Verfügung, die es ermöglicht, nahezu alle Operationen aus der *C/C++* Seite durchzuführen, die in *Java* möglich sind. Allerdings verlieren *Java*-Programme, die native Bibliotheken verwenden, zwangsläufig ihre Plattformunhängigkeit. Der native Code solcher Anwendung muss für unterschiedliche Systeme neu kompiliert werden.

Kapitel 4

Anforderungen und Designentscheidungen

Im diesem Kapitel werden zunächst die Anforderungen des *SEC-Client* beschrieben, die von der Laufzeitumgebung für eine funktionierende Ausführung erfüllt sein müssen. Im Abschnitt Designentscheidungen 4.2 wird zunächst erläutert, warum bei der Portierung des Quellcodes das *NDK* eingesetzt wurde. Insbesondere wird beschrieben, warum eine Java-Hülle für die Portierung des *SEC-Client*s auf *Android* eingesetzt wurde. Es folgt eine Diskussion über verschiedene Funktionalitäten des Application-Frameworks, die *Android* zur Verfügung stellt um die gestellten Anforderungen umzusetzten. Nach Abwägung von Vor- und Nachteilen verschiedener Implementationsmöglichkeiten werden Designentscheidungen gefällt. Zuletzt wird erläutert, warum zusätzlich eine Anpassung der *Android*-Version des *RMF* erforderlich ist.

4.1 Anforderungen

Der *SEC-Client* wurde in *C++* für *Linux* als Kommandozeilen-Anwendung geschrieben. Ein- und Ausgaben des Programms erfolgen über die Konsole. Für die Verbindung und Kommunikation mit der *Server*-Komponente von *SEC* werden *TCP/UDP*-Netzwerkverbindungen verwendet. Da Experimente zu beliebigen Zeitpunkten durchfür-

bar sein sollen, ist der *SEC-Client* als endlicher Zustandsautomat implementiert. Dabei wechselt dieser lediglich zwischen Zuständen und wird damit prinzipiell beliebig lange und ohne Unterbrechung ausgeführt. Des Weiteren ist der *SEC-Client* in der Lage, Dateien zu lesen und schreiben. Experimentdaten werden entgegen genommen oder Logdaten werden gespeichert. Durchzuführende Experimente werden in einer *XML*-basierten Datei beschrieben. Die darin enthaltenen Befehle werden vom *SEC-Client* zeitgesteuert ausgeführt. Das Starten eines (Mess-)Programmes erfolgt dabei über ein *shell*-Kommando. Wichtig hierbei ist, das der *SEC-Client* unabhängig von den durchgeführten Kommandos weiter im Hintergrund läuft um weitere Aufgaben durchzuführen und um den Ablauf des durchgeführten Experimentes zu überwachen. Der *SEC-Client* und die aufgerufenen Programme müssen daher parallel laufen.
Es ergeben sich zusammenfassend im wesentlichen folgende Operationen, die in der ausgeführten Laufzeitumgebung möglich sein müssen:

1. Programm Ein- und Ausgaben
2. Beliebig lange Laufzeit
3. Nebenläufigkeit der Programme
4. Netwerkkommunikation
5. Dateioperationen
6. Starten von beliebigen (Mess-)Programmen
7. Kommunikation mit (Mess-)Programmen

Im nächsten Abschnitt folgt eine Diskussion, wie diese Anforderungen an die *Android*-Laufzeitumgebung bei der Portierung des *SEC-Client*s bestmöglichst umgesetzt werden können. Die daraus resultierenden Designentscheidungen werden im Implementations-Kapitel 5 dieser Arbeit umgesetzt.

4.2 Designentscheidungen

Bevor die Anforderungen des *SEC-Clienten* an die *Android*-Laufzeitumgebung behandelt werden, muss eine grundsätzliche Designentscheidung bezüglich der Portierung des *C++* Quellcodes getroffen werden. Es ist abzuwägen, ob ein Umschreiben des Code in *Java* oder der Einsatz des *NDK* in diesem Fall sinvoller ist. Wie im Grundlagen-Kapitel 3.3 beschrieben wurde, bringen beide Methoden Vor- und Nachteile mit sich.

In *Java* verfasste *Android*-Anwendungen haben direkten Zugriff auf das *Android-Application-Framework*, und können die gesamte angebotene *API* nutzen. Jedoch kostet das Neuverfassen des Quelltextes in einer anderen Programmiersprache Zeit und erfordert bei jeder Weiterentwicklung des *SEC-Clienten* eine Anpassung des *Java*-Quellcode. Zudem haben *C/C++* und *Java* verschiedene Sprachkonzepte, die jeweils in der anderen Programmiersprache nicht verfügbar sind. So werden im nativen Code des *SEC-Clienten* Funktionszeiger und Mehrfachverbungen eingesetzt. Diese Sprachelemente sind in Java nicht verfügbar und forcieren eine stärke Abweichung des portierten *Java*-Code, um diese Funktionalitäten nachbilden zu können. Gleichzeitig wird dadurch die Fehleranfälligkeit erhöht und birgt die Gefahr, ein anderes Laufzeitverhalten als das ursprünglich in C++ verfasste Programm aufzuweisen.

Der Einsatz des *NDK* für die Portierung des *SEC-Clienten* bringt mehrere Vorteile mit sich. Das Umschreiben des nativen Quelltextes in *Java* entfällt und die damit verbundenen Probleme werden vermieden. Der *SEC-Client* verwendet Funktionen der *C++* Standardbibliothek nach dem *C++ 03* Standard (ISO/IEC 14332:2003) [Amf12]. Die Implementation der *C++* Standardbibiliothek im *NDK* decken die benötigten Standard-Funktionalitäten ab. Zusätzlich werden für *Threads* und die Netzwerkkommunikation *POSIX* Threads und *Sockets* benötigt. Auch diese Bibliotheken stehen im *NDK* zur Verfügung.

Vergleicht man die Optionen, die zum Portierung des *SEC-Client* zur Verfügung stehen, so überwiegen die erwähnten Nachteile beim Neuschreiben der Anwendung in *Java*. Aus diesem Grunde haben wir uns für die Verwendung des *NDK* entschieden.

Das *NDK* bietet zwei Möglichkeiten, nativen Code zu kompilieren. Zum einem lässt sich der *C/C++*-Code als komplett natives Programm im *Executable and Linkable Format (ELF)* kompilieren. Bei diesen *Executable* handelt es jedoch nicht um eine *Android*-Anwendung im eigentlichen Sinne. *ELF*-Programme werden nicht in der Anwendungsschicht (Abschnitt 3.1.1.4) sondern direkt auf dem *Linux*-Kernel ausgeführt. Das *Android*-System mit seinen Sicherheits- und Energiesparfunktionen werden dabei umgangen. Dadurch dass *Executables* nicht als *Zygote*-Kindprozess (Abschnitt 3.1.1.1) gestarten werden, fehlt die *Dalvik VM*-Instanz. Es können keine *JNI*-Operationen durchgeführt werden, um auf die Funktionalitäten des *Android-Application-Framework* zugreifen zu können. Des Weiteren sind aus Sicherheitsgründen *Partionen*, auf die gewöhnliche Nutzer zugreifen können, für die Ausführung von *Executables* gesperrt. Es sind Administratorrechte (*Rootrechte*) erforderlich, um *Executables* auf Systempartionen zu kopieren, auf denen die Ausführung von *ELF*-Dateien möglich ist. Unter Umständen müssen *Executables* als *Administrator* ausgeführt werden, so dass diese mit den größtmöglichsten Rechten versehenen in der Lage sind, das System zu beschädigen.

Gewöhnlich wird das *NDK* dazu eingesetzt, *C/C++*-Code zu einer *Shared Library* zu kompilieren, die in einer *Java Android*-Anwendung eingebettet und zur Laufzeit geladen wird. In diesem Fall handelt es sich um eine typische *Android*-Anwendung, welche nicht die genannten Nachteile einer *ELF*-Datei mitbringt. Diese Methode erfordert jedoch die Entwicklung einer *Java*-Hülle. Die Interaktion zwischen *C/C++* und *Java*-Code die *JNI*-Schnittstelle. Dieses hochsensible Interface erfordert teilweise komplexe Operationsschritte zur Kommunikation zwischen beiden Sprachen und ist zudem fehleranfällig.

Insgesamt überwiegen jedoch die Nachteile bei der Vewendung einer *Executable*, so dass wir uns entscheiden, eine *Java*-Anwendungshülle zu erstellen, welche den *C/C++*-Code des *SEC-Clienten* als dynamische Bibliothek einbindet. In Kapitel 6 werden beide genannten Verfahren zur Ausführung von nativen Code auf dem *Android*-Betriebssystem detailierter beschrieben und verglichen.

Die im vorherigen Abschnitt erwähnten Anforderungen an die *Android*-Plattform erfordern eine Anpassung der portierten *Android*-Anwendung. Um den nativen Code der Applikation so wenig wie möglich zu verändern, muss der Großteil der Anpassungen in der

Java-Hülle erfolgen. Damit der *C++*-Code des *SEC-Clienten* und die *Java*-Hülle miteinander in beide Richtungen kommunizieren können, müssen die entsprechenden *JNI*-Funktionalitäten sowohl auf der *Java*-Seite, als auch im nativen Code implementiert werden. Um eine Vermischung von *JNI*-spezifischen Code mit dem Code des *SEC-Client* zu vermeiden, empfiehlt es sich, den *JNI*-Code in einer separaten *C++* Datei auszulagern. Diese Datei stellt dem *SEC-Client* abgekapselt Funktionen zum Informationsaustausch bereit und führt alle notwendigen, teilweise komplizierten *JNI*-Operationen durch.
Die folgenden Designentscheidungen werden unter den genannten Gesichtspunkten getroffen.

4.2.1 Ein- und Ausgaben

Der *SEC-Client* ist als kommandozeilenbasiertes *Linux*-Programm entworfen. Ein- und Ausgaben der Anwendung erfolgen daher über diese Benutzerschnittstelle. Um in *Android*-Anwendungen eine Benutzeroberfläche zu ermöglichen, die eine Interaktion mit dem Benutzer erlaubt, stellt das *Android-Application-Framework* die *Activity*-Klasse (im folgenden Activity) zur Verfügung. Eine *Activity* enthält ein Fenster, welches typischerweise den gesamten Bildschirm mobiler Geräte ausfüllt. In diesem Fenster lassen sich mit verschiedenen *View*-Klassen wie *Buttons*, *Container* oder *Textfelder* eine grafische Benutzeroberfläche darstellen, die in der Lage ist, mit dem Benuter zu interagieren. Da sich meistens nicht alle Funktionen einer App auf den relativ kleinen Bildschirmen mobiler Geräte darstellen lassen, enthalten *Android*-Anwendungen meistens mehrere *Activities*. Neben *Services* (Abschnitt 4.2.2) und *Receivern* (Abschnitt 4.2.6 u. 4.2.7) gehören Activities zu den Kernkomponenten einer *Android*-Anwendung. In *Android*-Anwendungen ist die Ausführung von Programmcode an einer dieser Komponenten und deren Lebenszyklus gebunden. Die verschiedenen Zustände die eintreten können, werden durch verschiedene *Callback*-Methoden repräsentiert. Diese werden bei einem Zustandswechsel automatisch aufgerufen. So wird beim Initialisieren einer neuen *Activity* die alte *Activity* von einem aktiven in einen passiven Zustand versetzt. Unter Umständen wird diese vom *Android*-System komplett aus dem Speicher entfernt. Diese Vorgehensweise des Betriebssystems ist damit zu erklären, dass mobile Geräte über begrenzte Hardware-Ressourcen verfügen und ein sparsamer Umgang mit den verfügbaren Mitteln sich po-

sitiv auf die Gesamtperformance auswirkt. Beim alltäglichen Gebrauch von *Android*-Applikationen ensteht häufig der Eindruck, dass *Activities* stets im Hintergrund weiter vorhanden sind. Denn bei der der Rückkehr zu einer *Activity* findet man häufig den Zustand wieder, der vor dem Verlassen einer *Activity* herrschte. Der Grund dafür sind die erwähnten Callback-Methoden, die genutzt werden, um Daten zu sichern, bevor die *Activity* in den passiven Zustand wechseln oder entladen werden. Bei einer Fortführung oder Neustart der *Activity* werden die gespeicherten Daten verwendet, um den ursprünglichen Zustand wieder herzustellen.

Wie in 4.1 beschrieben, muss sicher gestellt werden, dass der *SEC-Client* unter *Android* beliebig lange und und ohne Unterbrechung läuft. Dadurch wird gewährleistet, dass dieser zu jedem beliebigen Zeitpunkt in der Lage ist, mit dem *Server* oder dem Messprogramm zu kommunizieren. Ausgeführter Programmcode in *Android* sind an Grundkomponenten wie *Activities*, *Services* oder *Receiver* (diese werden in folgenden Abschnitten vorgestellt) gebunden. *Activities* werden in *Android*-Applikationen im sogenannten *main-Thread* gestartet. Dieser wird häufig auch als *UI Thread* bezeichnet, da in diesem *Thread* ebenfalls alle Operationen bezüglich der grafischem Benuteroberfläche durchgeführt werden. Eine Ausführung des *SEC-Clients* innerhalb einer *Activity* würde daher zur Blockierung der *main/UI-Thread* führen. Zudem würde beim Entfernen der *Activity* aus dem Speicher, die fortwährende Ausführung des *SEC-Clienten* nicht mehr gewährleistet. Durch Erstellen eines *Threads* ist es möglich, die Ausführung des *SEC- Clienten* vom Haupt-*Thread* zu entkoppeln. Jedoch ergibt sich dadurch das Problem, dass bei einem Aufruf einer neuen *Activity* die *Activity*, in der der *SEC-Client* in einem separaten *Thread* läuft, in einen passiven Zustand wechselt. Unter Umständen kann die *Activity* aus dem Speicher entladen werden. Zwar würde der *SEC-Client* weiterlaufen, da er sich einem gesonderten *Thread* befindet, doch wären alle Referenzen zum Zugriff auf den *Thread* oder auf den *SEC-Client* verloren. Solche Referenzen lassen sich innerhalb der *Callback*-Methoden nicht speichern, die das *Application-Framework* zu Sicherungszwecken zur Verfügung stellt.

Zusammenfassend benötigen wir *Activities*, um eine Benutzeroberfläche in der Anwendungen zur Verfügung zu stellen, die Ein- und Ausgaben des *SEC-Clienten* vermittelt. Aufgrund der genannten Eigenheiten von *Activities*, darf die Ausführung des *SEC-Clienten* nicht daran gebunden sein.

4.2.2 Beliebig lange Laufzeit

Innerhalb einer *Android* Anwendung muss gewährleistet sein, dass der *SEC-Client* beliebig lange und ohne Unterbrechung läuft. Zudem dürfen andere Operationen wie *UI*-Interaktionen innerhalb der Anwendung die Ausführung des *SEC-Clienten* nicht beeinflussen. Im vorherigen Abschnitt wurde beschrieben, dass wir diese Forderungen durch Bereitstellen eines eigenen *Thread* innerhalb einer *Activity* erfüllen können. Wird jedoch eine neue *Activity* geladen und aktiv, z.B. bei einem Aufruf einer *Android* Anwendung durch den *SEC-Clienten*, so wechselt die *Activity*, in der der *SEC-Client* ausgeführt wird, in einen passiven Zustand Unter Umständen wird diese *Activity* vom *Android*-Betriebssystem aus dem Speicher entladen. Die Möglichkeit, auf den Thread zuzugreifen, wäre damit verloren. Eine weitere Basiskomponente von *Android*-Anwendungen, die dieses Problem nicht mit sich bringen, sind *Services*. Sie verfügen über keine grafische Oberfläche, und sind als Dienste zu betrachten, die unabhängig von Anwendungs- oder Benutzeraktivitäten im Hintergrund laufen. Zudem sind Services nicht an den Lebenszyklus von Activities gebunden. Daher eignen sich *Services* prinzipiell als Container für die Ausführung des *SEC-Clienten*. Allerdings wird ein *Service* gleichermaßen im *main/UI-Thread* ausgeführt wie eine *Activity* und würde den Haupt-*Thread* bei der Ausführung des *SEC-Clienten* blockieren. Daher muss innerhalb eines Services zusätzlich ein *Thread* erstellt werden, der die Nebenläufigkeit des *SEC-Clienten* sicherstellt.

Eine vom Benutzer gestartete *Android*-Anwendung beginnt stets mit einer grafischen Benuteroberfläche, die durch eine *Activity* repräsentiert wird. Um zusätzlich einen Service zu starten, muss dieser von der *Activity* explizit gestartet werden. Befindet sich der zu startende *Service* innerhalb der Applikation, so spricht man von einem *lokalen Service*. Generell werden Grundkomponenten[1] von *Android* Anwendungen wie *Activities*, *Services* oder *Broadcast-Receiver* durch sogenannte *Intents* gestartet. *Intents* sind Objekte, die neben Informationen zum Zielobjekt zusätzliche Daten enthalten können. Diese werden der Zielkomponente beim Aufruf übergeben. Weitere Details zu *Intents* sind in Abschnitt 4.2.7 und 4.2.7, sowie Implementationskapitel 5 zu finden. Mit Hilfe von *In-*

[1] Es existieren noch weitere Grundbausteine einer *Android*-Anwendung wie *ContentProvider*, die in dieser Arbeit jedoch nicht von Bedeutung sind. In der Literatur und der *Android*-Dokumentation wird des Weiteren nicht klar definiert, welche Funktionalitäten des *Android-Application-Frameworks* zu den Basiskomponenten einer Anwendungen gezählt werden. In dieser Arbeit werden lediglich *Activities*, *Services* und *BroadcastReceiver* zu den die relevanten Komponenten gezählt und erwähnt

tents ist es also möglich, ein Service von einer *Activity* aus zu starten. Jedoch laufen diese getrennt von einander und es besteht keine Verbindung zwischen beiden Komponenten, um miteinander zu kommunizieren. Das *Android-Application-Framework* stellt Funktionen bereit, um eine Verbindung zu einem Service herzustellen. Man spricht in diesem Fall von einem *Bound Service*, bzw. von einem *Client/Server*-Verhältnis. *Services* können sowohl mit *Activities*, als auch mit anderen *Services* Bindungen eingehen. Dabei kann ein *Service* theoretisch mit beliebig vielen *Clienten* Verbindungen eingehen. Wird eine Bindung hergestellt, so muss dem *Clienten* ein Objekt übergeben werden, welches das *IBinder*-Interface implementiert. Dieses Interface basiert auf der *Binder*-Technik (Abschnitt 3.1.1.1), die im *Android-Kernel* implementiert ist. Sie ermöglicht die Kommunikation zwischen Komponenten innerhalb und ausßerhalb eines Prozesses. Das *IBinder*-Interface ist hochkomplex, so dass es sich nicht empfiehlt, es selbst zu implementieren. Es sind daher Klassen zu verwenden, die das *IBinder*-Interface bereits implementiert haben oder von solchen ableiten.

In diesem und im letzten Abschnitt wurden *Activities*, *Services* und ihre Eigenschaften diskutiert, die wir nutzen können, um den bisher behandelten Anforderungen des *SEC-Clienten* an die *Android*-Laufzeitumgebung gerecht zu werden. Zudem wurde beschrieben, wie *Activities* und *Services* miteinander kommunizieren können.
Daraus ergibt sich nachfolgender Aufbau (Abbildung 4.1) für unsere *Android*-Anwendung. Die Benutzeroberfläche der Applikation wird in einer *Activity* implementiert. Damit der *SEC-Client* ungestört und ohne Unterbrechung im Hintergrund laufen kann, wird die Ausführung des nativen Code des *SEC-Clienten* in einen Thread innerhalb eines Services ausgelagert. *Activity* und Service gehen dabei Bindung ein, um miteinander zu kommunizieren. *Activity* und *Service* laufen im main/UI-Thread.

4.2.3 Nebenläufigkeit von Programmen

Wie in Abschnitt 4.2 beschrieben, muss gewährleistet sein, dass der *SEC-Client* und sowie von ihm aufgerufene (Mess-)Programme parallel ausgeführt werden. Eine nebenläufige Ausführung stellt sicher, dass beispielweise ein Messprogramm ohne Unterbrechung messen kann und somit Ergebnisse nicht verfälscht werden. Zudem muss der *SEC-Client* währenddessen in der Lage sein, weitere Aufgaben durchzuführen. Ist bei-

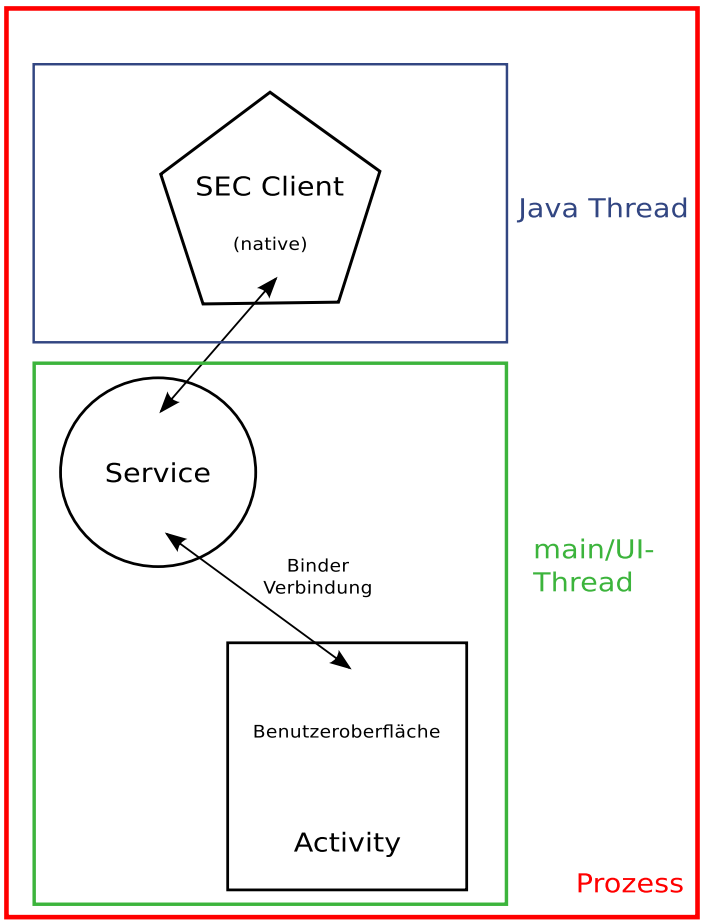

Abbildung 4.1: Design/Aufbau des Android SEC-Clients

spielweise die Ausführung von *Android*-Anwendungen, die vom *SEC-Client* aufgerufen werden sollen an *Activities* gebunden, so ergeben sich die in den vorherigen Abschnitten (4.2.1 und 4.2.2) beschriebenen Probleme. Es empfielt sich daher, auch für diese (Mess-)Programme, die vom *SEC-Client* in *Android* aufgerufen werden sollen, ebenfalls in einer solchen *Java*-Hülle zu integegrieren. Somit hätten wir folgenden Entwurf (Abbildung 4.2) für den *SEC-Clienten* und ein beliebiges (Mess-)Programm wie das *RMF*, das vom *SEC Client* genutzt werden soll.

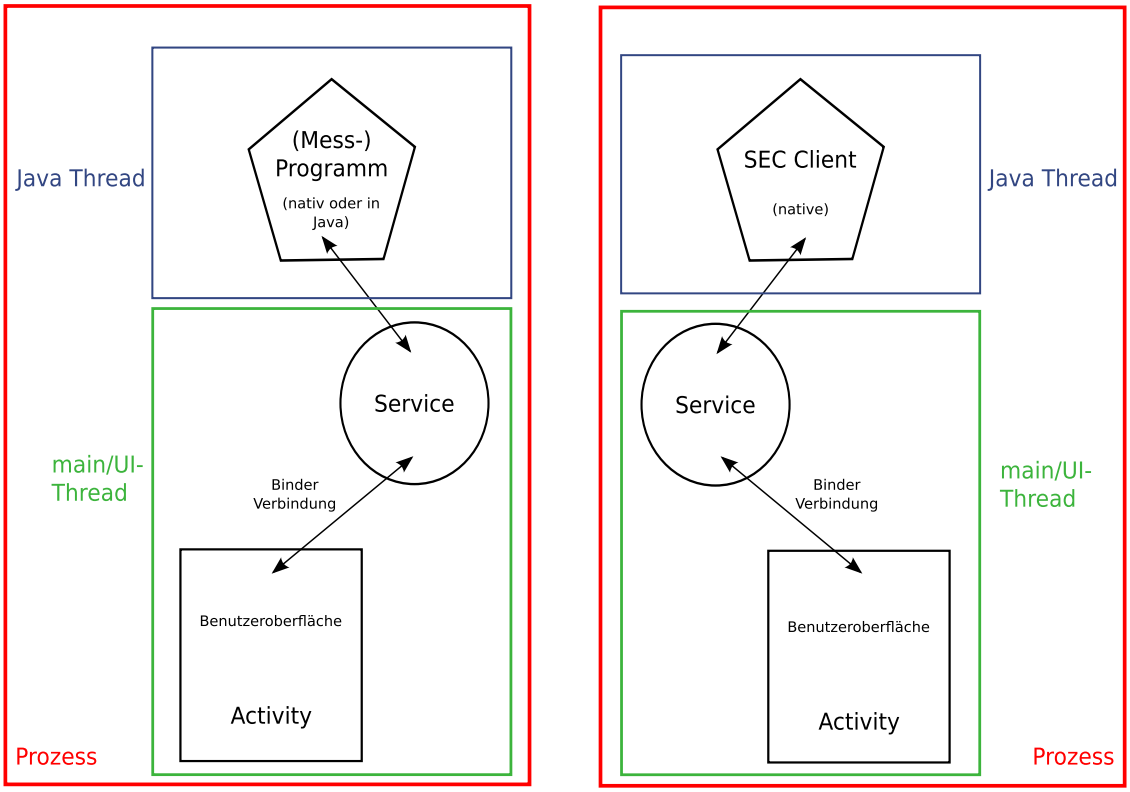

Abbildung 4.2: Design/Gegenüberstellung einer beliebigen (Mess-)Applikation mit dem Android SEC-Client

4.2.4 Netzwerkkommunikation

Eine der Hauptaufgaben des *SEC-Clienten* besteht darin, Aufgaben vom *SEC-Server* entgegenzunehmen, um diese auf dem Zielgerät durchzuführen. Dafür baut der *SEC-Client* eine *TCP*-Verbindung zum *Server* auf. Die Nutzung von *TCP* stellt dabei unter anderem die Integrität der übertragenden Daten sicher. Es wird gewährleistet, dass Aufgaben, die durchgeführt werden sollen, vollständig, unverfälscht und in der richtigen Reihenfolge am Zielgerät ankommen. Auch Logdaten über die durchgeführten Aufgaben werden nach Abarbeitung dieser per *TCP* zum *Server* geschickt. Während der Durchführung von Aufgaben kommuniziert der *SEC-Client* mit dem *Server* per *UDP*, um Statusnachrichten auszutauschen. Zwar bietet UPD keinen zuverlässigen Transport der Daten, jedoch hat eine fehlerhafte Datenübertragung kein Einfluss auf die Aufgaben, die durchgeführt

werden. Der *SEC-Client* nutzt für *TCP* und *UDP* Funktionalitäten *POSIX*-Sockets. Wie in 4.1 beschrieben, stehen die dazu notwendigen Bibliotheken im *NDK* zur Verfügung, so dass diese Funktionalitäten nicht mit Hilfe des *Android-Application-Frameworks* in der *Java*-Hülle des *Android-SEC-Clienten* implementiert werden müssen.

Allerdings verwendet *Android* ein Rechtesystem (Abschnitt 3.1.1.3) zur Steuerung der Anwendungsrechte. Ohne eine entsprechende Berechtigung ist beispielweise der Zugriff auf das Dateisystem oder die Kamera nicht erlaubt. Auch für den Zugriff auf Netwerkfunktionalitäten wird eine Berechtigung benötigt. Jede *Android*-Applikation enthält eine sogenannte *Android-Manifest*-Datei, in der *Metadaten* gepeichert werden. Diese im *XML*-Format vorliegende Datei enthält eine Vielzahl von Informationen über eine Anwendung, die dem Betriebssystem vor der Ausführung einer Anwendung bekannt sein müssen. Unter diesen *Metadaten* können sich unter anderem sogenannte *Permissions* befinden, die beschreiben, für welche Operationen die Anwendung eine Berechtigung haben möchte. Bei der Installation einer App werden diese Berechtigungsanfragen vom *Android*-Installationsprogramm aus der *AndroidManifest.xml*-Datei gelesen und dem Benutzer angezeigt. Zur erfolgreichen Installation einer Anwendung muss der User diesen Anfragen zustimmen, so dass *Android* der zu installierenden Anwendung die geforderten Berechtigungung erteilen kann. Dieses Rechtesystem in *Android* macht Anwendungen transparenter. Der Benutzer weiß vor der Installation einer Applikation, auf welche Resourcen diese zugreifen darf. Da der im nativen Code vorliegende *SEC-Client* innerhalb einer *Android-Java-Anwendungshülle* steckt, wird die gesamte Applikation auch vom System als normale *Android* Anwendung wahrgenommen und unterliegt damit auch dem beschrieben Rechtesystem von *Android*. Für den Zugriff auf Netzwerkfunktionalitäten wird daher eine *Permission* benötigt, die in der *AndroidManifest.xml* eingetragen werden muss.

4.2.5 Dateioperationen

Aufgaben, die der *SEC-Client* vom *Server* entgegennimmt, werden in Form von Dateien auf dem Zielgerät gespeichert. Auch Logdaten werden, bevor diese zum Server geschickt werden, zunächst in Dateien gesichert. Der *SEC-Client* nutzt für Dateioperationen Funktionen der Standard *C++* Bibliothek, die im *NDK* ebenfalls zur Verfügung

stehen. Auch hier müssen diese Funktionalitäten nicht in der *Java*-Hülle des *Android SEC-Clienten* bereitgestellt werden. Allerdings werden für Dateioperationen analog zu Netzwerkzugriffen eine Berechtigung vom *Android*-Betriebssystem benötigt. Die entsprechende *Permission* muss daher in der *AndroidManifest.xml* eingetragen werden.

4.2.6 Starten von beliebigen (Mess-)Anwendungen

Zu den Aufgabenbereich des *SEC-Clienten* gehört es, beliebige (Mess-)Programme aufrufen zu können, die sich auf der *Client*-Machine befinden. Intern verwendet das Programm dabei die *system()*-Funktion aus der *C*-Standard-Bibliothek *stdlib.h*. Unter Linux führt die Funktion einen gewünschten Befehl in der Kommando-Konsole aus und kann daher problemlos zum Starten von Programmen genutzt werden. Wie bereits in Abschnitt 4.2.2 beschrieben, werden im *Android* Betriebsystem sogenannte *Intents* eingesetzt, um Anwendungen und ihre Komponenten wie *Activities*, *Services* oder *Broadcast-Receiver* zu starten. Dieses Konzept ist *Android* spezifisch und steht in den nativen *NDK*-Bibliotheken nicht zur Verfügung. Daher müssen die Funktionalitäten zum Starten von beliebigen (Mess-)Programmen in der *Java*-Hülle implementiert werden. Für den Start der verschiedenen Grundbausteine einer Anwendung (*Activities*, *Services*, *Services*, *Broadcast-Receiver*) bietet die *Android-API* entsprechene Funktionen an, um *Intents* abzuschicken. Man unterscheidet dabei zwischen *expliziten* und *impliziten Intents*. Zum gezielten Aufrufen einer Komponente, deren genauer Name bekannt ist, werden *explizite Intents* verwendet. Möchte man keine spezifische Komponente aufrufen, so werden *implizite Intents* eingesetzt. Diese beschreiben Eigenschaften, welche die zu aufzurufende Komponente erfüllen muss. Eine Komponente, die auf *implizite Intents* reagieren möchte, muss sogenannte *Intent-Filter* implementieren. Das *Android*-Betriebssytem vergleicht die Daten eines *impliziten Intents* mit den *Intent-Filtern* aller installierten Anwendungen und erstellt eine Liste mit passenden Applikationen. Der Benutzer wird dann aufgefordert sich für eine gewünschte Anwendung zu entscheiden, die mit dem *impliziten Intent* gestartet werden soll. Diese Technik ist den meisten *Android*-Benutzer im alltäglichen Gebrauch vertraut. Beispielsweise erscheint innerhalb einer App beim Aufrufen einer Mail-Funktion ein Auswahlbildschirm mit allen verfügbaren Mail-Anwendungen (oder zumindest alle, die ein entsprechenden *Intent-Filter* implementiert haben). *Impli-*

zite Intents in unserer Anwendung zu verwenden macht wenig Sinn. (Mess-)Programme sollen durch den *SEC-Clienten* gezielt gestartet werden. Des weiteren erfordern *implizite Intents* Interaktionen seitens des Benutzers, was dem Prinzip einer unbeaufsichtigen Experimentsteuerung widerspricht. Wir werden daher *explizite Intents* verwenden, um (Mess-)Programme in *Android* zu starten.

Wie bereits erläutert, können mit Hilfe von *expliziten Intents* gezielt Bausteine einer Applikation aufgerufen werden. Wir haben daher die Möglichkeit, *Activities*, *Services* oder *Broadcast-Receiver* anzusprechen. Zwar sind *Brodcast-Receiver* nicht in unserem aktuellen Entwurf enthalten, doch der Vollständigkeithalber werden diese vorgestellt und diskutiert, ob deren Einsatz in unser Design sinnvoll ist. Wie in jedem anderen Betriebssystem auch, werden in *Android* eine Vielzahl von Nachrichten innerhalb des Systems versendet. Diese *Broadcasts* in Form von *Intents* informieren über bestimmte Ereignisse wie den Akkustand oder den Empfang einer SMS. Auch Anwendungen können solche *Broadcast*-Nachrichten verschicken. Applikationen, die einen *Broadcast-Receiver* enthalten, können mit Hilfe von *Intent-Filter* spezifizieren, auf welche *Broadcast-Intents* reagiert werden sollen. Bei einem passenden *Intent* wird der *Broadcast-Receivert* gestartet, der ähnlich wie *Services* über keine grafischen Benutzeröberfläche verfügt. Diese Technik könnten wir nutzen, um (Mess-)Programme durch den *SEC-Client* zu initialisieren. Dazu müsste ein *Broadcast-Receiver* mit einem spezifischen *Intent-Filter* im gewünschten (Mess-)Programm implementiert werden. Der *SEC-Client* müsste zum Starten des *Broadcast-Receivers* im (Mess-)Programmes einen passenden *Broadcast-Intent* verschicken. Der aktivierte *Broadcast-Receiver* startet schließlich das eigentliche (Mess-)Programm. Der Einsatz von *Broadcast-Receiver* zieht jedoch in unserem Fall mehrere Nachteile mit sich. *Broadcasts* sind global verschickte Nachrichten, so dass eine gewisse Zeit benötigt wird, um bei einem passenden Empfänger anzukommen und verarbeitet zu werden. Des Weiteren muss ein *Broadcast-Receiver* an den *Service* weiter deligieren, der das eigentliche (Mess-)Programm enthält. Dafür wird ein weiterer *Intent* benötigt, was erneut Zeit kostet. Des Weiteren haben *Broadcast-Receiver* einen völlig anderen Lebenszyklus, als *Activities* oder *Services*. Aktivierte *Broadcast-Receiver* werden nach relativ kurzer Zeit (ca. 10 Sekunden) wieder beendet. Gleichzeitig wird der dazugehörige *Dalvik-VM*-Prozess beendet. Damit werden auch andere Anwendungskomponenten wie *Activities* oder *Services* beendet, die ebenfalls im selben *Dalvik-VM*-Prozess ausgeführt werden. Um diesen Umstand zu vermeiden, müssen andere Anwendungskomponenten

vom *Broadcast-Receiver* in einem seperaten Prozess gestartet werden. Die Verwendung von *Broadcast-Receivern* für unsere Anwendung erweist sich daher als aufwändig und ist zudem nicht die schnellste Methode um Anwendungen zu starten.

Weniger kompliziert und effizienter ist das Starten einer *Activity* des aufzurufenden Programmes. Durch Verwendung eines *expliziten Intents* kann ein schnellerer Start gewährleistet werden, als dies bei *Broadcast-Receiver* der Fall ist. Jedoch hat auch diese Methode einige Nachteile. Zum einem muss zunächst die grafische Oberfläche einer *Activity* geladen werden, was Zeit kostet. Zum anderen muss hier ebenfalls an den *Service* deligiert werden, der das eigentliche (Mess-)Programm enthält.

Die am wenigsten aufwändigste und gleichzeitig effektivste Methode, um (Mess-)Programme in *Android* zu starten, ist die Verwendung von *expliziten Intents* um die *Service*-Komponente der Anwendung direkt zu starten. Es werden dadurch zeitraubende Zwischenstation vermieden, und der Implementierungsaufwand wird reduziert. Ein weiterer Vorteil, den nur *Services* bieten, ist die Möglichkeit, eine Bindung einzugehen, die zur Kommunikation zwischen dem *SEC-Clienten* und dem (Mess-)Programm genutzt werden kann. Dieses Verfahren wird bereits im bisherigen Design verwendet, um die Kommunikation zwischen *Activity* und *Service* zu ermöglichen. Im nächsten Abschnitt wird beschrieben, wie auf diese Technik zur *Interprozess-Kommunikation* zwischen der*Android-SEC-Client*-Anwendung und einem (Mess-)Programm verwendet werden kann.

4.2.7 Kommunikation mit (Mess-)Programmen

Eine letzte Anforderung an das *Android* Betriebssystem gilt es noch zu behandeln. Der *SEC-Client* muss in der Lage sein, mit einem gestarteten (Mess-)Programm zu kommunizieren. Der *SEC-Client* baut dazu eine lokale *TCP*-Verbindung mit dem (Mess-)Programm auf. Dieses Feature und das Kommunikation-Protokoll müssen dabei explizit von (Mess-)Programm implementiert werden. In Abschnitt 4.2.4 wurde erörtert, dass die dazu notwendigen Bibliotheken im *NDK* vorhanden sind, so dass lediglich eine entsprechende *Permission* zum Netzwerkzugriff in der *Android-Manifest*-Datei eingetragen werden muss.

Unter Android befinden sich sowohl der *SEC-Client*, als auch ein (Mess-)Programm jeweils in einer *Java*-Hülle. Die Kommunikation zwischen beiden Anwendungen findet direkt zwischen den eigentlichen ausgeführten (nativen-)Programmen innerhalb der Hülle statt. Die *Java*-Hüllen werden daher nicht involviert und es findet kein Datenaustausch zwischen diesen statt. Für die Verwendung des *SEC-Clienten* unter *Android* ist eine Kommunikation zwischen den *Java*-Hüllen nicht erforderlich. Aus folgenden Gründen kann es sinnvoll sein diese zu implementieren. Zum einem enthalten die *Java*-Hüllen wichtige Funktionalitäten, die von nativen Programme unter Umständen benötigt werden (Bsp. *Permissions* oder *Intents*). Sie bilden die Basis für die Ausführung der eigentlichen Programme innerhalb dieser Hüllen. Eine Verbindung zwischen den Hüllen kann beispielweise genutzt werden um den Zustand einer *Android*-Anwendung abzufragen oder Befehle zu senden. Des Weiteren kann diese Kommunikationschnittstelle genutzt werden, um Ausgaben des *SEC-Clienten* und eines (Mess-)Programmes innerhalb einer der beiden Anwendungen anzuzeigen. Im Gegensatz zu einem *Linux*-Desktop-PC kann in *Android* die Benutzeroberfläche nur einer Anwendung zur selben Zeit angezeigt werden.

Für die Kommunikation zwischen *Android*-Anwendungen gibt es mehre Möglichkeiten, von denen die wichtigsten hier genannt werden sollen. Ähnlich wie der *SEC-Client* könnte man eine *TCP*-Verbindung über eine lokale *Loopback*-Schnittstelle implementieren. Weitere *Android* spezifische Optionen zur Kommunikation bietet das *Application-Framework*, die im Folgenden vorgestellt und verglichen werden. Im vorherigen Abschnitt wurden *Broadcast-Receiver* vorgestellt. Diese könnten für die Kommunikation zwischen *Android* Anwendungen genutzt werden. Doch bringen diese neben den bereits geschilderten Nachteilen das Problem mit sich, dass unter Umständen aufeinander folgende Nachrichten aufgrund ihrer *Broadcast*-Natur verzögert beim Empfänger ankommen. Eine weitere Möglichkeit zum Datenaustausch zwischen zwei Apps bieten die ebenfalls vorgestellten *expliziten Intents*. Wir nutzen diese bereits zum Starten von *Activities* bzw. zum Starten der *Service*-Komponente innerhalb eines (Mess-)Programmes. Ein Versenden eines weiteren *expliziten Intents* an einen bereits gestarteten *Service* bewirkt, dass die entsprechende *Callback*-Methode innerhalb der *Service*-Klasse erneut aufgerufen wird. Da *Intents* in der Lage sind, zusätzliche Informationen zu speichern, können diese zum Datenaustausch zwischen zwei Anwendungen (in unseren Fall zwischen zwei *Services* genutzt werden. Als Nachteil ist anzuführen, dass durch die Nutzung von *In-*

tents ein *Overhead* um die eigentliche Nachricht entsteht, die ausgetauscht werden soll. In erster Linie wurden *Intents* designed, um Anwendungskomponenten einer *Android* Anwendung zu starten. Sie werden in der Regel nicht für einen stetigen Informationsaustausch genutzt.

Im Abschnitt 4.2.2 wurde das *IBinder*-Interface vorgestellt, das lokal innerhalb einer Anwendung zum Herstellen einer Verbindung zwischen einer *Activity* und einem *Service* genutzt wird. Die selbe Technik kann auch verwendet werden, um eine Bindung zwischen zwei *Service*-Komponenten verschiedener Anwendungen herzustellen. Über diese Verbindung kann ein direkter Datenaustausch zwischen beiden *Services* stattfinden. Allerdings ist hierbei auf eine Besonderheit zu achten. Da sich beide *Services* in verschiedenen Anwendung befinden, laufen diese auch in unterschiedlichen Prozessen. Aus Stabilitäts- und Sicherheitsgründen gewährt das *Android*-Betriebssystem ohne weitere Anstrengungen den Zugriff auf den Speicherbereich fremder Prozesse nicht. Zwar ist es möglich, ein *Service* aus einer anderen Anwendung explizit im eigenen Prozess starten zu lassen, doch beinhaltet diese Methode einen wesentlichen Nachteil. Ein Programmabsturz einer Anwendung führt zur Terminierung des *Dalvik*-Prozesses und damit auch zum Beenden anderer Anwendungen im selben Prozess. Daher müssen wir eine Interprozesskommunikation zwischen zwei *Services* durchführen. Das *IBinder*-Interface im *Application-Framework* zur *Service*-Bindung basiert auf der *Binder*-Technik und erlaubt problemlos die Kommunikation innerhalb eines Prozesses. Auch die Interprozesskommunikation ist damit möglich, erfordert jedoch weitere Anstrengungen in der Implementierung.

Für solche Zwecke stellt die *Android*-Plattform *AIDL* zur Verfügung. *AIDL* (*Android Interface Definition Language*) ermöglicht in einer *.aidl*-Datei eine einheitliche Schnittstelle für zwei Verbindungspartner (z.B. zwei *Services* aus unterschiedlichen Anwendungen) zu definieren, die Interprozesskommunikation durchführen möchten. Aus der *.aidl*-Datei wird mit Hilfe eines im *SDK* integrierten *Tools* eine *.java*-Klassendatei generiert, welche von der *Binder*-Klasse erbt (somit das geforderte *IBinder*-Interface implementiert). Zusätzlich sind die mit *AIDL* definierten Schnittstellen in der *.java*-Datei in Form einer inneren abtrakten Klasse enthalten. Durch Vererbung dieser Klasse werden die Schnittstellen konkret implementiert. Man erhält letztendlich eine auf *AIDL* basierende Klasse, die zusätzlich *IBinder*-Interface implementiert. Eine Instanz dieses Objektes wird bei einer

Bindung zu einem Client übergeben. Die Kombination aus *AIDL* und *IBinder*-Interface erlaubt, durch die *Binder*-Technik des Betriebssytems Interprozesskommunikation zwischen Anwendungen durchzuführen. *AIDL* übernimmt dabei alle notwendigen Operationen, um Daten, die über Prozessgrenzen transportiert werden sollen, in primitive Datentypen zu zerlegen. Diese können vom Betriebssystem interpretiert werden können und in den Speicherbereich eines anderen Prozesses tranportiert werden. Wichtig ist hierbei, dass der Kommunikationspatner ebenfalls in Besitz der spezifischen *.aidl* sein muss, um die genaue Schnittstelle zu kennen über welche die Kommunikation stattfinden soll. Ansonsten würde er nur wissen, dass die Referenz die er bei einer Bindung erhält, lediglich das *IBinder*-Interface implementiert. Mit Hilfe der *.aidl*-Datei kann eine Art *Typecast* durchgeführt werden, so dass die in *AIDL* klar definierten Schnittstellen genutzt werden können. Bei dieser Technik ist zu beachten, dass *AIDL*-Funktionsaufrufe aus dem *Thread*-Context des Aufrufers erfolgen. Somit muss insgesamt die Implementation der Applikationen *Thread*-sicher sein. Im Ganzen erweist sich die Verwendung von *AIDL* als komplex und aufwändig. Von Google wird der Einsatz dieser Technik nur für besondere Anlässe wie spezielle *Multithreading*-Anwendung empfohlen.

Glücklicherweise existiert eine weitere, weniger komplexere Möglichkeit zur Interprozesskommunikation. Es können sogenannte *Messenger* eingesetzt werden, um Nachrichten in Form von *Message*-Objekten zu einem bestimmten Empfänger zu versenden. *Messenger*-Objekte können Referenzen generieren, die das *IBinder*-Interface impelementieren und so zur Bindung zu einem *Service* genutzt werden können. Darüber hinaus basieren *Messenger* auf *AIDL* und können daher zur Interprozesskommunikation genutzt werden. Zudem bündeln *Messenger* Nachrichten in einer *Queue* und arbeiten diese in einem einzigen *Thread* ab. Der Einsatz von *Messenger* ist daher *Thread*-sicher. *Messenger*-Objekte haben bereits alle Funktionen implementiert, die für eine Interprozesskommunikation benötigt werden und können direkt eingesetzt werden. Für die Kommunikation zwischen zwei Anwendungen benötigen wir keine speziellen Aufrufe aus verschieden Threads. Aus den genannten Gründen werden wir daher zur Interprozesskommunikation zwischen den Java-Hüllen des *SEC-Clients* und einem (Mess-)Programm die *Messenger*-Klasse verwenden. Genauere Details zur Implementation finden sich in Kapitel 5.

Zusammenfassend haben wir folgenden Entwurf (Abbildung4.3) für den *SEC-Client* und einem beliebigen (Mess-)Programm wie das *RMF*.

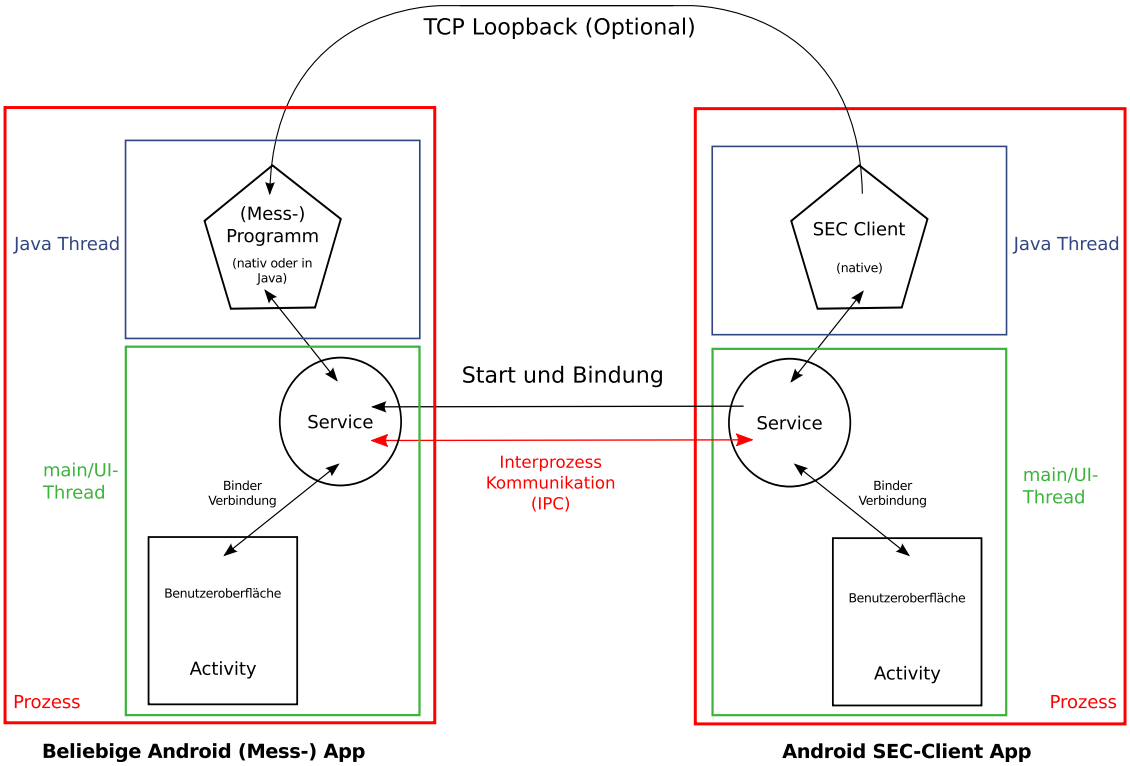

Abbildung 4.3: Design, Gegenüberstellung und Kommunikation einer beliebigen Messapplikation mit dem Android SEC-Client

Kapitel 5

Implementationsprozess und Hürden

Im vorherigen Kapitel 4 wurde ausführlich der Designentwurf der *Android SEC-Client*-Anwendung beschrieben. Die dafür entworfende *Java*-Hülle dient ebenfalls als Basis für beliebige (Mess-)Programme wie beispielsweise das *Rate Measurement Framework*, um eine nebenläufige Ausführung zum *SEC-Clienten* zu gewährleisten.

In diesem Kapitel wird der Implementationsprozess des *Android SEC-Clienten* beschrieben. Zunächst wird im Abschnitt 5.1 kurz auf die verwendete Entwicklungsumgebung und die Testgeräte eingegangen. Ebenfalls werden hier besondere Projekteigenschaften der entwickelten *Android*-Applikation beschrieben. In den darauf folgenden Abschnitten, wird konkret auf die Implementation der Anwendung eingeganen. Es werden neben der Funktionsweise, Besonderheiten und Hürden beschrieben, die im Laufe der Entwicklungsphase überwunden werden mussten. Die Entwicklung der *Java*-Hülle erfolgt auf Basis des vorgestellten Entwurfsmuster aus Kapitel 4. Die Komponenten und Funktionen der *Java-Hülle* wurden während der Implementationsphase sukzessiv in mehreren Phasen entwickelt und ausgiebig getestet. So konnten *Bugs* bei der Anwendungsinteraktion leichter gefunden und behoben werden. Bis zur Fertigstellung der Hüllen wurden zu Test- und Debuggingzwecken Funktionalitäten des *SEC-Clienten* und Interaktionen mit anderen (Mess-)Programmen simuliert. Insgesamt lässt sich der Entwicklungsprozess in drei Phasen unterteilen, die im folgenden näher erläutert werden. Zu Beginn wurde die reine *Java*-Hülle entwickelt (siehe Abschnitt 5.2).

Im nächsten Schritt wurde die *JNI*-Schnittstelle in *C/C++* implementiert und die Interak-

tion mit allgemein nativen Code getestet (Abschnitt 5.3). Nachdem die Anwendungshülle fertiggestellt war, um den eigentlichen *SEC-Client C/C++*-Code einzubetten, wurde das Laufverhalten der Hülle gestestet. In diesen Stadium mussten einige Hürden überwunden werden, um eine stabile Ausführung der erstellten Anwendungshülle zu gewährleisten (Abschnitt 5.4). Im letzten Schritt folgte die Anpassung und Integration des *SEC-Client C/C++*-Code in die *Android*-Anwendungshülle (Abschnitt 5.5). Analog dazu wurde das *RMF* ebenfalls in einer solchen *Java-Hülle* integriert und zusammen mit der *Android SEC-Client*-Anwendung getestet (Abschnitt 5.6).

5.1 Entwicklungsumgebung, Testgeräte, und Projekteigenschaften

Als Entwicklungsumgebung kam das von Google bereitgestellte *ADT*-Bundle (Build-Version: v22.2.0-822323) zum Einsatz. Dieses enthält neben der Open-Source Entwicklungsumgebung *Eclipse*, das zur *Android*-Anwendungsentwicklung notwendige *Android-SDK* und das *ADT-Plugin* (siehe Abschnitt 3.2). Als *NDK*-Distrubition wurde das *Crystax-NDK* (Revision 8) verwendet (siehe Abschnitt 3.3.

Als Testgeräte kamen die *Samsung*-Smartphone-Geräte *Galaxy S II (i9100)* und *Galaxy Nexus* zum Einsatz. Die technischen Daten zu beiden *Android*-Smartphones sind in Tabelle 7.1 aus Abschnitt 7.1 zu finden.

Android-Anwendungen besitzen analog zur *Java*-Programmierung einen *package*-Name. Da dieser vom *Android*-Betriebsystem verwendet wird um Applikationen zu identifizieren, muss dieser eindeutig sein um Konflikte mit anderen installierten Anwendungen zu vermeiden. Goolge empfiehlt als Präfix für den *Package*-Namen die *Domain* der zugehörigen Organisation in umgekehrter Reihenfolge zu verwenden. Die *Android SEC-Client*-Anwendung im Rahmen der Bachelorarbeit an der Heinrich-Heine-Universität am Lehrstuhl für Rechnernetze und Kommunikationssysteme (Fakultät Informatik) entwickelt. Daher wurden folgende *Package*-Benennungen vorgenommen: `de.hhu.cs.cn.sec` und de.hhu.cs.cn.rmf. gewählt. Bei einer Veröffentlichung der Applikation(en) im *Goolge Play Store* verlangt Google beim Programmieren des *Java*-Codes die Einhaltung der

Android Code Style Guideline. Diese wurden während der Implementation ebenfalls berücksichtigt, um bei eine möglichen offiziellen Veröffentlichung der Anwendung(en) keine Anpassungen im Quelltext vornehmen zu müssen. Während der Programmierung wurde zunächst bewusst auf den Einsatz von *Exceptions* verzichtet. Durch Programmabstürze konnten Probleme gezielt lokalisiert und behoben werden. Für *Debugging*-Zwecke wurde *LogCat* (Abschnitt 3.2) verwendet. Dieses *Android Logging*-System erlaubt Ausgaben in verschiedenen Kategorien, welche beliebig gefiltert ausgewertet werden können.
In diesem Kapitel werden die Begriffe *Funktion* und *Methode* synonym verwendet. Zudem werden diese kursiv ohne Paramater dargestellt. Wichtige Parameter werden in Klammern mitangeben.

Nachdem in diesen Abschnitt allgemeine Rahmeneigenschaften beschrieben wurden, folgt die Implementation der reinen *Java*-Hülle, die im nächsten Abschnitt erörtert wird.

5.2 Entwicklung der reinen Java-Hülle

Zunächst wurde der reine *Java*-Teil der Anwendung entwickelt (Abb. 5.1). Diese besteht aus einer *Activity* als Benutzeroberfläche und einem *Service* als Hintergrundsdienst. Beide Komponenten werden durch die Klassen *ActivitySEC* und *ServiceSEC* repräsentiert, die jeweils von der *Activity*- bzw. *Service*-Klasse erben. Der *Service* kann über die Benutzeroberfläche gestartet und beendet werden.

5.2.1 Starten und Bindung zum Service

Zum Starten der *ServiceSEC*-Klasse durch die *ActivitySEC*-Klasse wird ein *expliziter Intent* erstellt, der mit der Methode *startService()* aufgerufen wird. Zusätzlich wird für den Datenaustausch eine Bindung zwischen *Activity* und *Service* eingegangen. Dazu ruft die *Activity* die *bindService()*-Methode auf. Es ist hierbei zu beachten, dass der Aufruf dieser Methode erfolgt, nachdem der *Service* durch *startService()* gestartet wurde. Wird nur *bindService()* aufgerufen, wird zwar ebenfalls der *Service* gestartet und zusätzlich

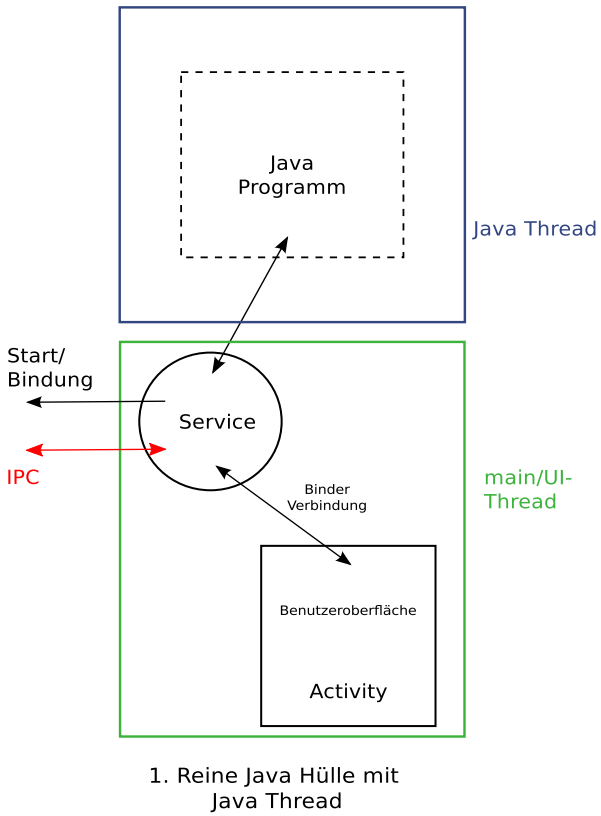

Abbildung 5.1: Entwicklung der reinen *Java*-Hülle mit *Java*-Thread

eine Verbindung aufgebaut, jedoch wird der *Service* beendet, sobald keine Bindung mehr zum Clienten besteht. Da unser *Service* jedoch unabhängig von der *Activity* und sonstigen Interaktionen ausgeführt werden soll, rufen wir zunächst *startService()* auf, gefolgt von *bindService()*.

Der folgende Codeausschnitt 5.1 zeigt, wie die *ServiceSEC*-Klasse bei einer Verbindungsanfrage ein *Binder*-Objekt generiert, welches zur Kommunikation zwischen beiden Komponenten genutzt werden kann.

Listing 5.1: Lokaler Verbindungsaufbau zwischen *Activity* und *Service*

```
1 // inner class inside ServiceSEC class, extends Binder class
2 // which implements IBinder interface
3
4 public class BinderServiceSEC extends Binder {
5
6   public ServiceSEC getService(ActivitySEC activitySEC) {
```

```
7
8         // save SEC activity reference
9         mActivitySEC = activitySEC;
10
11        // we are inside ServiceSEC, that's why we
12        // had to use ServiceSEC as prefix
13        return ServiceSEC.this;
14    }
15 }
16
17 // ServiceConnection interface callback method
18 // inside ActivitySEC Class
19
20 public void onServiceConnected(ComponentName name, IBinder service) {
21     ...
22     // cast IBinder object to our extended Binder class
23     BinderServiceSEC binderServiceSEC = (BinderServiceSEC) service;
24
25     // get ServiceSEC instance and pass ActivitySEC reference
26       mServiceSEC = binderServiceSEC.getService(ActivitySEC.this);
27 }
```

Die *bindService()*-Methode benötigt neben dem zugehörigen *Service-Intent* ein Objekt, welches das *ServiceConnection-Interface* implementiert. Dieses *Interface* beschreibt unter anderen die *Callback*-Methode*OnServiceConnected()* (Zeile 20), die als Parameter ein *Binder*-Objekt liefert. Um zu verstehen, wie ein *Binder*-Objekt zum Zugriff auf den *Service* genutzt werden kann, muss an dieser Stelle erörtert werden, was bei einer Bindung auf der *Service*-Seite geschieht. Hier wird bei einer Verbindungsanfrage die *Callback*-Methode *OnBind()* aufgerufen, die als Rückgabewert ein *Binder*-Objekt liefert, welches dem Clienten durch die Methode *OnServiceConnected()* als Paramter übergeben wird. Um den Zugriff auf unsere *Service*-Klasse zu erlauben, erstellen wir innerhalb dieser eine innere Klasse *BinderServiceSEC* (ab Zeile 4), die von der *Binder*-Klasse erbt. Zusätzlich implementieren wir die Methode *getService()* (Zeile 6), die als Rückgabewert die Instanz der von ihr umgebenden *ServiceSEC*-Klasse liefert. Gleichzeitig verlangt die Methode als Paramter eine Instanz auf die *ActivitySEC*-Klasse. Der *Service* erhält durch die Referenz Zugriff auf alle *public*-Elemente der *ActivitySEC*-Instanz. In der *ActivitySEC*-Klasse kann innerhalb der *OnServiceConnected()-Callback*-Methode

das *Binder*-Objekt zur einer Referenz der *BinderServiceSEC*-Klasse durch *Typecasting* umgewandelt werden (Zeile 23). Durch diese Referenz können wir *getService()* aufrufen, um eine Referenz auf die *ServiceSEC*-Instanz zu erhalten. Dadurch erhält auch die *ActivitySEC*-Klasse direkten Zugriff auf den *public*-Bereich der *ServiceSEC*-Klasse.

5.2.2 Java-Thread im Service

Innerhalb des *Services* wird ein *Java-Thread* erstellt. Um die Nebenläufigkeit dieses *Threads* zu evaluieren, wird periodisch nach einer frei einstellbaren Zeit Nachrichten in Form von *Android Toasts* [1] ausgegeben. Analog dazu wurde eine zweite Instanz dieser *Java*-Hülle erstellt, um die nebenläufige Programmausführung eines beliebigen (Mess-)Programmes zu simulieren. Da *Services* keine sichtbare Benutzeroberfläche besitzen, konnte mit Hilfe von *Toasts* die gleichzeitige Ausführung der beiden Anwendungen unabhängig von anderen Aktivitäten getestet werden.

Es gibt in *Android* jedoch eine Besonderheit bezüglich des Zugriffs auf die grafische Benutzeroberfläche aus einem *Thread* heraus zu beachten. Ruft man aus einem separaten *Thread*-Kontext Methoden auf, die als Ergebnis grafische Operation auf der Benutzeroberfläche durchführen, so wird die Applikation aufgrund einer Sicherheits-*Exception* beendet wird. In *Android* werden *UI*-spezifische Objekte wie *Views*, auf den sogenannten *main/UI-Thread* ausgeführt. Nur Objekte, die ebenfalls im diesem *Thread* ausgeführt werden, dürfen auf *UI*-Objekte zugreifen. Grundbausteine einer *Android*-Applikation wie *Activities* und *Services* werden im *main/UI-Thread* ausgeführt, so dass es hier beim Zugriff auf *UI*-Elemente keine Probleme gibt. Um aus einem separaten *Thread* denoch auf die *GUI* zuzugreifen (das Anzeigen von *Toasts* gehört ebenfalls dazu), wird die *Handler*-Klasse benötigt. Der dazugehörige Quellcode-Abschnitt ist in Listing 5.2 zu entnehmen. Eine *Handler*-Instanz kann sich mit einem gewünschten *Thread* verbinden, um Nachrichten oder ausführbare Objekte, sogenannte *Runnables*, in einer *Queue* abzulegen. Die *Queue* wird schließlich vom *Thread* abgearbeitet.

[1] *Toasts* sind *Views*, die lediglich einen Text als Nachricht enthalten, unabhängig davon, in welcher Applikation sich man befindet sichtbar dem Benutzer für kurze Zeit angezeigt werden.

Listing 5.2: Zugriff auf den *main/UI-Thread* mittels der *Handler*-Klasse

```
1 Handler handler = new Handler();
2
3 // pass runnable to main/UI thread
4 handler.post(new Runnable() {
5   @Override
6   public void run() {
7     Toast.makeText(getContext(), "Hello", Toast.LENGTH_SHORT).show();
8   }
9 }
```

5.2.3 IPC zwischen beiden Services

Wie im Abschnitt 4.2.7 beschrieben, soll die *Java*-Hülle des *SEC-Clienten* den *Service* innerhalb einer beliebigen (Mess-)Applikation aufrufen und zu Kommunikationszwecken eine Interprozessverbindung zu ihm aufbauen. Analog zu einer Bindung mit einem lokalen *Service* (Abschnitt 5.2.1) wird zunächst von der *ServiceSEC*-Klasse ein *expliziter Intent* erstellt. Da es sich hierbei um einen *Service* aus einer anderen Applikation handelt, muss im *Intent* der vollständig aufgelöste *Package*-Pfad des Ziel-*Services* angegeben werden. Beispielsweise ist dieser für die *RMF*-Anwendung aus dem *Package*-Namen der Applikation und dem *Package*-Struktur innerhalb der Anwendung zusammengesetzt: `de.hhu.cs.cn.rmf.service.ServiceRMF`. Des Weiteren wird aus Sicherheitsgründen innerhalb der Ziel-Anwendung ein *Intent-Filter*-Eintrag der *AndroidManifest.xml* benötigt, welches den *Service* kennzeichnet, so dass dieser von anderen Anwendungen aufgerufen werden kann.

Nachdem der *Service* in der anderen Anwendung gestartet wurde, wird durch die *bindService()* eine Bindung zum Service aufgebaut. Der folgende Quelltext-Auszug (Listing 5.3) zeigt die Unterschiede zur lokalen *Service*-Bindung, die in Abschnitt 5.2.1 vorgenommen wurde.

Listing 5.3: IPC-Binder Verbindung mittels der *Messenger*-Klasse mittels der *Handler*-Klasse

```
1 // callback method inside target service class
```

```
2  @Override
3  public IBinder onBind(Intent intent) {
4      ...
5
6      // return our messenger binder for communication
7      return mMessengerServiceIn.getBinder();
8  }
9
10 // callback method inside calling service
11 @Override
12 public void onServiceConnected(ComponentName name, IBinder service) {
13
14     // cast IBinder to Messenger object
15     mMessengerServiceOut = new Messenger(service);
16 }
```

Wie in Abbschnitt 4.2.7besprochen wurden, muss das *Binder*-Objekt, welches in der *Callback*-Methode *OnServiceConnected()* als Parameter übergeben wird, eine *AIDL*-Schnittstelle implementiert haben, um die Kommunikation über Prozessgrenzen hinweg zu ermöglichen. Wir verwenden dafür *Messenger*-Klasse (Zeile 18), deren Implementation auf *AIDL* basiert. Im Ziel-*Service* wird bei einer Verbindungsanfrage durch die *getBinder()*-Methode (Zeile 7) einer bereitgestellten *Messenger*-Instanz ein Objekt generiert, welches das *IBinder*-Interface implementiert. Die *ServiceSEC* kann dieses Objekt zu einem *Messenger*-Objekt umwandeln (Zeile 15).

Zum Versenden von Nachrichten wird mit Hilfe der *send*-Methode der *Messenger* Instanz ein *Message*-Objekt an den Ziel-*Service* geschickt. Der Quelltext-Ausschnitt in Listing 5.4 zeigt, wie ankommende Nachrichten im Ziel-*Service* abgearbeitet werden. Zum Empfangen von Nachrichten werden *Messenger*-Instanzen, deren Referenzen bei einer Bindung dem Clienten übergeben wurden, mit einem *Handler*-Objekt (Abschnitt 5.2.2) initialisiert (Zeile 2). Der *Handler* arbeitet eintreffende Nachrichten in einer *FIFO-Queue* ab und ruft dazu die *Callback*-Methode *handleMessage()* (Zeile 8) auf. Als Parameter erhält man ein *Message*-Objekt, das von Absender versandt wurde. Innerhalb des *Message*-Objektes kann der Absender ebenfalls eine Referenz auf ein *Messenger*-Objekt ablegen (Zeile 11), die der Empfänger nutzen kann, um selbst Nachrichten an den Absender zu verschicken.

Listing 5.4: Abarbeitung der Nachrichten-*Queue* durch die *Handler*-Klasse

```
1  // initialize Messenger with a Handler
2  Messenger mMessengerIn = new Messenger(new IncomingHandler());
3
4  // extended Handler class for our messenger
5  class IncomingHandler extends Handler {
6
7    @Override
8    public void handleMessage(Message msg) {
9        ...
10       // extract bundle and reply messenger from message object
11       mMessengerOut = msg.replyTo;
12       ...
13       // answer sender
14       mMessengerOut.send(msg);
15   }
16 }
```

Verwendet man *Messenger* zur Interprozesskomunikation, so müssen Daten durch das *Message*-Objekt versenden auf eine spezielle Art verpackt werden. Wie in Abschnitt 4.2.7 erläutert wurde, müssen Daten, die über Prozessgrenzen transferiert werden, in primitive Einzelteil zerlegbar sein. Dadurch können sie vom Betriebssystem richtig interpretiert und im Speicherbereich des Zielprozesses korrekt abgelegt werden . Dazu müssen Daten, die verschickt werden sollen, das sogenannte *Parcelable*-Interface implementieren. Das *Android SKD* bietet dazu die *Bundle*-Klasse, welches die *Parcelable*-Schnittstelle implementiert und verschiedene Datentypen als <*String key, Datentyp value*>-Mapping speichern kann. Ein *Bundle* Objekt kann daher problemlos zum Datenaustausch zwischen zwei Prozessen genutzt werden. Hängt man Daten an ein *Message*-Objekt an, die nicht die geforderte Schnittstelle implementieren, so führt der Versand der Nachricht bei einer Interprozesskomunikation zur einer *Remote-Exception*.

5.2.4 Hilfsklassen

Viele Funktionalitäten des *Android SDK* wie das Anzeigen einer Benachrichtung (*Notification*) oder der Zugriff auf den externen Speicher, erfordern häufig mehrere Programm-

befehle. Oft müssen zunächst Referenzen zu diversen *Android Managern* beschafft werden, Objekte eines bestimmten Typs generiert werden und/oder Resourcen ausgelesen werden. Solche Funktionalitäten, die häufig im Programm verwendet werden, blähen daher den Programmcode auf und erschweren die Fehlersuche, da häufig bestimmte Referenzen wieder freigegeben werden müssen und eine Nichtbeachtung an anderer Stelle im Programm zu Kollisionen mit anderen Befehlen führen kann.

Um dieses Probleme zu vermeiden, wurden einige Hilfsklassen geschrieben. Abbildung 5.2 zeigt den Kommunikationsfluss zwischen den Klassen unsere *Android*-Anwendung. Die Pfeile zeigen dabei die Zugriffsrichtung einer Klasse auf eine andere.

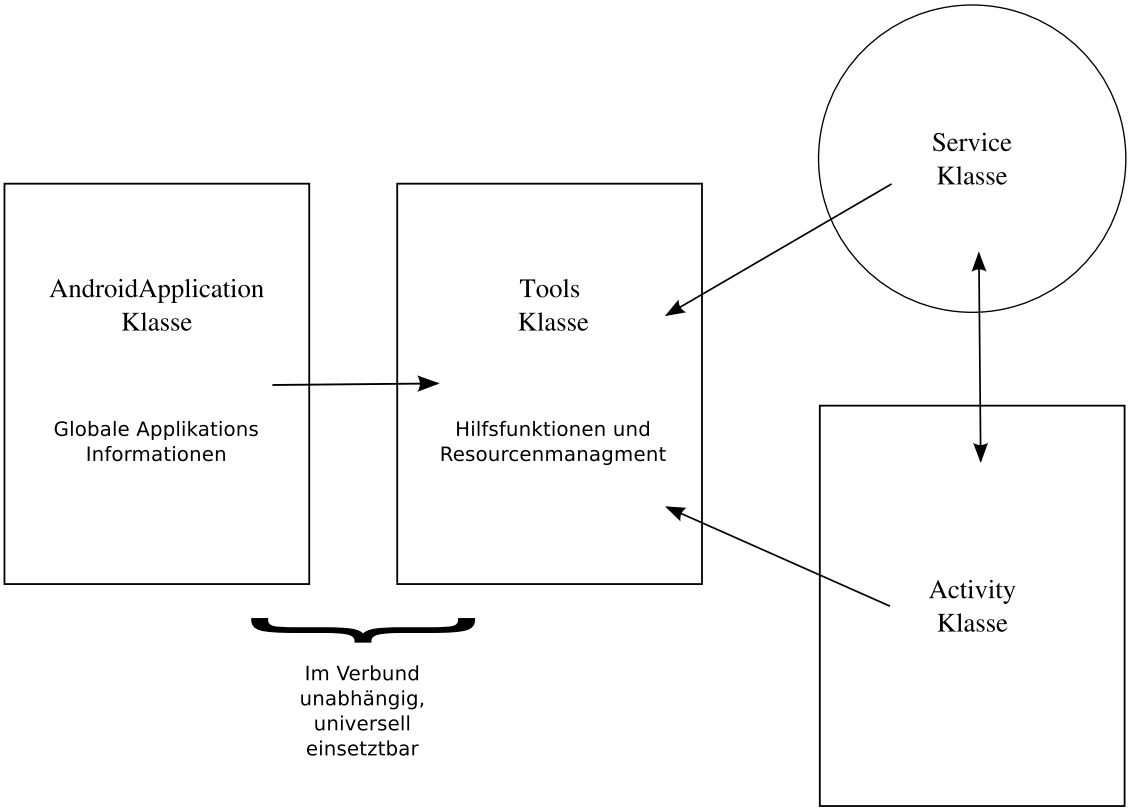

Abbildung 5.2: Interaktion mit den Hilfsklassen *Tools* und *AndroidApplication*

Einer dieser Hilfsklasse ist die statische *Java*-Klasse *Tools*. Diese stellt häufig genutze Funktionen zur Verfügung und kümmert sich intern um die dazu notwendigen Operationen. Außerdem verwaltet sie zentralisiert alle notwendigen Resourcen. Der Designfocus bei der Implementierung lag dabei auf Unabhängigkeit gegenüber Basiskompo-

nenten wie *Activities*, *Services* oder projektspezifische Klassen. Dazu musste zunächst ein grundlegendes Problem gelöst werden. Viele Funktionen der *Android API* benötigen den sogenannten *Application-Context*. Anhand des *Context* kann das Betriebssystem eine Anwendung identifizieren und bestimmte Funktionen für diese Applikation durchführen. In Klassen, welche die Basiskomponenten einer *Android*-Anwendung repräsentieren wie *Activities* oder *Services*, kann der *Context* problemlos durch *getApplicationContext()* ermittelt werden. In anderen Klassen steht diese *Methode* nicht zur Verfügung.

Da die *Tools*-Klasse unabhängig von *Activity*- oder *Service*-Klassen bleiben soll, wurde eine weitere Hilfsklasse *AndroidApplication* erstellt, die ebenfalls unabhängig argiert, und die *Tools*-Klasse mit applikationspezifischen Daten wie den aktuellen *Context* versorgt. Die *AndroidApplication*-Klasse erbt von der *Application*-Klasse, die Methoden zur Ermittlung des *Context* und andere allgemeine Informationen zur Verfügung stellt. Objekte dieses Klassentypes können in der *AndroidManifest*-Datei eingetragen werden. Diese werden von der Laufzeitumgebung zum Programmstart geladen und stehen global über die gesamte Lebensdauer einer Anwendung zur Verfügung. Die *AndroidApplication*- und *Tools*-Klasse sind als Verbund zu betrachten, die unabhängig von anderen Klassen in einem Projekt funktioneren und ohne Anpassung in anderen *Android*-Anwendungen verwendet werden können. Da diese Hilfsklassen global und statisch zur Verfügung stehen, können diese von allen Komponenten einer Applikation verwendet werden. Die Auslagerung und Zentralisierung bestimmter Funktionalitäten hat zudem den Vorteil, dass Resourcenbelegungen transparent verwaltet werden können und es zu keinen Kollisionen kommt. Beispiele dafür und besondere Funktionen dieser Hilfsklassen werden im Laufe der Arbeit vorgestellt.

5.3 Interaktion der Java-Hülle mit dem C/C++ Code

Nachdem im letzten Abschnitt Besonderheiten der Implementation der reinen *Java*-Hülle beschrieben wurden, widmet sich dieser Abschnitt der Implementation der *JNI*-Schnittstelle in die Anwendungshülle. Diese ermöglicht die Interaktion von *Java*-Code mit nativen *C/C++*-Code. Abbildung 5.3 zeigt dabei den modularen Aufbau der Anwendung.

Aus der *Java*-Seite können *C/C++*-Funktionen aufgerufen werden um den *SEC-Clienten* aufzurufen und zu kontrollieren (Abschnitt 5.5). Gleichzeitig soll es möglich sein, *Java*-Methoden vom *SEC-Clienten* aufzurufen um *Android*-spezifische Funktionen wie das Starten einer *Android*-(Mess-)Anwendung durchzuführen. In unserer Anwendung erstellen wir die Schnittstelle zwischen beiden Programmiersprachen in der *C++*-Datei *jniglue.cpp*. Diese dient als Schnittstelle für Funktionsaufrufe aus der *Java*-Seite und delegiert diese weiter an den *SEC-Clienten*. Gleichzeitig stellt die *jniglue.h* dem *SEC-Clienten* Funktionen zum Durchführen von Operationen auf der *Java*-Seite bereit. Alle notwendigen *JNI*-Operationen werden in der *jniglue.cpp*-Datei durchgeführt. Eine Vermischung von *JNI*-spezifischen Code im *SEC-Clienten* wird vermieden. Zudem erhöht die Schnittstelle imsgesamt die Transparenz des Anwendungscodes.

Des Weiteren wurde in diesem Entwicklungsabschnitt analog zum *Java*-Thread aus Abschnitt 5.3.2 auf der nativen Seite ein *POSIX*-Thread erstellt, um die Nebenläufigkeit von nativen *Threads* im Kombination mit einem *Service*-Hintergrundsdienst zu testen.

5.3.1 Aufruf von C/C++ Funktionen aus Java

Zum Aufruf einer *C/C++* Funktion aus *Java* heraus muss diese Funktion zunächst deklariert werden. Im Folgenden werden native Funktionen, die aus *Java* aufgerufen werden können, stets mit einem kleinen *n* als Präfix gekennzeichnet. Als einfaches Beispiel nehmen wir die die Funktion *nNativeIntialize()*. Diese soll von der *Service*-Klasse aufgerufen werden, um die native *C/C++*-Schnittstelle (*jniglue.cpp*) zu initialisieren. Um diese dem *Java*-Compiler als native Funktion zu kennzeichnen, wird das Schlüsselwort *native* verwendet. Insgesamt sieht die Deklaration für die *nNativeIntialize()*-Funktion in *Java* folgendermaßen aus: *private native void nNativeInitialize();*

Im nächsten Schritt wird eine *C++ Header*-Datei erstellt, welche die passende Funktionsdeklaration enthält. Das *Java Development Kit (JDK)* beinhaltet das Tool *javah*, welches die nativen Funktionsdeklarationen aus einer *Java*-Klasse liest und eine entsprechende *C/C++*-Headerdatei generiert (siehe Listing 5.5). Bei der Verwendung von *Android Java*-Klassen wie der *Service*-Klasse mit *javah* ist zu beachten, dass beim Aufruf von *javah* als Parameter die verwendetete *Android*-Bibliothek als *Bootstrap Classpath*

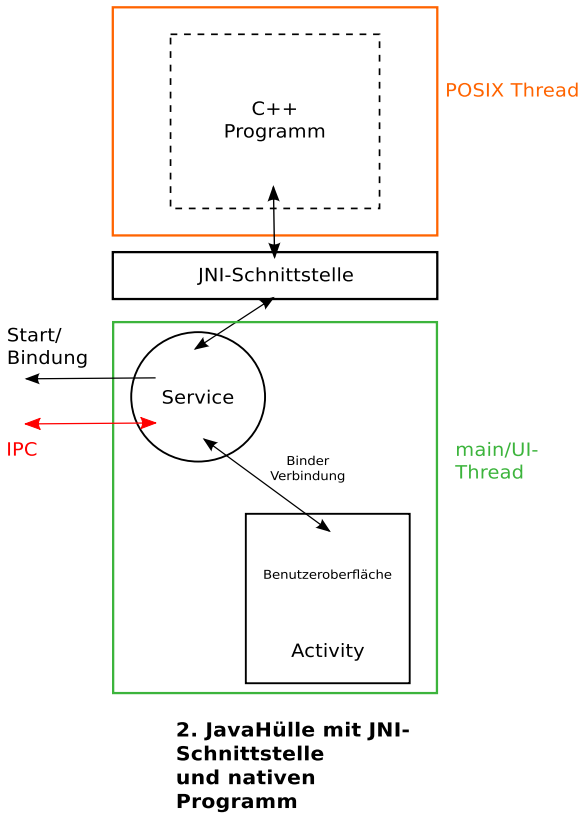

Abbildung 5.3: Entwicklung der *JNI*-Schnittstelle mit einem *POSIX*-Thread

angeben wird (*javah .... -bootclasspath /.../sdk/platforms/android-APIlevel/android.jar*). Ansonsten bricht *javah* die Erstellung der *Header*-Datei mit der Fehlermeldung ab, dass die Java-Klasse nicht gefunden werden konnte. Die Funktionsdeklarationen in der generierten *Header*-Datei haben folgende Form, die von der normalen C/C++-Syntax abweicht:

JNIEXPORT <returnType>JNICALL Java_<PackageName>_<Klassenname>_ <Methodenname>(JNIEnv , jobject, <parameters>...).*

Auffällig sind dabei die Schlüsselwörter *JNIEXPORT* und *JNICALL*, welche diese als native Funktion kennzeichnen, die durch *JNI* aufgerufen werden kann. Im Funktionsnamen folgt nach dem Wort *Java* der vollständige *Package*-Pfad der nativen Methode, die in der *Java-Klasse* deklariert wurde. Dabei werden die üblicherweise verwendeten Punkte durch Unterstriche ersetzt. Als erster Parameter wird stets ein Zeiger(*JNIEnv**)

auf die *JNI*-Funktionstabelle geliefert. Als zweiter Parameter folgt eine Referenz auf das *Java*-Objekt (*jobject*), welches den nativen Funktionsaufruf tätigt. Danach folgt die eigentliche Parameterliste der nativen Funktion, die in *Java* deklariert wurde. Dabei besitzt jeder primitive *Java*-Datentyp eine entsprechende *JNI*-Datentyp-Repräsentation (Tabelle 5.1).

Java ist als plattformunabhängige Sprache implementiert, die sicherstellt das primitive Datentypen unabhängig von der Ausführungsumgebung stets die gleiche Bytegröße besitzen. In *C/C++* hängt die Länge der primitiven Datentyp jedoch von der Plattform ab. Um sicherzustellen, dass primitive *Java*-Datentyp auf *C/C++*-Datentypen passender Bytegröße abgebildet werden, stehen passende *JNI*-Datentypen bereit. Diese sind *typedefs*, die zu der Definition des *C99*-Standarts verweisen, die garantieren, dass beispielsweise der int64_t-Datentyp eine Größe von 64bits besitzen und so als Container für den *Java long*-Datentyp verwendet werden kann. Da *C/C++* keine *Java*-Klassen kennt, wird für deren Repräsentation der *JNI*-Datentyp *jobject* verwendet. Die häufig verwendete *Java*-Klasse *String* wird durch den *JNI*-Datentyp *jstring* dargestellt, wobei dies letzendlich ein *typedef* auf *jobject* ist. Der *jobject*-Datentyp kann nicht direkt benutzt werden, um auf *Java*-Objekte zuzugreifen. Dafür sind *JNI*-Funktionen notwendig, die einen Zugriff auf ein *Java*-Objekte über die *Dalvik VM* durchführen (siehe Abschnitt 5.3.2).

Java Typ	JNI Typ	C Typ	Stdint C Type
boolean	Jboolean	unsigned char	uint8_t
byte	Jbyte	signed char	int8_t
char	Jchar	unsigned short	uint16_t
double	Jdouble	double	double
float	jfloat	float	float
int	jint	Int	int32_t
long	jlong	long long	int64_t
short	jshort	Short	int16_t

Tabelle 5.1: *Java*-Datentypen und die dazugehörige *JNI*-Datentypen

In unserem konkreten Beispiel sieht der Aufbau der generierten *Header*-Datei folgendermaßen aus (Listing5.5):

Listing 5.5: Durch *Javah* generiete *Header*-Datei

```
1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class de_hhu_cs_cn_sec_service_ServiceSEC */
4
5  #ifdef __cplusplus
6  extern "C" {
7  #endif
8
9  /*
10  * Class:     de_hhu_cs_cn_sec_service_ServiceSEC
11  * Method:    nNativeInitialize
12  * Signature: ()V
13  */
14 JNIEXPORT void JNICALL
       Java_de_hhu_cs_cn_sec_service_ServiceSEC_nNativeInitialize
15   (JNIEnv *, jobject);
16
17 #ifdef __cplusplus
18 }
19 #endif
```

Wichtig ist hierbei, die Funktionsdeklaration für einen *C++*-Compiler (*ifdef*) als *extern C* zu markieren (Zeile 6). Im Gegensatz zur Sprache C, gibt es in *C++* das Konzept von überladenen Funktionen. Um überladene Funktionen zu unterscheiden, baut der *C++*-Compiler im Symbolnamen der Funktion neben dem Funktionsnamen auch die Parameter ein[2]. Damit diese Funktionen auch von *C*-Code verwendet werden kann, werden die Funktionsdeklarationen für den *C++*-Compiler als *extern C* markiert, so dass dieser lediglich den Funktionsnamen in den Symbolnamen verwendet.

Nachdem die *Header*-Datei generiert wurde, wird diese in der *jniglue.cpp* inkludiert und die beschriebene Funktion definiert. Wir nutzen die *nNativeInitialize*-Funktion um *JNI*-Operationen durchzuführen, die als Ergebnis Referenzen auf *Java*-Methoden liefern, die wir aus *C/C++* aufzurufen wollen. Die dazugehörigen *JNI*-Operationen und Besonderheiten werden im nächsten Abschnitt detailierter beschrieben.

[2]Dieses Verfahren wird auch als *name mangeling* bezeichnet.

5.3.2 Aufruf von Java-Methoden aus C/C++

Um *Java*-Methodenaufrufe aus *C/C++* aufzurufen, sind zunächst *JNI*-Operationen notwendig, die in diesen Abschnitt vorgestellt werden. Zunächst benötigt man eine Referenz auf die *Java*-Klasse, in der wir eine Methode aufrufen wollen. Über den *env**-Zeiger, der auf die *JNI*-Funktionstabelle verweist, rufen wir die Funktion *env->GetObjectClass(jobject)* auf. Dabei ist *jobject* eine Referenz auf die *Java*-Klasseninstanz, in welcher der Methodenaufruf erfolgen soll. Den Rückgabewert (*JNI*-Datentyp *jclass*) verwenden wir im nächsten Schritt, um die Methoden ID (*jmethodID*) der gewünschten *Java*-Methode zu ermitteln. Diese erhalten wir als Rückgabewert der *JNI*-Funktion *env->GetMethodID(jclass, "Methodenname", "Signature"*. Dabei wird der Methodenname als gewöhnliche *C/C++*-Zeichenkette angegeben. Da auch in *Java* Methoden überladen werden können, muss die Signature mit angegeben werden. Der allgemeine Syntax dazu lautet *"(Paramaterliste)Rückgabewert"*, wobei zwischen den einzelnen Parametern kein Leerzeichen vorhanden ist. Die Kodierung der Datentypen ist in Tabelle 5.2 (nicht vollständig) einzusehen. *Java*-Objekte beginnen mit einem "L", gefolgt vom *Package*-Pfad und einem abschliessenden ";" Beispielsweise sieht der *JNI*-Funktionsaufruf zur Ermittlung der Methoden ID für die *Java*-Methode *boolean testMethod(String, int, double[])* wie gefolgt aus: *env->GetMethodID(jclass, "testMethod", "(Ljava/lang/String;I[D)Z").*

Java Type	Nativer Typ	Native Array Typ	Typ Kodierung	Array Typ Kodierung
boolean	jboolean	jbooleanArray	Z	[Z
byte	jbyte	jbooleanArray	Z	[Z
char	jchar	jcharArray	C	[C
double	jdouble	jdoubleArray	D	[D
float	jfloat	jfloatArray	F	[F
int	jint	jintArray	I	[I
long	jlong	jlongArray	J	[J
short	jshort	jshortArray	S	[S
Object	jobject	jobjectArray	L	[L
String	jstring	N/A	L	[L
Class	jclass	N/A	L	[L
Throwable	jthrowable	N/A	L	[L
void	void	N/A	V	N/A

Tabelle 5.2: *Java*-Datentypen und die dazugehörige *JNI*-Signaturen

Mit Hilfe der ermittelten *Java*-Methoden ID können wir einen Aufruf auf der *Java*-Seite durchführen. Dabei gibt es für alle in *JNI* möglichen Rückgabewerte eine entsprechende *JNI*-Funktion. Die allgemeine Form dazu lautet *env-CallReturntypeMethod(jobject, jmethodID, va_arg)*.

Analog zur vorgestellten Vorgehensweise existieren für statische *Java*-Klassen und Methoden entsprechende *JNI*-Funktionalitäten. Weitere *JNI*-Funktionen werden im Verlauf dieser Arbeit vorgestellt.

5.3.3 Besonderheiten in JNI

In *JNI* gibt es einige Besonderheiten zu beachten, deren Nichteinhaltung zu Programmabstürzen führt. So liegen Objekt-Referenzen, die bei einem nativen Funktionsausruf aus *Java* als Parameter mit übergeben werden, als lokale Referenzen vor. Sobald die Funktion verlassen wird, löscht der *Garbage Collector* der *Java*-Laufzeitumgebung das Objekt. Um diese Objekt-Referenzen im nativen Programm für eine spätere Verwendung zu speichern, müssen diese durch die *JNI*-Funktion *env->GetGlobalRef()* in eine globale Referenz umgewandelt werden. Diese wird von der *Dalvik VM* nicht gelöscht. Globale Referenzen müssen am Ende eines natives Programmes wieder freigegeben (*env->DeleteGlobalRef*) werden, damit diese von dem *Garbage Collector* eingesammelt und gelöscht werden können.

Des Weiteren ist der *env**-Pointer lokal für einen *Thread* generiert und kann nicht in anderen *Threads* verwendet werden. Insbesondere sind nativ erstellte *POSIX Threads* der *Dalvik VM* nicht bekannt. Um einen *Thread* an die *Dalvik VM* anzukoppeln, muss die *JNI*-Funktion *attachCurrentThread()* aufgerufen werden, die zudem eine passende *env**-Referenz liefert, die nur innerhalb dieses *Thread* für emphJNI-Funktionsausrufe verwendet werden kann. Wird ein Thread beendet, so muss dieser durch die *detachCurrentThread()*-Funktion wieder von der *Java VM* entkoppelt werden. Beide erwähnten Funktionen werden durch die *JavaVM**-Referenz aufgerufen. Dieser Verweis auf die ausführende *Java VM* kann ohne Bedenken gespeichert und *Thread*-unabhängig verwendet werden.

Zudem ist bei der Kompilierung von nativem Code, der *JNI*-Funktionen verwendet, darauf zu achten, ob ein *C* oder *C++*-Compiler eingesetzt wird. In *C* exitieren keine Objekte, so dass *env* ein *structure* ist, welcher Funktionszeiger enthält. Bei der Verwendung ist darauf zu achten, den Pointer zu dereferenzieren ((*env)->).

5.3.4 Kompilierung und Einbindung von C/C++ Code in Android

Die Kompilierung des nativen Code zu einer dynamischen Bibliothek erfolgt mit Hilfe des *NDK-Buildsystems*. Dazu wird ein *Android*-spezifisches *Makefile* (*Android.mk*) benötigt. Der allgemeine Aufbau der *Android.mk*-Datei sieht folgendermaßen aus (Listing 5.6):

Listing 5.6: Allgemeiner Aufbau der *Android.mk*

```
1 LOCAL_PATH := $(call my-dir)
2
3 include $(CLEAR_VARS)
4
5 # here we give our module name and source file(s)
6 LOCAL_MODULE    := sharedlibraryname
7 LOCAL_SRC_FILES := sourcefile_1.cpp sourcefile_2.cpp sourcefile_3.cpp
8
9 include $(BUILD_SHARED_LIBRARY)
```

In der ersten Zeile wird durch Aufruf des *my-dir*-Makros das Arbeitsverzeichnis ermittelt, in dem sich die *Makefile* befindet. Der nächste Befehl fügt ein weiteres *Makefile* (CLEAR_VARS) ein, das gesetzte *NDK*-spezifische Variablen löscht. Dies ist notwendig, da *Makefiles* verschachtelt bzw. verkettet ausgeführt werden können und vorherige Einstellungen nicht unbedingt bei der Kompilierung eines Moduls erwünscht sind. Mit *LOCAL_MODULE* wird der Name der dynamischen Bibliothek angegeben, wobei im Ergebnis gemäß *Linux*-Konvention für eine *shared library* der Dateiname den Präfix **lib** und das Postfix **.so** erhält (Bsp. **lib**modulename**.so**). In der nächsten Zeile werden, durch Leerzeichen getrennt, die einzelnen *C/C++*- Quelldateien angegeben, die in die dynamische Bibliothek eingebunden werden sollen. Das eigentliche Kompilieren und Linken erfolgt dabei durch die BUILD_SHARED_LIBRARY-*Makefile*, welches als letztes

eingebunden wird (Zeile 9). Es ist wichtig, dass die nativen Quelldateien im *Android*-Projektordner im Unterverzeichnis *jni* vorliegen. Nur so erkennt das *NDK-Buildsystem* die Quelldateien. Fertige Bibliotheken werden in das Projektunterverzeichnis *libs* abgelegt.

Im *Java*-Quellcode (siehe Listing 5.7) wird die dynamische Bibliothek durch die statische Methode *loadLibrary()* der Klasse *System* geladen. Als Parameter wird dabei der Name der Bibliothek ohne den **lib**-Präfix und dem **.so** angegeben. Überlicherweise geschieht dies in einem statischen Programmblock, so dass die Bibliothek vor dem ersten Aufruf einer nativen Funktion von *Classloader* geladen wird.

Listing 5.7: Laden einer dynamischen Bibliothek innerhalb eines statischen *Java*-Block

```
1 // native function declarations
2 private native void nFunction_1();
3 private native void nFunction_2();
4 ...
5 // statically load shared library
6 static {
7   System.loadLibrary("sharedlibraryname");
8 }
```

5.3.5 Implementierung JNI-Schnittstelle

In den vorherigen Abschnitten wurde ausführlich erläutert, wie die Kommunikation zwischen *Java* und *C/C++* Code durch *JNI* erfolgt. Während auf der *Java*-Seite native Funktionsaufrufe wie normale Methodenaufrufe im Programmcode durchgeführt werden, erweist sich der Aufruf von *Java*-Methoden aus *C/C++* als komplizierter Vorgang, bei dem viele Besonderheiten berücksichtigt werden müssen (Abschnitt 5.3.3). Da die Kommunikation zwischen dem *C/C++* Code des *SEC-Clienten* und der *Java*-Hülle zwangsläufig über *JNI*-Operationen erfolgen muss, wurde das *C++*-Modul *jniglue* erstellt, welches als Schnittstelle zwischen beiden Anwendungsbereichen dient. Dabei soll die *jniglue.cpp*-

Datei native Funktionsaufrufe aus *Java* regulieren, um den *SEC*-Clienten zu steuern. Gleichzeitig bietet die *jniglue.h*-Headerdatei dem *SEC-Clienten* Funktionen an, um mit der *Java*-Seite zu kommunizieren. Dabei werden alle notwendigen *JNI*-Operationen im *jniglue*-Modul durchgeführt. Dadurch wird eine Vermischung von *JNI*-Code mit dem *SEC-Client*-Code vermieden. Des Weiteren wird die Anwendungsentwicklung transparenter, da die teilweise komplexen *JNI*-Operationen abgekapselt im *jniglue*-Modul durchgeführt werden und sowohl der *Java*-Hülle als auch dem *C/C++*-Code des *SEC-Clienten* eine einfach zu handhabende Schnittstelle geboten wird. Der modulare Aufbau der Anwendung mit integrierter *JNI*-Schnittstelle ist in Abbildung 5.4 zu finden. Analog zu Abschnitt 5.2.4 zeigen die Pfeile die Zugriffsrichtungen der einzelnen Module:

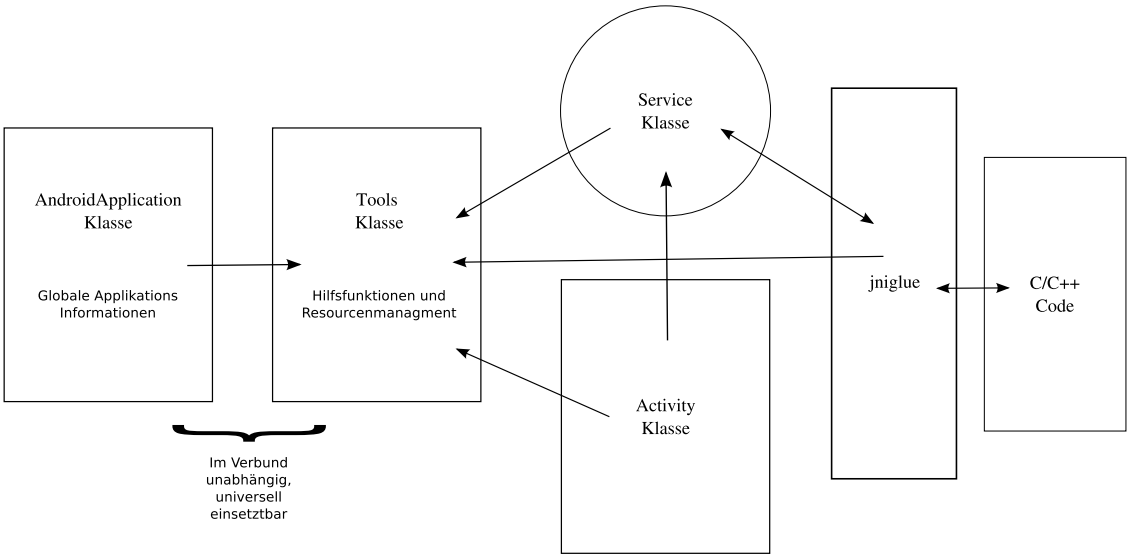

Abbildung 5.4: Interaktion zwischen den Modulen der *Android*-Anwendung

Zunächt wird beim Starten des *Services* die dynamische Bibliothek geladen, die den *C/C++*-Code der Anwendung enthält. In dieser befindet sich auch das *jniglue*-Modul, in welcher automatisch durch die *Dalvik VM* die *Callback*-Funktion *JNI_OnLoad()* aufgerufen wird. Diese übergibt als Parameter einen Zeiger(*JavaVM**) auf die *Dalvik VM*. Diese Referenz speichern wir ab, so dass diese global im *jniglue*-Modul verfügbar ist. Wie bereits in Abschnit 5.3.3 erwähnt, kann der *JNIenv*-Zeiger nicht immer *gecached* werden. In Funktionen, die aus *Java* durch die *Dalvik VM* aufgerufen werden, wird der passende *JNIenv*-Verweis als Parameter mitgeliefert, so dass dies kein Problem darstellt. Jedoch sind Aufrufe aus dem nativen Code des *SEC*-Clienten gewöhnliche *C/C++*-

Funktionsaufrufe, in denen der *JNIenv*-Zeiger nicht zur Verfügung steht. Daher wurde in der *jniglue.cpp*-Datei die Hilfsfunktion *getJNIenv()* implementiert, die mit Hilfe des gecachten *JavaVM*-Zeiger einen passende *JNIenv*-Referenz generiert.

Des Weiteren enthält das *jniglue*-Modul die Funktion *nNativeInitialize()*, die vor Benutzung der *JNI*-Schnittstelle aufgerufen werden muss. In dieser werden Referenzen zu benötigten *Java*-Methoden ermittelt und in globale Referenzen umgewandelt, so dass auch diese Verweise für den späteren Gebrauch gespeichert werden können. Dabei werden ebenfalls Verweise zu Methodenaufrufen aus der *Tool*-Klasse gespeichert. Der Vorteil beim Zugriff auf die *Tools*-Klasse ist, dass einerseits *Android*-Funktionalitäten durch einen einzigen Methodenaufruf anbietet, wofür normalerweise mehrere Befehle benötigt werden. Es werden dadurch *JNI*-Operationen im *jniglue*-Modul gespart, wodurch die Komplexität des Codes abnimmt. Gleichzeitig profitiert die gesammte Anwendung davon, da die *Tools*- Klasse bestimmte Resourcen zentral verwaltet und somit keine Kollisionen entstehen können.
Das Cachen und Bereitstellen von Funktions- und Klassenreferenzen hat den Vorteil dass, die *JNI*-Operationen zur Ermittlung dieser, nicht bei jedem *Java*-Methodenaufruf im C/C++ Code erneut ausgeführt werden müssen. Zudem wird die Performance der Gesamtanwendung gesteigert, da *JNI*-Operationen teuer sind.

5.3.6 IPC zwischen nativen POSIX Threads

Der *SEC-Client* und das *Rate Measurement Framework* arbeiten intern mit nativen *POSIX Threads*. Zur Überprüfung der Nebenläufigkkeit von nativen *Threads*, die innerhalb von einem *Service* gestartet werden, wurde ein *POSIX-Thread* erstellt welcher durch die *jniglue*-Schnittstelle auf die *Java*-Seite zugreift um analog zum *Java*-Thread im Abschnitt 5.2 periodisch eine *Toast*-Nachricht auszugeben. Hierfür musste der *POSIX Thread* an die *Dalvik VM* angekoppelt werden, um durch *JNI*-Operationen auf die *Tools*-Klasse zugreifen zu können. Des Weiteren musste innerhalb der *Tools*-Klasse die Methode, die *Toasts* ausgibt, angepasst werden. Hier musste der *Handler* (Abschnitt 5.2.2), welcher *UI*-Operationen an die *Queue* des *main/UI-Thread* schickt, explizit mit dem *Looper*[3]

[3]Ein *Looper* in Android ist eine Art Nachrichten-Schleife innerhalb eines *Threads*, der ankommende Anfragen in einer *Queue* abarbeitet.

des *main/UI*-Threads initialisiert werden. Diese Initialisierung muss bei Aufrufen aus *Threads* erfolgen, die nicht durch *Java* erstellt wurden.

Bevor die Anwendungshülle genutzt werden konnte, um den *SEC-Clienten* zu integrieren (bzw. ein beliebiges (Mess-)Programm), wurde die entwickelte Hülle getestet.

5.4 Stabilisierung der Java-Hülle

Zum Testen der Laufeigenschaften der entwickelten *Android*-Anwendung wurden zwei Instanzen der Anwendungshülle gestartet, wobei die *Services* beider Applikationen eine Interprozessbrücke aufbauen. Zur Evaluierung der Nebenläufigkeit beider Anwendungen zeigen die *POSIX*-Thread innerhalb der *Services* periodisch eine *Toast*-Nachricht an. Zudem schickt ein *POSIX* Thread über die Interprozessbrücke dem anderen *POSIX*-Thread eine Nachricht zu, auf die dieser antwortet. Zum Protokollieren der Ereignisse wurde die *C++*-Klasse *TransTimeLogger* geschrieben, die unter anderem Uhrzeit und Transferzeit der Nachrichten mit einer Genauigkeit im Nanosekundenbereich protokolliert.

Beide Anwendungen zeigten zunächst das gewünschte Verhalten. Jedoch hörte eine der beiden Anwendungen scheinbar willkürlich nach einer unbestimmten Zeit auf, die Transferdaten empfangene Nachrichten in der Log-Datei zu protokollieren. Bei einigen Testläufen waren sogar beide Anwendungen davon betroffen. Zudem brach bei manchen Testläufen eine Seite der Interprozessbrücke zusammen, sodass nur ein *Service* dem anderen *Service* Nachrichten zuschicken konnte. Des Weiteren wurde nach einer unbestimmten Zeit einer der beiden *Services* ohne ersichtlichen Grund. Nach einiger Zeit schließlich auch der andere. Bei anderen Testläufen wiederum wurde einer der *Services* mit der Fehlermeldung beendet, dass die Anwendung abgestürzt sei. Erschwerend kam hinzu, dass ein bestimmter Fehlertyp nicht reproduzierbar war, um diesen genauer zu untersuchen, so dass die Anwendungen scheinbar bei jedem Durchlauf ein anderes Verhalten aufwiesen. Dieses Verhalten zeigte sich auf beiden Testgeräten (siehe Abschnitt ref*secMessungen:Testgeräte*). Am Ende stellte sich heraus, dass es sich dabei um mehrere Probleme und Besonderheiten *Androids* handelte, die dynamisch miteinander wirkten und dadurch bei jedem Durchlauf der Anwendungen ein anderes Lauf-

zeitverhalten erzeugten. Die Fehleranalyse und die Behebung der Probleme werden in chronologischer Reihenfolge in den folgenden Abschnitten geschildert.

5.4.1 Die JNI-local reference table

Bei der Fehlersuche brachte die Verwendung von *LogCat* zunächst keinen Erfolg, da entweder keine Auffälligkeiten angezeigt wurden, die Verbindung zum *Adb*-Server abbrach, oder bei einem Programmabsturz hin und wieder die Fehlermeldung *Signal 11, SSegmentation Fault"* anzeigt wurde. Dieser Fehler, der bei einer Speicherzugriffsverletzung auftritt, war jedoch nicht genauer zu lokalisieren, da er nur selten auftrat. Insgesamt wurden die Debugausgaben und Programmausgaben in Form von *Toasts* erhöht, was bei einem Durchlauf den ersten Fehler offenlegte. Eine der Anwendungen stürzte ab, wobei folgende Fehlermeldung (Listing 5.8) über *LogCat* angezeigt wurde:

Listing 5.8: *JNI-local reference table*-overflow Fehlermeldung durch *LogCat*

```
1 E/dalvikvm(6001):JNI ERROR (app bug): local reference table overflow (
      max=512)
2 W/dalvikvm(6001):JNI local reference table (0x40f4f4d0) dump:
3 W/dalvikvm(6001):Last 10 entries (of 512):
4  ...
5 W/dalvikvm(6001):1 of java.lang.Class
6 W/dalvikvm(6001):511 of java.lang.String (511 unique instances)
7 E/dalvikvm(6001):Failed adding to JNI local ref table (has 512 entries
      )
```

Werden durch *JNI*-Operationen *Java*-Objekte initialisiert, so wird eine lokale Referenz für dieses Objekt erstellt und intern in einer Tabelle gespeichert. Diese kann jedoch maximal 512 Referenzen verwalten. Daher ist es wichtig, eine lokale Referenz wieder freizugeben, sobald diese nicht mehr benötigt wird. In unserer Anwendung ruft zum Beispiel das *jniglue*-Modul zum Anzeigen von *Toasts* die Hilfsmethode *showShortToast()* in der *Tools*-Klasse auf. Diese benötigt als Paramater einen *Java String*, welcher als *Toast* angezeigt wird. In der *jniglue.cpp* wird daher durch die *JNI*-Funktion *env->NewStringUTF("Zeichenkette")* ein *Java String* erzeugt. Ebenfalls wurde zur Interprozesskommunikation aus dem *POSIX-Thread* ein *Java*-Methode mit einem *Java String*

als Parameter aufgerufen, um eine Nachricht an den anderen *POSIX Thread* zu schicken. Dabei wurde ebenfalls durch die *C/C++*-Seite ein *Java String* erzeugt. Diese lokalen Referenzen werden nach dem *Java*-Methodenaufruf offenbar vom *Garbage Collector* der *Dalvik VM* nicht eingesammelt. Das führt letzendlich zum Überlauf der *JNI-local reference table* und damit unvermeidlich zum Programmabsturz.

Dadurch, dass zu *Debug*-Zwecken in beiden Anwendungen bei jedem Programmdurchlauf verschiedene Parameter eingestellt wurden, folgte, dass *Toasts* und Interprozessnachrichten in unteschiedlicher Häufigkeit und in verschiedenen Intervallen angezeigt/versendet wurden. Daher wurde die *JNI-local reference table* unterschiedlich schnell gefüllt. Das erklärt, warum die Programme zu verschiedenen Zeitpunkten und unabhängig voneinander abstürzten. Das Problem wurde behoben, indem die Referenz beim Erstellen eines *Java Strings* aus *C/C++* nach dem *Java*-Methodenaufruf wieder gelöscht wurde. Listing 5.9 zeigt die dazu notwendigen *JNI*-Operationen.

Listing 5.9: Freigeben von lokalen *JNI*-Referenzen

```
1 // create Java String Object through Dalvik VM and save reference
2 jstring jStrMessage = env->NewStringUTF(charbytes);
3
4 // call Java-method to stream out given char sequence
5 env->CallStaticVoidMethod(gToolsClass, gShowShortToast, jStrMessage);
6
7 // delete local reference
8 env->DeleteLocalRef(jStrMessage);
```

Nachdem das Problem behoben wurde, stürzte die Applikationen nicht mehr durch eine Fehlermeldung ab. Weitere Testläufe ergaben jedoch, dass die Anwendungen weiterhin willkürlich nach einer unbestimmten Zeit nicht mehr ausgeführt wurden, die Interprozessbrücke in einer Richtung unterbrochen wurde und der Schreibvorgang der Log-Daten abbrach. Es gab daher noch mindestens ein weiteres Problem, das gefunden und behoben werden musste.

5.4.2 CPU Wakelock

Zu Debugging Zwecken wurden die *Toast*-Nachrichten, die vom *POSIX*-Thread angezeigt wurden, durchnummeriert, wodurch bei einem Testlauf auf beiden Geräten ein weiteres Problem ermittelt wurde. Während die Anwendungen auf dem *Galaxy S2* im *Standby*-Modus weiterhin *Toasts* angezeigten, wurde die Ausführung auf *Galaxy Nexus* pausiert.

Damit Anwendung während des Standby-Modus weiterhin von der *CPU* ausgeführt werden, muss ein sogenannter *CPU Wakelock* gesetzt werden. Dieses *Wakelock* funktionert mit einem *Reference Counting*-Prinzip. Das bedeutet, dass vom *Android*-Betriebssystem gezählt wird, wie viele Anwendungen einen *CPU Wakelock* gesetzt haben. Solange dieser Counter nicht auf null gesetzt wurde, können Anwendungen während des *Standby*-Modus weiterhin ausgeführt werden. Auf dem *Galaxy Nexus* befanden sich kaum Anwendungen, so dass die meiste Zeit kein *CPU Wakelock* gesetzt war, wodurch dieses Problem entdeckt wurde.
Das Problem wurde behoben, indem die nötigen Funktionalitäten um ein *Wakelock* zu setzten in der *Tools*-Klasse implementiert wurden. Durch die zentrale Verwaltung der dazugehörigen Resourcen zum (De-)Aktivieren eines *CPU Wakelocks* wird sichergestellt, dass unsere Anwendung nicht versehentlich mehrmals ein *Wakelock* aktiviert und vergisst, alle wieder freizugeben.

Nach der Anpassung wurden beide Anwendungen weiteren Testläufen unterzogen. Beide Anwendungen wurden nun konstant und unabhängig vom *Standby*-Modus und den *CPU Wakelocks* anderer Anwendungen ausgeführt. Jedoch wurden die Schreibvorgänge beim Loggen Interprozesskommunikationdaten scheinbar willkürlich nach 30-45 Minuten unterbrochen. Das Aktiveren bzw. Deaktivieren bestimmter Programmmodule wie *Java Threads*, Interprozesskommunikation oder *POSIX Threads* hatte keinerlei Einfluss darauf, ob beide Anwendungen beendet wurden. Die Überlegung lag nahe, dass offenbar die *Services* von Betriebssystem beendet wurden. Im Normalfall beendet Android *Services* oder Anwendungen generell, wenn wenig freier Arbeitsspeicher verfügbar ist. Beide Geräte verfügen über einen *Gigabyte* Arbeitsspeicher. Bei der Ausführung verbrauchten die Anwendungen jeweils ungefähr sechs *Megabyte*. Google gibt in der Dokumentation an,

stets darauf zu beachten ist, einen gestarteter *Service* wieder beendet werden muss, da ansonsten dieser weiterläuft und dadurch Strom verbraucht. Praktisch schien das nicht zu funktionieren, beide Anwendungen wurden stets nach ungefähr 30-45 Minuten beendet. Es musste eine Möglichkeit gefunden werden, das Betriebssystem daran zu hindern, die *Services* zu beenden.

Um *Services* davor zu schützen, vom Betriebssystem beendet zu werden, bietet das *Android Application Framework* die Möglichkeit, einen *Service* im *Foreground*-Modus zu starten. Das *Android OS* behandelt *Foreground-Services* mit höherer Priorität und beendet diese nur im Notfall. Jedoch verlangt das Betriebssystem von der Anwendung, dass bei einer Ausführung des *Services* im *Foreground*-Modus eine *Notification* in der Status Bar angezeigt wird. Da *Services* im *Foreground*-Modus besondere Prioritäten bei der Ausführung genießen, muss der Benutzer immer erkennen können, dass solch ein Hintergrunddienst ausgeführt wird, welcher schließlich Strom verbraucht. Eine *Notification* zu diesem Zweck zu erstellen und diese zu aktualisieren erfordert mehrere Zwischenschritte. Es müssen mehrere Klassen wie der *NotificationBuilder*, *NotificationManager* verwendet werden. Zudem müssen Resourcen wie *Bitmaps* für das *Notification*-Symbol ausgelesen werden. Des Weiteren muss eine *Notification ID* und ein *PendingIntent* erstellt werden. Dieser beschreibt, welche Aktion beim Klicken auf den *Notification*-Eintrag in der Statusbar ausgeführt wird.
Damit diese aufwändigen Vorgänge, nicht jedes mal beim Aktualisieren einer *Notification* nicht den Programmcode aufblähen, wurden diese Funktionalitäten in der *Tools*-Klasse ausgelagert. Durch einfache Funktionsausrufe wie *getNotification()* und *updateNotification()* kann leicht eine *Notification* erstellt und aktualisiert werden. Auch für *C/C++* hat der einfache Funktionsaufruf Vorteile, da ansonsten komplizierter *JNI*-Operationen notwendig sind, um mit den genannten *API*-Klassen eine *Notification* zu erstellen.

5.4.3 Service Foreground Modus und Android API Bug

Nach der Implementation des *Foreground*-Modus in den *Services* wurden weitere Testläufe gestartet. Jedoch wurden weiterhin beide Anwendungen vom Betriebssystem nach 30-45 Minuten geschlossen. Fast jedem *Android*-Benutzer ist ein *Service* im *Foreground*-Modus bekannt. Anwendungen, die *Musik* wiedergeben, sind Beispiele dafür. Diese wer-

den beliebig lang ausgeführt und von Betriebssystem in der Regel nicht beendet. Offentlich gab es noch mindestens einen weiteren Fehler, der dazu führte, dass *Services* im *Foreground*-Modus beendet werden.

Nach vielen Recherchen wurde eine *Bug*-Beschreibung [Rob13] gefunden, die in *Android*-Revisionen mit der *API 14+*(ab Version 4.0 *Ice Cream Sandwich*) auftreten kann. Beim Binden eines Services muss ein *Flag* angegeben werden, welches das Verhalten des *Services* beeinflußt. Dabei gibt es *Flags* und *Flag*-Kombinationen, die unerwünschte Nebeneffekte bei der Ausführung eines *Services* haben können. In unseren Anwendungen wurde dabei der *BIND_ABOVE_CLIENT-Flag* gewählt, welcher dem Betriebsystem mitteilt, dass dieser *Service* mit einer höhreren Priorität ausgeführt werden soll. Dieses *Flag* kann jedoch bei *Android* Versionen ab 4.0 dazu führen, dass das *Betriebssytem* nach den *Service* nach 30 - 45 Minuten beendet.

Um das Prozessverhalten unseren Anwendungen zu untersuchen, wurde das *Android*-Sytemtool *dumpsys* verwendet, welches Informationen über den Systemzustand liefert. Interessant war dabei die *Process LRU Liste* (siehe Listing 5.10, die laufende Prozesse auflistet und deren Priorität angibt. Dabei sind Prozesse, die weiter unten in der Liste aufgeführt werden, Kandidaten, die als nächstes vom Betriebsystem beendet werden. Beim Starten unserer Anwendung wurde der dazugehörige Prozess zunächst als *visible process with an activity (adj=vis)* (Zeile 4) ausgeführt und stand oben in der *Process LRU list*.

Listing 5.10: *dumpsys-LRU*-liste zu Beginn der Programmausführung

```
1 Process LRU list (sorted by oom_adj):
2 PERS # 4: adj=sys  /F  trm= 0 379:system/1000 (fixed)
3 PERS #31: adj=pers /F  trm= 0 607:com.android.nfc/1027 (fixed)
4 Proc # 7: adj=vis  /FA trm= 0 8094:de.hhu.cs.cn.sec/u0a10068 (visible)
5 ...
```

Nach ungefähr 15 - 20 Minuten ergab eine weitere *dumpsys*-Ausgabe jedoch folgendes Bild (Listing 5.11):

Listing 5.11: *dumpsys-LRU*-liste nach 15-20 Minuten

```
1 Process LRU list (sorted by oom_adj):
2 PERS # 4: adj=sys  /F  trm= 0 379:system/1000 (fixed)
```

```
3 PERS #29: adj=pers /F  trm= 0 607:com.android.nfc/1027 (fixed)
4 ...
5 ...
6 Proc #22: adj=bak  /B  trm= 0 8094:de.hhu.cs.cn.sec/u0a10068 (bg-
      services)
```

Unser Prozess wurde vom Betriebssystem als *empty background processa (dj=bak)* (Zeile 6) markiert und befand sich unten in der *Process LRU*-Liste. Vom Betriebssystem werden solche Einträge als Aufforderung gewertet, als nächstes beendet zu werden. Tatsächlich wurden die *Services* nach insgesamt 30 - 40 Minuten beendet.

Die Anwendungshüllen wurden angepasst, so dass diese im *Foreground*-Modus gestartet wurden und die Bindung der *Services* mit Standart-*Flags* (*BIND_AUTO_CREATE*) durchgeführt wurde. Die darauffolgenden Testläufe verliefen erfolgreich. Die *Services* liefen sieben Tage ohne Unterbrechung durch, bis sie manuell per Hand beendet wurden. Die Interprozessbrücke blieb stabil, so dass Nachrichten während der Laufzeit problemlos in beide Richtungen ausgetauscht werden konnten. Auch das Protokollieren der Interprozesskommunikation wurde nicht unterbrochen und funktionierte ohne Unterbrechungen. Die entwickelte Anwendungshülle konnte erfolgreich stabilisiert werden.

5.5 Anpassung und Integrierung des SEC-Clienten

In den vorherigen Abschnitten wurde erläutert, wie eine stabile *Java*-Anwendungshülle mit integrierter *JNI*-Schnittstelle erstellt wurde. Diese gewährleistet eine nebenläufige Ausführung und stellt zudem über das *jniglue*-Modul eine Schnittstelle bereit, über die der *Java*-Code mit dem *C/C++*-Code der Anwendung kommunizieren kann. Die notwendige Kommunikation zwischen dem *SEC-Clienten* und der *Java*-Hülle induziert, dass Anpassungen am *C++*-Code des *SEC-Clienten* vorgenommen werden müssen. Eine direkte *Android*-spezifische Anpassung im Quellcode hätte jedoch den Nachteil, dass der Quellcode sich nicht mehr für normale *Linux*-Desktop Computer kompilieren lässt. Man müsste daher den ursprünglich Code beibehalten und bei jeder Weiterentwicklung des *SEC-Clienten* eine Anpassung vornehmen, um den *SEC-Clienten* in die *Java*-Anwendungshülle der *Android*-Anwendung integrieren zu können. Diese notwendigen

Anpassungen würden jedoch den Vorteil, den die Benutzung des *NDKs* mit sich bringt, relativieren. Bereits geschriebener *C/C++* Code kann nicht direkt übernommen werden und müsste verändern werden. Das Ziel war es daher, den Quellcode des *SEC-Clienten* so anzupassen, dass dieser sich sowohl für *Linux*-Dektop Computer als auch für die *Android*-Anwendungshülle kompilieren und verwendet lässt, ohne vorher plattformspezifische Änderungen vornehmen zu müssen. Zu diesem Zweck wurde die *C++* Klasse *OSHelper* geschrieben. Diese kapselt Betriebssystemspezische Funktionen, so dass der Quellcode transparent und plattformunabhängig bleibt.

5.5.1 Die C++ OSHelper-Klasse

Die *OSHelper*-Klasse stellt statische Funktionen bereit um betriebssytemspezifische Operationen an erforderlichen Stellen auszuführen. Durch *Präprozeossor*-Anweisungen wird eine bedingte Kompilierung für ein bestimmtest Betriebssystem erzwungen (siehe Listing 5.12). Dadurch wird gewährleitet, dass beispielsweise nur Anweisungen ausgeführt werden, die unter *Android* bzw. *Linux* erforderlich sind. Zur Unterscheidung der Zielplattform bei der Kompilierung kann nicht das *Compiler*-Symbol *__linux__* verwendet werden, da *Android* auf den *Linux*-Kernel basiert und dieser im *NDK Buildsystem* ebenfalls definiert ist. Zwar bietet das *NDK* das Symbol *__ANDROID__* an, um einen Kompiliervorgang für ein *Android*-Gerät zu identifizieren, jedoch lässt sich damit nicht unterscheiden, ob eine *Shared Library* (mit Schnittstelle für *Java*-Hülle) oder eine *ELF Executable* (ohne *Java*-Hülle) erstellt wird.

Zur klaren Identifizierung definieren wir daher ein eigenes Symbol *__ANDROID_APP__*, die in der *Android Makefile* zur Kompilierung einer dynamischen Bibliothek eingetragen wird. Innerhalb der *OSHelper*-Klasse werden durch bedingte *Präprozessor*-Befehle die für die *Android*-Anwendung notwendigen *Header*-Dateien eingebunden bzw. Anweisungen ausgeführt. Um den Code der *OSHelper*-Klasse selbst transparent zu erhalten, werden *Android*-spezifische Funktionen an das *jniglue*-Modul delegiert. Intern arbeitet die *OSHelper*-Klasse mit einer *Singleton*-Instanz, wodurch der Zugriff auf die *Java*-Hülle kontrolliert durchgeführt wird. Ein weitere Vorteil der *Singleton*-Instanz ist der Umstand, dass spätestens beim Beenden des Prozesses die *Destructor*-Funktion aufgerufen wird, welche Aufräumarbeiten erledigt. Details zu den Funktionsnalitäten und Implementie-

rung der *OHHelper*-Klasse werden in den nächsten Abschnitten erläutert.

Listing 5.12: Verwendung bedingter Kompilierung innerhalb der *OSHelper*-Klasse

```
1 #ifdef __ANDROID_APP__
2         ...
3         // include android specific header
4         // and/or delegate to jniglue-function
5         ...
6 #endif
```

5.5.2 Starten des SEC-Clienten

Als *C++*-Konsolenprogramm wird der SEC-Client durch die *main*-Funktion gestartet und erhält dabei seine Startparameter als *Zeiger* auf ein *char-Array*. Als Parameter werden die *IP*-Adresse des *SEC-Servers* und der *Port* zur Verbindungsanfrage benötigt. Damit die *main*-Funktion von der *Java*-Anwendungehülle durch die *JNI*-Schnittstellt aufgerufen werden kann, wurde zur *cpp*-Datei, welche die *main*-Funktion erhält (*Sec.cpp*), eine entsprechende *Header*-Datei erstellt. Diese enthält eine Deklaration der *main*-Funktion, so dass diese von außen aufgerufen werden kann. Diese Anpassungen am Quellcode haben bei der Kompilierung für *Android* oder *Linux* keine Auswirkungen und können daher im Projekt beibehalten werden.

Auf der *Java*-Seite der Anwendung werden die Startparameter als *String-Array* verpackt und durch Aufruf der nativen Funktion *nStartSEC(String args[])* an die *JNI*-Schnittstelle zur weiteren Verarbeitung übergeben (siehe Listing 5.13).

Listing 5.13: Übergabe und Konventierung von Programmstart-Parametern

```
1
2 // get size of given array
3 jsize numOfArgs = env->GetArrayLength(jobjarr);
4
5 char *args[numOfArgs];
6
7 for(int i=0; i<numOfArgs; i++) {
8
```

```
 9    // get i. element out of object array, convert to jstring
10    jstring jstr = (jstring) env->GetObjectArrayElement(jobjarr, i);
11
12    // convert jstring to c string and store it in a temporary buffer
13    const char *charbuffer = env->GetStringUTFChars(jstr, 0);
14
15    // get length in order to copy this c string into a new char array
16    int cstrLength = strlen(charbuffer);
17
18    // create space for char array (add 1 for '\0' termination symbol)
19    args[i] = new char [cstrLength + 1];
20
21    // copy cstring, cause charbuffer is bound to jstr, for
22    // jni reason we've to release it as soon as possible
23    strcpy(args[i], charbuffer);
24
25    // release charbuffer
26    env->ReleaseStringUTFChars(jstr, charbuffer);
27
28    // delete local ref
29    env->DeleteLocalRef(jstr);
30 }
31
32 // start SEC!
33 main(numOfArgs, args);
```

Da C/C++ keine *Java*-Klassen kennt, wird der *Java-String-Array* durch den *JNI*-Datentyp *jobjectArray* repräsentiert. Zunächst müssen die einzelnen Elemente extrahiert werden (Zeile 10). Der Inhalt wird in einen temporären *char*-Buffer kopiert (Zeile 13), um ihn in ein vorbereitetes *char*-Array zu übertragen (Zeile 19). Nach dem Kopiervorgang müssen die verwendeten *JNI*-Referenzen wieder freigegeben werden (Zeile 26 und 29), um Programmabstürze zu vermeiden. Die Parameter liegen nach dieser Prozedur in einem Format vor, womit der *SEC-Client* gestartet werden kann (Zeile 3). Es ist jedoch zu beachten, dass der erste Eintrag im *Array* auf den Programmnamen oder Ausführungsumgebung verweist und daher die eigentlichen Parameter ab der zweiten Position im Array gespeichert werden.

5.5.3 Verbindungaufbau zum Server und Speichern der Experimentdateien

Nachdem der *SEC-Client* gestartet wurde, verbindet sich dieser mit dem *SEC-Server*, um anstehende Experimente in Form von Dateien herunterzuladen und diese im Ausführungsverzeichnis zu speichern. Wie in Kapitel 4 beschrieben wurde, sind für Netzwerk- und Dateisystemoperationen entsprechende *Android Permissions* notwendig. Diese werden in der *AndroidManifest*-Datei eingetragen und erfordern daher keine weiteren Anpassungen im *C++* Code des *SEC-Clienten*.

Bezüglich der Dateioperationen muss jedoch eine Besonderheit des *Android*-Betriebssystem beachten werden. *Android* Anwendungen werden im sogenannten *Internal Storage* eines *Androids* Gerätes installiert. Der *Internal Storage* enthält wichtige Systempartitionen [Bra13], auf die der normale *Android* Benutzer nicht ohne *Root*-Berechtigung zugreifen kann. Innerhalb des *Internal Storage* werden die nativen Bibliotheken einer *Android*-Anwendung im Unterverzeichnis *data/app.package.name/lib* abgelegt. Dies ist somit das Arbeitsverzeichnis des *SEC-Clienten*. Da Schreibzugriffe auf diese Partion nur über *Root*-Zugriffe erlaubt sind, kann der *SEC-Client* keine Dateien in diesem Arbeitsverzeichnis speichern. Zwar meldet *fstream* kein Fehler, der Schreibvorgang wird vom Betriebssystem jedoch nicht physisch durchgeführt. Ein Lesevorgang führt zu einer *Exception* bei der Ausführung des *SEC-Clienten*.

Im Gegensatz zum *Internal Storage* dürfen Anwendungen mit einer entsprechenden Berechtigung auf den *External Storage* Dateioperationen durchführen. Der *External Storage* beinhaltet alle Partionen, worauf auch der Benutzer ohne *Root*-Berechtigung zugreifen kann (Beispielsweise Musik-, Foto-, Videoverzeichnis oder eine externe Speicherkarte). *Android* bietet dabei einen Anwendungsspezifischen *Cache*-Ordner zum Speichern von Anwendungsdaten. Dieser *Cache*-Ordner wird allerdings bei einer Deinstallation der Anwendung ebenfalls gelöscht. Da bei unseren Anwendungen die permanente Speicherung von Messdaten wichtig ist, nutzen wir einen anwendungsspezifischen Ordner im Hauptverzeichnis des *External Storages*. Anwendung und Benutzer können problemlos auf die Daten zugreifen. und ein versehentliches Löschen durch das Betriebssystem wird vermieden.

Damit der *SEC-Client* plattformunabhängig beim Speichern und Lesen von Dateien das richtige Verzeichniss verwendet, stellt die *OSHelper*-Klasse die statische Funktion *OSHelper::getFullFilePath(fileName)* zur Verfügung. Während diese Anweisung unter *Linux* den Pfad einer Datei nicht verändert, wird unter *Android* durch die *JNI*-Schnittstelle an die *getExternalStorageAppPath()*-Methode der *Tools*-Klasse delegiert. Dieser ermittelt dynamisch das passende Arbeitsverzeichnis für eine Anwendung. Da die *fstream*-Klasse der *C++ Standart Bibliothek* unter *Android* keine Verzeichnisse neu anlegen können, stellt die *getExternalStorageAppPath()*-Methode der *Tools*-Klasse sicher, dass das entsprechende Verzeichniss bei Bedarf neu angelegt wird.

5.5.4 Starten von (Mess-)Anwendungen

Durch den *SEC-Clienten* durchgeführte Experimente, werden in einer *XML*-basierten Datei beschrieben. Dabei werden (Mess-)Programme, die gestartet werden sollen, innerhalb eines <*system*> oder <*include*>[4]-Tags angeben. Der eigentliche Befehle zum Starten eines (Mess-)Programmes wird innerhalb der *Tags* im *Linux Shell*-Form kodiert.Beispielsweise sieht ein Eintrag in einer Experimentdatei folgendermaßen aus:

```
<system>./Messprogramm parameter1 parameter2 ...</system>
```

Der durch den *Tag* eingeschlossene Befehl wird durch die *system()*-Funktion der *C-Bibliothek stdlib.h* ausgeführt. Wie in Abschnitt 4.2.4 erläutert wurde, werden *Android* Anwendungen jedoch durch *Intents* gestartet. Diese Funktionalitäten stehen im *Application-Framework* zur Verfügung, und sind nicht Bestandteil der *NDK*-Bibliothek. Des Weiteren wird zum Starten einer *Android*-Anwendung der dazugehörige *Package*-Name benötigt. Abgesehen davon, dass wir an dieser Stelle eine Anpassung des *SEC-Clienten* durchführen müssen, kann der Inhalt einer gewöhnlichen Experimentdatei nicht verwendet werden, um eine *Android*-Application zu starten. Damit jedoch ein Experiment unabhängig von den verwendeten *Client*-Plattformen (*Android* oder *Linux*) verwendet wer-

[4] <*include*>*-Tags* dienen dazu, mit (Mess-)Programmen eine lokale *TCP*-Verbindung aufzubauen, wodurch weitere Befehle an diese gesendet werden können. Dabei wird innerhalb des *Tags* eine Port angegeben, an welchem das (Mess-)Programm für ein Verbindungsaufbau lauschen soll. Zusätzlich wird im *Tag* der (Mess-)Anwendung ein *Alias*-Namen vergeben. Mit diesem Namen können weitere Anweisungen in der Experimentdatei auf ein bestimmtes (Mess-)Programm bezogen werden.

den kann, wurde das Protokoll zur Experimentbeschreibung angepasst, so dass *Android*-spezifische Angaben gemacht werden können, wie folgender Beispiel-*Tag* zeigt:

```
<system android-package="packagename">...</system>
```

Durch Verwendung des *android-packagename*-Attribut kann der *Package*-Name einer *Android*-Anwendung mitangeben werden. Beim Ausführen eines Experiments ruft der *SEC-Client* dabei die Funktion *OSHelper::executeInOS()* auf, die durch Delegation an die *Java*-Hülle sicherstellt, dass die entsprechende *Android*-(Mess-)Anwendung gestartet wird. Die *Java*-Hülle des *SEC-Clienten* erstellt dazu einen passenden *expliziten Intent* und hängt die Startparameter als Zusatzinformation an den *Intent* dran.
Diese Veränderungen am Experimentbeschreibungs-Protokoll und am *SEC-Clienten* sind plattformunabhängig implementiert, so dass als *Clienten* sowohl *Android*- als auch *Linux*-Geräte teilnehmen können.

5.5.5 Programmausgaben des SEC-Clienten

Als Kommandozeilenprogramm gibt der *SEC-Client* Programmausgaben über das *C++ Standard Output Stream*-Objekt *cout* bzw. Fehler über *cerr* aus. Das *Android*-Betriebsystem leitet dabei diese Standardausgaben nach */dev/null/* um. Daher sind Programmausgaben des *SEC-Clienten* nicht direkt auf der Benutzeroberfläche der *Android*-Anwendung sichtbar und erfordern eine Anpassung. Um den Quellcode des *SEC-Clienten* weiterhin plattformunabhängig zu halten, so dass für Programmausgaben weiterhin die *C++*-Standardausgabe Objekte *cout* und *cerr* verwendet werden können, müssen diese bei der Ausführung als *Android*-Anwendung zur *Java*-Anwendungshülle umgeleitet werden. Dazu bieten sich zwei grundlegende Ansätze an.

Zum einem könnte man eine *Wrapper*-Klasse für *cout/cerr* schreiben, die durch Überladen des «-Operator eine ähnliche Schnittstelle wie die *cout/cerr*-Objekte bietet und Ausgaben zur *Java*-Hülle umleitet. Durch bedingte Kompilierung könnte man durch den *DEFINE-Präprozessor*-Befehl eine Instanz der *Wrapper*-Klasse als *cout/cerr* definieren. Diese Methode hat jedoch einige Nachteile. Da die *Wrapper*-Klasse als Ersatz für *cout/-cerr* dient, müssen auch alle Funktionen zur Verwendung von Zeichen-*Manipulatoren*

implementiert werden. Des Weiteren muss in jeder Quellcode-Datei ein *Redefine* von *cout/cerr* stattfinden, die unter Umständen zur Kollision mit anderen Klassen der *C++ Standard Bibliothek* führen kann.

Ein anderer Ansatz ist es, den internen *Streambuffer* von *cout/cerr* durch eine Instanz einer eigener *streambuf*-Klasse auszutauschen. Der Vorteil ist dabei, dass weiterhin die normale *cout/cerr*-Instanz verwendet werden kann, und somit die Implementation der Schnittstelle für *Manipulatoren* und andere *Stream*-Funktionen entfällt. Wir können intern durch unsere eigene *Stream Buffer*-Klasse direkt auf die fertig formatieren Zeichen zugreifen und diese auf der *Android Java*-Seite ausgeben. Des Weiteren vermeiden wir durch Redefinierung der *cout/cerr*-Objekte Kollisionen mit andern Klassen der *C++ Standard Bibliothek*.

Für diesen Zweck wurde die *C++*-Klasse *androidbuf* entwickelt, welche von der *streambuf*-Klasse erbt und dafür sorgt, dass der Inhalt des *Stream Buffers* über die *JNI*-Schnittstelle zur *Java*-Seite transferiert wird. Damit wir den Inhalt des Buffers bei einer Synchronisierung oder einem Überlauf entleeren und in der *Android*-Anwendung anzeigen können, überschreiben wir die *virtuellen* Funktionen *sync()* und *overflow()*. Diese Technik erlaubt es uns, *Strings* in einer *Android*-Anwendungen flexible an verschiedenen Stellen auszugeben. Beispielweise kann eine Ausgabe über das *LogCat*, als *Toast* oder über die *Notifaction*-Bar erfolgen. Bevor wir die *Stream Buffer* der *cout/cerr*-Instanzen ersetzen, ist zwingend erforderlich, da bei der Terminierung der Anwendung die *cout/cerr*-Instanzen versuchen, die Stream Buffer zu entleeren, was zur Speicherzugriffsverletzung führen würde.

Diese Funktionalitäten sind in der *OSHelper*-Klasse implementiert (siehe Listing 5.14), die bei einer Ausführung unter *Android* für eine Umleitung der Standardausgabe sorgt. Dadurch, dass die statische *OSHelper*-Klasse intern als *Singleton* konzipiert ist, kann der Schreibpuffer der Standardausgabe nicht mehrmals umgeleitet werden. Zudem sorgt der *Destructor* der *Singleton*-Instanz dafür, dass beim Programmende der Standart-Schreibpuffer wieder hergestellt wird. Das Kapseln der geschilderten Funktionalitäten durch die *OSHelper*-Klasse macht den Einsatz sicher, sorgt für Transparenz und ermöglicht einen plattformunabhängigen Quellcode des *SEC-Clienten*.

Listing 5.14: Umleitung der *cout/cerr*-Ausgaben durch die *OSHelper*-Klasse

```
1 // Sec.cpp
2 int main(int argc, char *argv[])
3 {
4   // call function at program start to redirect to Android Java side
        if necessary
5   OSHelper::redirectCout();
6   ....
7   // restore original stream buffer before program terminates
8   OSHelper::restoreCout();
9 }
```

5.5.6 Beenden des SEC-Clienten

Der *SEC-Client* ist als endlicher Zustandsautomat entwickelt worden, der lediglich zwischen Zuständen wechselt und diese niemals verlässt. Unter *Linux* wird der *Client* durch die Tastenkombination *Str + C (SIGINT Signal)* beendet. In unserer *Android*-Anwendung stellt dieser unendlich laufende Zustandsautomat jedoch ein Problem dar. Native Anwendungen, die innerhalb einer *Android*-Applikation ausgeführt werden, sind an den Prozess gekoppelt und nicht an die Grundbausteine einer Anwendung wie *Acitvities* und *Services*. Beim Entladen dieser Komponenten und der damit vermeintlichen Anwendungsschließung läuft der *SEC-Client* im Hintergrund weiter. Um das native Programm beim Beenden des *Services* ebenfalls zu schließen bieten sich folgende Möglichkeiten an. Zum einen könnten wir den *Java-Thread*in dem der *SEC-Client* letztendlich ausgeführt wird, zwingen, sich zu beenden, jedoch ist in der *Java*-Dokumentation vermerkt, möglichst auf diese Methode zu verzichten. Ein manueller Stop des *Threads* ruft einen unvorhersehbaren Zustand der *Dalvik VM* hervor. Aufräumarbeiten in einer Anwendung könnten nicht mehr durchgeführt werden und Resourcen (Bsp. *CPU Wakelock, Notificationbar*) könnten nicht wieder freigegeben werden.

Eine andere Möglichkeit ist es, den *SEC-Clienten* in einem nativen *POSIX Thread* auszuführen und diesen mit der Funktion *pthread_cancel()* zu beenden. Leider ist diese Funktion aktuell sowohl im *Google NDK* als auch im *Crystax NDK* nicht implementiert. Alternativ kann die *pthread_kill()*-Funktion verwendet werden, um definierte Signale an

den *POSIX Thread* zu schicken. Allerdings führen Signale wie *SIGKILL* oder *SIGTERM* zum Beenden des gesamten Prozesses und damit zur unkontrollierten, sofortigen Terminierung der gesamten *Android*-Anwendung. Andere Signale wie *SIGUSR1* werden ignoriert oder nicht empfangen.

Ein weiterer Ansatz ist es, den Programmcode des *SEC-Clienten* zu verändern, so dass dieser den Zustandsautomat verlässt und damit das Ende der *main*-Funktion erreicht. Um dies kontrolliert und plattformunabhängig zu implementieren, wurde in der *Sec.h*-Datei, die als Schnittstelle für die *JNI*-Schnittstelle dient, die Funktion *closeSEC()* implementiert. Diese ruft in einer tieferen Ebene des Programmes die *closeClient()*-Funktion auf(siehe Listing 5.15). Die *closeClient()* selbst setzt ein *Flag* im Programm (Zeile 4). Dieser Schalter wird von der Schleife, die den Zustandswechsel kontrolliert, bei jedem Durchlauf überprüft. Somit kann die Schleife abgebrochen werden und das Programm sauber beendet werden. Eine Besonderheit bei der Implementierung war es, dabei den Zustandsautomaten zum Verlassen eines Zustandes zu zwingen, so dass die Kontrollschleife weiter ausgeführt werden kann und das Beenden-*Flag* überprüft werden kann. In einigen Zuständen werden Netzwerkfunktionen ausgeführt, bei denen auf den Empfang von Daten gewartet wird und die Programmausführung nicht fortgesetzt wird. Währenddessen kann das gesetzte *Flag* zum Beenden des Programmes nicht überprüft werden. Die *closeClient()*-Funktion schließt daher zusätzlich zum Setzen des *Flags* alle bestehenden Netwerk-*Sockets* (Zeile 7), so dass entsprechende Zustände, in denen Netzwerkoperationen stattfinden, verlassen werden und eine Überprüfung des Beenden-*Flags* in der Zustandskontrollschleife stattfindet. Der Vorteil bei dieser Implementierung ist, dass durch Verlassen der *main*-Funktion ein sauberes Beenden des nativen *C/C++*-Programmes durchgeführt wird und zudem der Quellcode weiterhin plattformunabhängig bleibt.

Listing 5.15: Die *closeClient()-Funktion*

```
1 // ClientControl.cpp
2 void ClientControl::closeClient(){
3   // set flag
4   m_keepClientAlive = false;
5
6   // close connections to force SEC client to leave state
7   m_tcpClient->closeConnection();
8 }
```

5.6 Anpassung und Integrierung des RMF

Der *C++* Code des *RMF* wurde bereits in der Arbeit [Olf13] erfolgreich für die Verwendung unter *Android* angepasst, so dass dieser analog zum *SEC-Clienten* in die *Java*-Anwendungshülle eingesetzt wurde. Zum Starten der *Rate Measurement Frameworks* durch die *Android SEC-Client*-Applikation wird zunächst in einer Experimentdatei wie in Abschnitt 5.5.4 innerhalb eines <*include*> oder <*system*>-*Tags* zum Starten des *RMF* der *Package*-Name des *Service*-Komponente angegeben, welche das *RMF* ausführt. In unserem Fall lautet der Pfad `de.hhu.cs.cn.rmf.service.ServiceRMF`. Im *SEC-Clienten* wird beim Ausführen des Experimentes durch die *OSHelper*-Klasse der Befehl zum Starten des *RMF*-Services zusammen mit den Startparamater durch die *JNI*-Schnittstelle an die *Java*-Hülle delegiert. Dieser erstellt einen passenden *expliziten Intent*, der losgeschickt wird. Der *RMF*-Service wird gestartet und extrahiert aus dem *Intent* die Startparameter für den *RMF*. Ein *Java*-Thread zur nebenläufigen Ausführung wird erstellt. Durch die *JNI*-Schnittstelle startet der *Thread* das *Rate Measurement Framework* mit den gewünschten Parametern.

5.7 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie sukzessiv eine *Java*-Anwendunghülle entwickelt und getestet wurde. Dabei mussten Besonderheiten des *Android*-Betriebssytem beachtet werden und einige Hürden überwunden werden, um eine stabile Ausführung zu gewährleisten. Nachdem die Anwendungshüllen fertiggestellt waren, konnte der *C++*-Code des *SEC-Clienten* eingesetzt werden. Für Interaktion des *C++*-Codes mit der *Java*-Seite waren Anpassungen des *SEC-Clienten*-Quelltextes notwendig. Hier lag die Herausforderung, die notwendigen Änderungen plattformunabhängig und tranparent vorzunehmen. Dieses Ziel konnte mit Hilfe der entwickelten *OSHelper*-Klasse und den beschriebenen Techniken erreicht werden. Der Quellcode bedarf bei der Kompilierung für *Linux* oder *Android*-Geräten keinerlei Anpassungen und wird mit entsprechenden Makefiles durchgeführt. Analog dazu wurde das für *Android* portierte *RMF* in die entwickelte Anwendungshülle eingesetzt und lässt sich mit dem *Android SEC-Clienten* verwenden.

Kapitel 6

Vergleich von Android Applikationen und mit Executables

Im vorherigen Kaptiel wurde ausführlich beschrieben, wie der *SEC-Client* erfolgreich als *Android*-Applikation implementiert wurde. Die entwickelte Anwendung besteht dabei aus einer Anwendungshülle, die in *Java* implementiert ist und dem *C++*-Code des *SEC-Clienten*. Aus dem nativen Code wird mit Hilfe des *NDK* eine dynamische Bibliothek erstellt, die zur Laufzeit in die *Android*-Anwendung eingebunden wird.

In Abschnitt 4.1 wurde bereits angeschnitten, dass es mit dem *NDK* auch möglich ist, *C/C++* Code als komplett native Programme im *ELF*-Format[1] für *Android*-Geräte zu kompilieren. In den folgenden Abschnitten werden verschiedene Aspekte beider Methoden beleuchtet. Am Ende des Kapitels folgt eine tabellarische Übersicht mit den Vor- und Nachteilen beider Techniken.

6.1 Entwicklung und Anpassungen

C/C++-Programme, die mit dem *NDK* als *Executables* kompiliert werden, benötigen keine *Java*-Hülle zur Ausführung unter *Android*. Die Entwicklung der *Java*-Komponente

[1] Das *Executable and Linkable Format* beschreibt das Standard-Binärformat für ausführbarer Programme unter UNIX-ähnlichen Betriebssystemen wie beispielsweise Linux

und die damit verbundene Verwendung des *JNI* entfällt.

Die Verwendung von *C/C++*-Code durch dynamische Bibliotheken in *Android*-Anwendungen erfordert die Entwicklung einer *Java*-Hülle. Wie in Kapitel 4 und 5 beschrieben wurde, kann die entwickelte *Java*-Anwendungshülle als Vorlage verwendet werden, um andere native Programme zu integrieren. Der Aufwand, eine komplett neue *Java*-Hülle zu schreiben, entfällt daher bei der Portierung von weiteren C/C++-Programmen.

6.2 Verfügbare Programm-Bibliotheken

Wie der Author in [Rat12] beschreibt, ist es nicht möglich, eine *Android*-Anwendung ohne *Java* zu schreiben. *Android*-Anwendungen basieren auf dem *Android-Application-Framework* (siehe Abschnitt 3.1.1.3), welches hauptsächlich in *Java* implementiert ist. *Android*-Anwendungen enthalten daher immer einen *Java*-Anteil und haben einen bestimmten Aufbau (*AndroidManifest*-Datei , *Activites*, *Service* etc.). Komplett native Programme im *ELF*-Format sind keine *Android*-Anwendung im klassischen Sinne und werden nicht auf der Anwendungsschicht der *Android*-Systemarchitektur ausgeführt. *Executables* werden daher nicht durch den *Zygote*-Prozess (siehe Abschnitt 3.1.1.1) gestartet, der eine Instanz der *Dalvik-VM* bereitstellt, geschweige denn Referenzen auf *VM* bereitgestellt um *JNI*-Operationen durchzuführen. Ein Zugriff auf das *Appication-Framework* ist nicht möglich, so dass *Executables* sich auf die Verwendung der *NDK*-Bibliothek beschränken müssen.

Im Gegensatz zu *Executables* sind dynamische Bibliotheken in *Android*-Anwendungen eingebettet. Durch die *Dalvik VM*-Instanz des dazugehörigen *Zygote*-Prozesses sind *JNI*-Operationen möglich. Theoretisch kann daher auf den gesamten Funktionsumfang des *Android-Application-Framework* und der *NDK*-Bibliotheken zugegriffen werden. Durch den Zugriff auf das *Application-Framework* sind dynamische Bibliotheken bei der Verwendung mit einer *Java*-Anwendungshülle klar im Vorteil bezüglich des Funktionsumfangs. Beispielsweise kann der die *Android SEC-Client*-Anwendungen durch Verschicken von *SMS*-Nachrichten über den Status einer Experimentausführung unterrichten.

Dieses Feature wurde in der *Tools*-Klasse implementiert und konnte nur durch Zugriff des *Android-Application-Frameworks* realisiert werden.

6.3 Ausführung

Da mit dem *NDK* kompilierte *Executables* Kommandozeilenprogramme darstellen, werden diese durch die *Shell* des *Android Linux-Kernel* ausgeführt. Dies kann mit Hilfe einer *Terminal*-Applikation geschehen oder über *ADB* erfolgen.

Weder das *Android SKD*, noch das *NDK* bieten die Möglichkeit, *Executables* direkt auf ein *Android*-Gerät zu installieren und auszuführen. Die dazu notwendigen Schritte müssen manuell vorgenommen werden. Zunächst müssen *Executables* auf das *Android-Gerät* kopiert werden. Als normaler Benutzer hat man dabei nur Zugriff auf den *External Storage*. Diese Partition ist jedoch für die Ausführung von *Executables* vom System aus gesperrt. Der *Internal Storage* erlaubt die Ausführung von *Executables*, so dass *ELF*-Dateien zunächst dort abgelegt werden müssen. Der Zugriff auf den *Internal Storage*, der wichtige System-Partitionen des *Android*-Betriebssystemes enthält, wird allerdings nur durch *Root*[2]-Rechte gewährt. Diese Administrator-Rechte werden aus Sicherheitsgründen werkseitig dem Benutzer nicht gewährt. Der Vorgang, dem Benutzer diese besonderen Rechte zu verleihen, wird als *rooten* bezeichnet. Je nach Modell und verwendeter Betriebssystem-Version kann sich dieser Prozess als aufwändig gestallten. Des Weiteren birgt der *Root*-Vorgang die Gefahr, Schäden an Systempartitionen zu verursachen. Bei vielen Herstellern erlischt zudem die Garantie.

Nachdem durch *Root*-Rechte eine *Executable* auf den *Internal Storage* abgelegt wurden, muss die Dateiberechtigung angepasst werden, so dass der Benutzer die *Executable* ausführen darf. Auch dafür sind *Root*-Rechte notwendig. Zum Starten der *Executable* sind unter Umständen ebenfalls wieder *Root*-Rechte erforderlich, um auf das Ausführungsverzeichnis zuzugreifen oder wenn beispielsweise die auszuführende *ELF* Dateioperationen auf dem *Internal Storage* durchführen möchte. Dieses Verfahren wurde auf

[2]Durch *Rooten* eines *Android*-Gerätes erhält man die größtmöglichste Rechte beim Durchführen von Operationen auf ein System oder Gerät. Vergleichbar mit einem Adminstrator

den Testgeräten getestet. Auf dem *Galaxy Nexus* war es trotz *Root*-Rechte nicht möglich über *ADB* oder die *Shell* Dateien auf dem *Internal Storage* abzulegen. Dies konnte nur mit speziellen *Root*-Anwendungen erreicht werden. Auf dem ebenfalls *gerooteten Galaxy SII* war es im Rahmen dieser Arbeit gänzlich unmöglich, *Executables* erfolgreich auszuführen.

Im Gegensatz zu nativen *Executables* kann die entwickelte *Android*-Anwendung problemlos durch *Android*-Entwicklungstools per Knopfdurck kompiliert, installiert und gestartet werden. Für diese Vorgänge sind keine *Root*-Rechte erforderlich. Auch in diesem Punkt ist die *Android*-Anwendung deutlich im Vorteil.

6.4 Sicherheit und Tranparenz

Executables sind nicht als *Android*-Anwendung zu betrachten und werden nicht in der Anwendungschicht (Abbschnitt 3.1.1.4) des *Android*-Betriebssystemes ausgeführt. Als komplett natives Programm werden sie auf der Schicht über den *Linux-Kernel* (Abschnitt 3.1.1.2) ausgeführt und umgehen damit komplett das Rechtesystem Androids. Der Benutzer hat keine Kontrolle darüber, auf welche Geräteresourcen die *ELF*-Datei bei der Ausführung zugreift. Ebenfalls werden *Android*-Features wie Energiesparfunktionen ausgehebelt, so dass ein rechenintensiver nativer Prozess schnell den Akku aufbrauchen kann. Dadurch, dass die *Executables* mit einer *Root*-Berechtigung ausgeführt werden, können diese unter Umständen wichtige Systemdateien oder Partitionen beschädigen. Des Weiteren können *ELF*-Dateien unsichtbar und unbemerkt vom Benutzer im Hintergrund ausgeführt, was die Transparenz und Kontrolle über das Programm stark einschränkt.

All die genannten Nachteilen treffen auf *Android*-Anwendungen nicht zu. An das Sicherheits- und Rechtesystem *Androids* gebunden, weiß der Benutzer vor der Installation welche Berechtigungen zur Ausführung benötigt werden. Die Anwender hat stets Kontrolle über eine Anwendung und kann diese bei Bedarf schließen. Zudem können sich Anwendung bei der Ausführung nicht gänzlich verstecken, da *Android* Systemanwendungen zur Verfügung stellt, die alle ausgeführten *Android*-Applikationen anzeigen.

6.5 Performance

Sowohl bei der Ausführung als *Executable*, als auch eingebettet in einer *Java*-Hülle als *dynamische Bibliothek* findet die Ausführung des kompilierten *C/C++* Programmcodes direkt auf dem Prozessor statt. Die *Dalvik VM* führt lediglich den *Java*-Code einer *Android*-Anwendung aus. Bezüglich der Performance macht es daher keinen Unterschied, in welcher Form nativer Code in *Android* ausgeführt wird:

6.6 Distribution

Die Weitergabe und Installation von *Executable* ist mit den in den vorherigen Abschnitten ausgeführten Nachteilen verbunden. Ein Benutzer braucht zwingend *Root*-Rechte, um eine *ELF*-Datei zu installieren und auszuführen. Diese Vorgänge sind umständlich und erfordern technische Kentnisse über das *Android*-Betriebssystem. Unter Umständen kann eine *Executable* auf einem Gerät gar nicht ausgeführt werden (Bsp. *Galaxy SII*). Das Programm selbst ist bezüglich des Resourcenzugriffs auf ein Gerät nicht transparent. Zudem kann die Sicherheit des Systems nicht gewährleistet werden.

Android-Anwendungen können problemlos als *apk*-Archivdatei oder über den *Google Play Store* verteilt werden. Zur Installation und Verwendung einer *Android*-Anwendung muss ein Benutzer nicht über besondere technische Kentnisse verfügen. Das Sicherheits- und Rechtesystem *Androids* gewährleistet eine höhere Transparenz einer *Android*-Applikation.

6.7 Tabellarische Übersicht

Im Folgenden findet sich eine tabellarische Übersicht der in diesem Kapitel beschriebenen Vor- und Nachteile bei der Verwendung einer *dynamische Bibliothek* (impliziert die Verwendung einer *Android*-Anwendung als Hülle) oder einer *Executable* um nativen Code auf dem *Android*-Betriebssystem auszuführen. Vor- und Nachteile werden dabei mit einem + bzw. - gekennzeichnet

	dynamische Bibliothek	Executable
Entwicklung	- Java-Hülle erforderlich	+ keine Anpassung erforderlich
verfügbare Programm-bibliotheken	+ *NDK* und *Android-Application-Framework*	- nur *NDK*
Kompilierung und Test	+ komfortabel über *Android SKD* + echte *Android*-App	- *Root* erforderlich zur Installation und Ausführung - aufwändig
Performance	+ maschinennah auf *CPU*	+ maschinennah auf *CPU*
Sicherheit	+ unterliegt *Android*-Rechtesystem + Transparent	- umgeht *Android*-Sicherheitssystem - keine Tranparenz
Distribution	+ komfortable über *App Stores* oder als *APK* + unkompliziert und Benutzerfreundlich	- *Root* erforderlich - aufwändig und technisches Wissen notwendig

Tabelle 6.1: Gegenüberstellung der Vor- und Nachteilen bei der Verwendung von *dynamischen Bibliotheken* und *Executables*

Es zeigt sich deutlich, dass die Verwendung einer *Java*-Anwendungshülle komfortabler und sicherer ist und einen größeren Funktionsumfang bietet. Die *Java*-Anwendungshülle erforderte zwar Entwicklungszeit. Sie kann jetzt jedoch dazu verwendet werden, um verschiedene *C/C++*-Programm wie den *SEC-Client* oder das *Rate Measurement Framework* in ihr einzubetten. Zukünftig lassen sich somit weitere *C/C++* Programm in ihr einfach integrieren.

Kapitel 7

Messungen und Auswertung

In diesem Kapitel werden die Ergebnisse einiger Messungen vorgestellt, die durchgeführt wurden, um die Laufzeiteigenschaften der entwickelten *Android*-Anwendungshülle zu untersuchen. In den folgenden Abschnitten werden Eigenschaften der verwendeten Testgeräte und die durchgeführten Messverfahren näher erläutert.

7.1 Testgeräte

Als Testgeräte wurden die *Samsung* Smartphones *Galaxy Nexus* und *Galaxy SII (i9100)* verwendet. Eine tabellarische Übersicht der technischen Eigenschaften ist in folgender Tablle 7.1 zu finden:

7.2 Interprozesskommunikation

Über die implementierte *Interprozessbrücke* zwischen den *Services* der Anwendungshüllen kann der *Android SEC-Client* Daten mit einem beliebigen (Mess-)Programm austauschen. Dabei müssen Informationen, die vom *C/C++*-Code des *SEC-Clienten* zur *C/C++*-Seite der anderen Anwendung transportiert werden sollen, mehrere Zwischen-

	Samsung Galaxy Nexus	Samsung Galaxy SII
Prozessorbezeichnung	ARM Cortex-A9, Texas Instruments OMAP 4460 CP	Samsung-Intrinsity Exynos S5PV310
Prozessortaktfrequenz	1,5 GHz Dual-Core-CPU	1,2 GHz Dual-Core-CPU
Arbeitsspeicher	1 GB	1GB
Interner Speicher	16 GB	16 GB
Betriebssystem	Android Jelly Bean (Version 4.3.0)	Android Jelly Bean (Version 4.1.2)
Mobilfunknetze	Pentaband-UMTS (850, 900, 1700, 1900, 2100 MHz) mit HSDPA und HSUPA, Quadband-GSM	GPRS, EDGE, UMTS, HSPA+ (bis zu 21.6 Mbit/s)
Funkverbindungen	Dual-Band-WLAN IEEE 802.11 a/b/g/n, Wi-Fi Direct, Bluetooth 4.0, Near Field Communication, A-GPS	Dual-Band-WLAN IEEE 802.11 a/n, Wi-Fi Direct, Bluetooth 3.0, A-GPS

Tabelle 7.1: Technische Daten des verwendeten *Android*-Testgeräte

stationen passieren. Zunächst müssen die Daten über die *JNI*-Schnittstelle an die *Java*-Seite tranferiert werden. Diese werden für den Transport über die *Interprozessbrücke* in einem *Bundle*-Objekt gepackt und versendet. Im Ziel-*Service* angekommen werden die Daten wieder entpackt und über die *JNI*-Schnittstelle zur *C/C++*-Seite übertragen.
Da bei Messungen das Timing ein wichtiger Faktor ist, darf die Übertragungszeit keinen großen Einfluss auf ein durchgeführtes Experiment haben. Der Transfer von Daten von einer *C/C++*-Seite benötigt dabei mehr Zeit, da hier die meisten Zwischenstationen passiert werden müssen.

Für die Messung wurden in den Anwendungshüllen auf der *C/C++*-Seite jeweils ein *POSIX-Thread* erstellt. Einer der *Threads* sendet dabei periodisch im Intervall von zehn Sekunden eine mit einem Zeitstempel versehene Nachricht. an den anderen, worauf dieser eine Antwort zurück schickt. Um die benötigte Transferzeit zu messen wurde die *C++*-Klasse *TransTimeLogger* entwickelt. Diese versieht die Daten beim Versenden aus der *C/C++*- Seite mit einem Zeistempel. Am Ziel angekommen, berechnet *TransTime-*

Logger aus diesem die Transportdauer und protokolliert diese in einer *Log*-Datei. Dabei ist es wichtig dass *TransTimeLogger* in beiden Anwendungen dieselbe Uhr als Referenz verwenden. Die *Android*-Systemuhr ist für präzise Messungen ungeeignet, da bei einer Synchronisation der Uhrzeit während einer Messreihe die Werte unbrauchbar werden. *TranTimeLogger* verwendet die *clock_gettime()*-Funktion der *POSIX-API*, wobei als verwendete Uhr im Parameter *CLOCK_MONOTONIC* verwendet wird. Diese Uhr startet zu einem systemabhängigen Zeitpunkt (Meistens beim Systemstart) und unterliegt keiner Synchronisierung während der Laufzeit. *clock_gettime()* liefert Messwerte mit einer Genauigkeit im Nanosekunden-Bereich.

Die Messungen wurden zwischen den Anwendungshüllen durchgeführt, die als *Container* für den *SEC-Clienten* und den *RMF* dienten. Die Bezeichnungen in den folgenden Grafiken beziehen sich daher nur auf die Anwendungshüllen, und nicht auf die eigentlichen Programme selbst. Bei der Messungen selbst wurden für beide Testgeräte durchgeführt. Die Anwendungshüllen wird dabei 90 Minuten auf einem Gerät ausgeführt, wobei im Intervall von 5 Sekunden, die *SEC-Client*- Anwendungshülle eine Nachricht (*Ping*) an die *RMF*-Anwendungshülle über die Interprozessbrücke geschickt wird. Die *RMF*-Hülle antwortet daraufhin sofort mit einem Nachricht (*Pong*). Insgesamt werden daher 1080 Nachrichten jeweils in eine Richtung versendet. Als Hinrichtung bezeichnen wir in den nächsten Unterabschnitten die Nachrichtenversand von der *SEC-Client*-Anwendunghülle zur *RMF*-Anwendunghülle.

7.2.1 Galaxy Nexus

In Abbildung 7.1 sind die Transferzeiten zwischen den Anwendungshüllen bei der Ausführung auf dem *Galaxy Nexus* zu erkennen. Die Maximalzeit zur Übertragung einer Nachricht beträgt dabei ungefährt 11 Millisekunden. Ingesamt liegen die meisten Werte zwischen 0,7 und 3 ms. Auffällig sind die Schwankungen, die bei der Rückrichtung stärker sind, als bei der Hinrichtung. Die *Peeks* fallen hier deutlich größer aus.

Die gemessen Werte wurden als kumulative Verteilungsfunktion (*CDF*) in Abbildung 7.2 aufgetragen. Am steilen Kurvenverlauf lässt sich gut erkennen, dass die meisten Werte bei der Hinrichtung im Bereich 1,1 und 1,6 ms liegen. Bei der Rückrichtung sind zwei

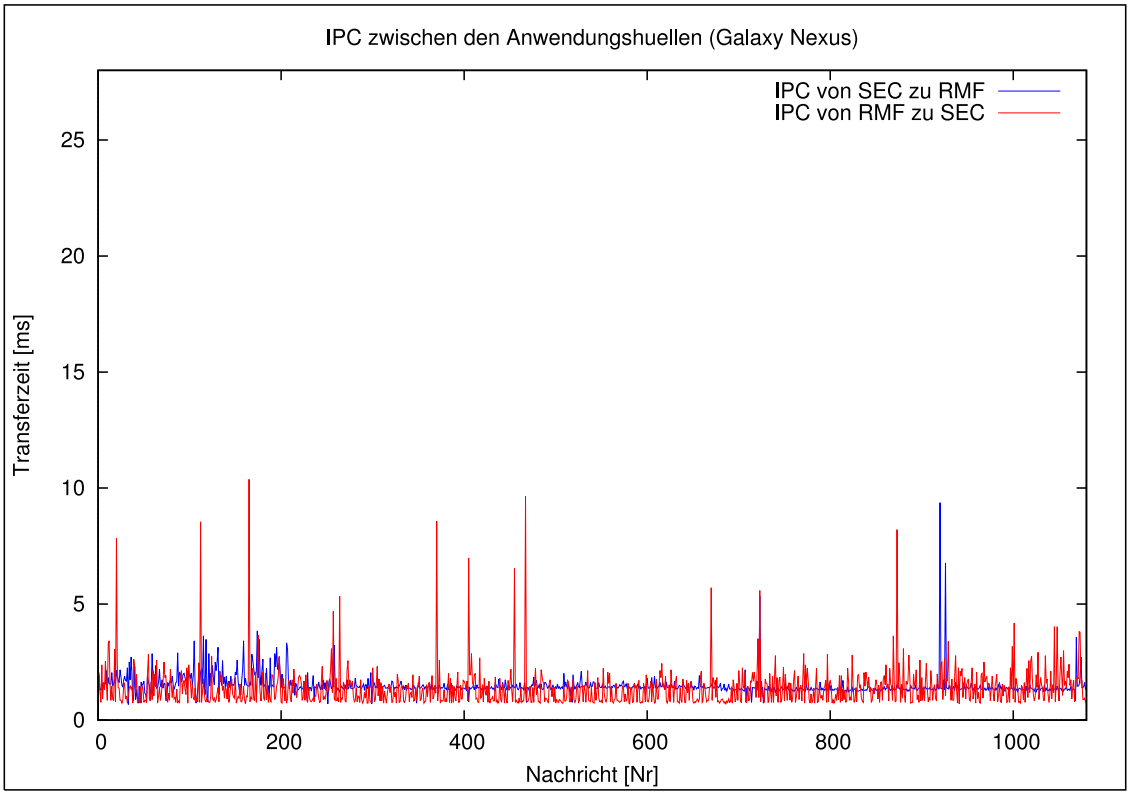

Abbildung 7.1: Interprozesskommunikation zwischen den Anwendungshüllen auf dem *Galaxy Nexus*

steile Kurvenabschnitte zu erkennen, die durch die stärkeren Schwankungen in der Transferzeit verursacht werden.

7.2.2 Galaxy SII

Auf dem *Galaxy SII* zeigt sich ein ähnliches Bild (siehe Abbildung 7.3) wie auf dem *Galaxy Nexus*. Die Transferzeiten liegen meisten in einem kleinem Intervall (0,8 - 3 ms), wobei die Schwankungen bei der Rückrichtung stärker sind als bei der Hinrichtung. Auffällig ist jedoch der Anstieg der durchschnittlichen Transferzeit zum Ende hin. Da es sich bei dem *Galaxy SII* um ein privat genutztes Smartphone handelt, sind mehr Anwendungen und Dienste installiert, die die Schwankungen verursacht haben könnten.

Auch in der *CDF*-Grafik (Abbildung 7.4) zeigt sich am steilen Kurvenverlauf, dass die

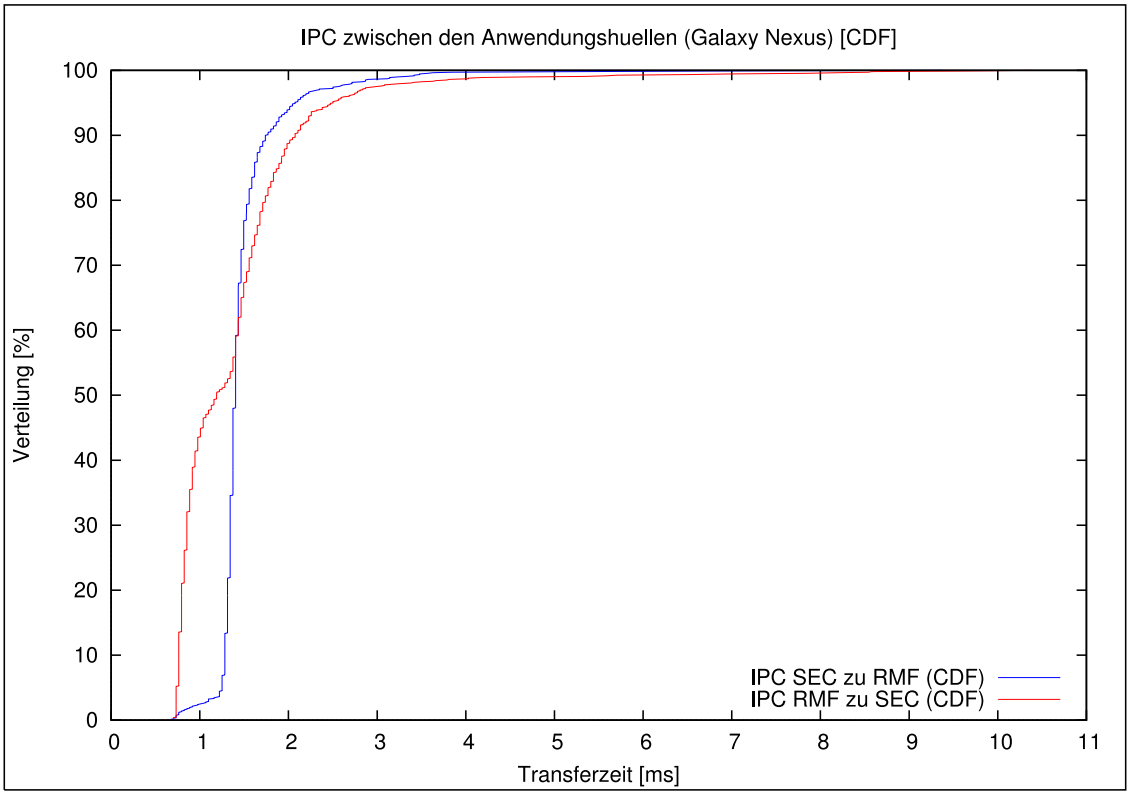

Abbildung 7.2: Interprozesskommunikation zwischen den Anwendungshüllen auf dem *Galaxy Nexus* (CDF)

meisten Werte sich relativ stabil in einem kleinen Bereich bewegen. Aufgrund der stärkeren Schwankungen bei der Rückrichtung ist dort der steile Kurvenabschnitt kürzer.

7.3 Starten der Anwendungshülle

Ein weiterer Aspekt der untersucht wurde, ist die Zeit, die benötigt wird um eine Anwendungshülle durch den *SEC-Clienten* zu starten. Da das eigentliche (Mess-)Programm in die *Java*-Hülle intergriert ist, muss dieser *Overhead* bei zeitgesteuerten Experimenten berücksichtigt werden. Ähnlich dem Messverfahren im vorherigen Abschnitt, wird ein Programmstart-Kommando vom *SEC-Client* mit einem Zeitstempel versehen, welcher im *Intent* zum Starten der *Anwendungshülle* mit versendet wird. Sobald in der Anwendungshülle die native Seite durch die *nNativeInitialize()*- Funktion (siehe Abschnit 5.3.1)

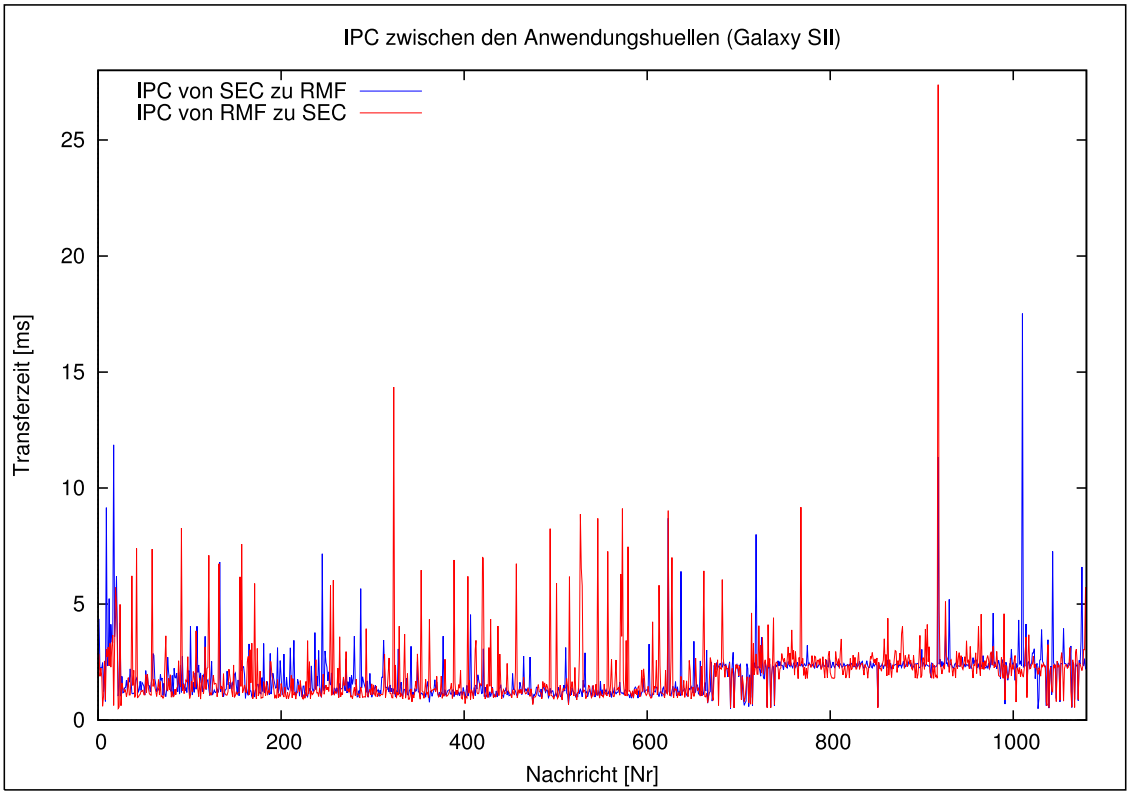

Abbildung 7.3: Interprozesskommunikation zwischen den Anwendungshüllen auf dem *Galaxy SII*

initialisiert wurde, wird die insgesamt verstrichene Zeit seit dem *SEC-Client*-Kommando protokolliert.

In Abbilung 7.5 sind dabei die Messwerte jeweils für das *Galaxy Nexus* und *Galaxy SII* eingetragen. Es sind unregelmäßige Schwankungen zwischen den Startvorgängen zu sehen, wobei das *Galaxy SII* deutlich mehr Zeit benötigt. Die könnte man zum einem auf die Vielzahl installierten Anwendungen und Dienste zurückführen. Auch die stärkere *CPU* des *Galaxy Nexus* könnte ein Grund für die schnelleren Startvorgänge sein.

In der *CDF*-Grafik zu den Messwerten (siehe Abbildung 7.6) machen sich die Schwankungen durch den flachen Kurvenanstieg bemerkbar. Es ist deutlich zu sehen, dass das *Galaxy Nexus* schneller als das *Galaxy SII* arbeitet. Die Werte liegen hier meisten im Bereich von 10 und 220 Millisekunden, während beim *SII* die Werte zwischen 30 und 360 Millisekunden liegen.

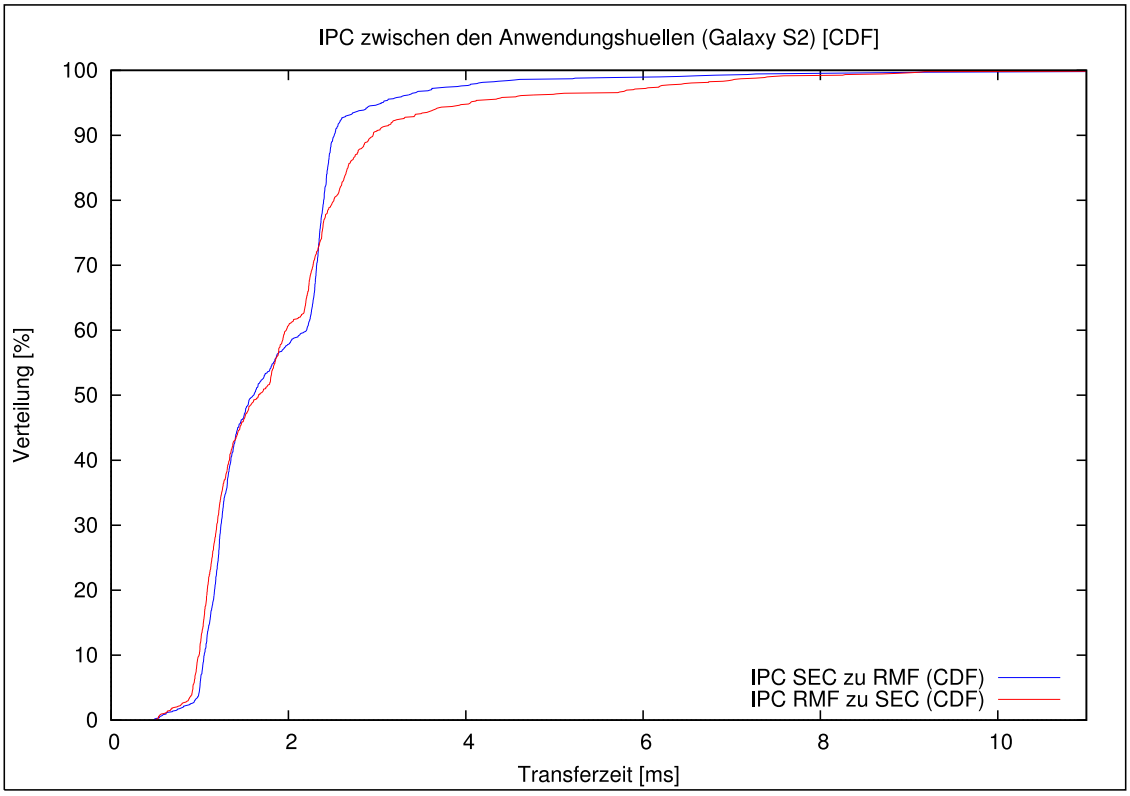

Abbildung 7.4: Interprozesskommunikation zwischen den Anwendungshüllen auf dem *Galaxy Nexus* (CDF)

7.3.1 Auswertung

Bei der Interprozesskommunikation zeigt sich anhand der *CDF*-Grafik deutlich, dass die Transferraten sich vorwiegend einem relativen kleinen Bereich bewegen. Die Rückrichtung unterliegt zwar stärkeren Schwankungen, jedoch sind die benötigten Transferzeit als relativ schnell zu bewerten (im Schnitt 3 Millisekunden) unter der Berücksichtigung, dass neben den Vorgängen im Betriebssystem zur Datenübermittlung zwischen zwei Prozessen, noch *JNI*-Operationen durchgeführt werden müssen um von einer *C++*-Seite zur anderen zu kommunizieren.

Bei der benötigten Zeit zum Starten einer *Android*-Applikation durch den *SEC-Client* zeigt sich in der *CDF*-Grafik durch die flachere Kurvenanstieg, dass die benötigten Startzeiten breiter gefächert sind. Es zeigen sich deutliche Unterschiede an beiden Geräten, die man auf die unterschiedliche Hardware und Belastung durch andere Prozesse zu-

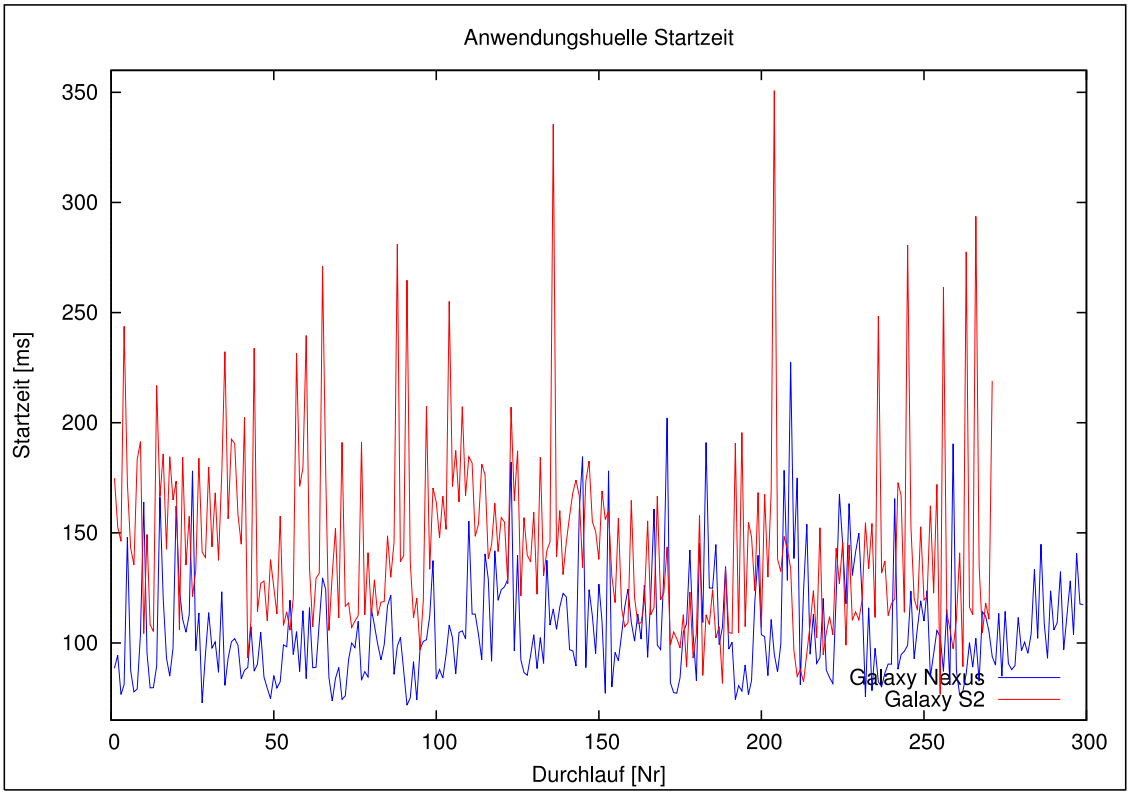

Abbildung 7.5: Benötigte Zeit zum Starten der Anwendungshülle auf beiden Testgeräten

rückführen kann. Mit einem Maximalwert von ungefähr 360 Millisekunden für die Initialisierung der entwickelten Anwendungshülle ist das *Forking* des *Zygote*-Prozesses als schneller Vorgang zu werten.

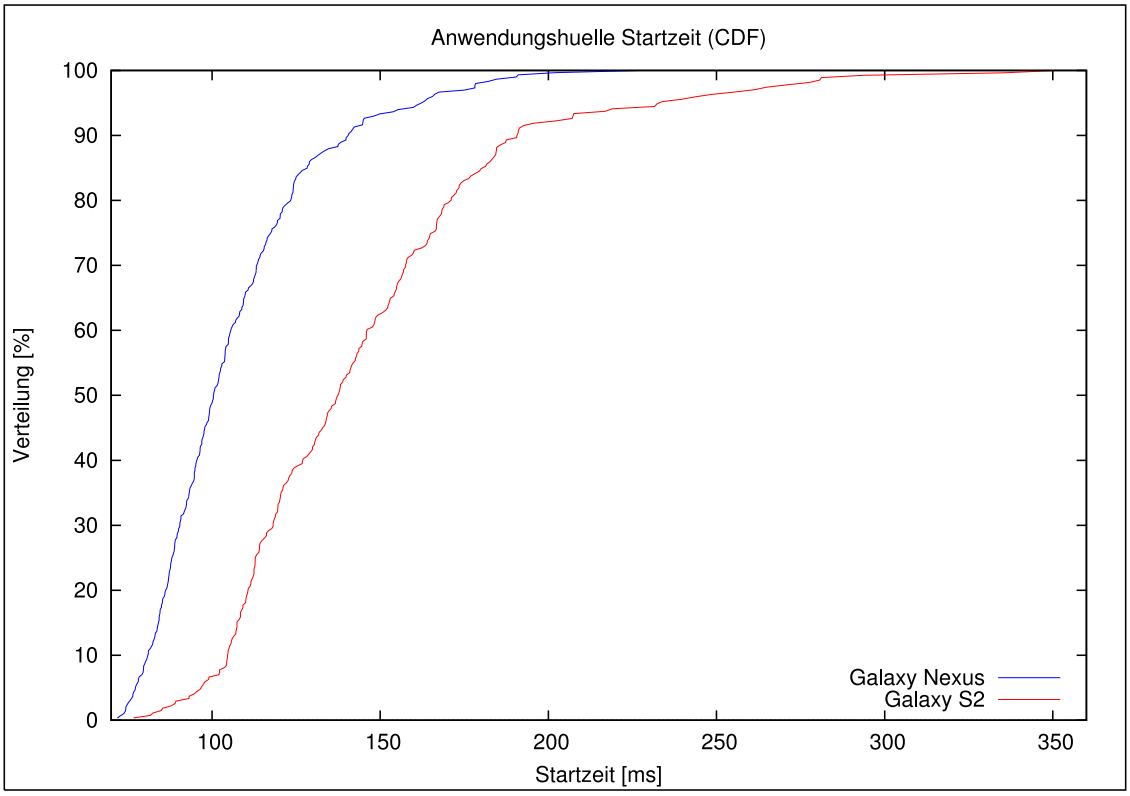

Abbildung 7.6: Benötigte Zeit zum Starten der Anwendungshülle auf beiden Testgeräten (CDF)

Kapitel 8

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde die *Client*-Komponente des *Stateful Experiment Control* als *Android*-Anwendung umgesetzt. Bezüglich der Portierung mussten anhand der Anforderungen des *SEC-Clienten* Designentscheidungen getroffen werden. *Android* ist in erster Linie eine *Java*-Plattform, bietet jedoch mit dem *NDK* die Möglichkeit, nativen Code auszuführen. Der in *C++* geschriebene *SEC-Client* ließ daher die Option offen, den Quelltext in *Java* neuzuschreiben oder das *NDK* zu verwenden. Nach Abwägung aller Vor- und Nachteilen fiel die Entscheidung auf den Einsatz des *NDK*.

Nativer Code kann dabei als *dynamische Bibliothek* in *Android*-Anwendungen eingebunden werden oder direkt als ausführbares Programm im *ELF*-Format erstellt werden. Bei Letzerem handelt es sich jedoch um keine echte *Android*-Anwendung im eigentlichen Sinne, da sie die *Android*-Anwendungsschicht umgeht und direkt auf dem *Linux-Kernel* ausgeführt wird. Neben besonderen Anstrengungen, die zur Ausführung unternommen werden müssen, bergen solche Anwendungen Sicherheitsrisiken und können nicht auf Funktionalitäten des *Android-Applikation-Frameworks* zugreifen.

Der *SEC-Client* wurde daher als *dynamische Bibliothek* entwickelt, um in einer *Java*-Anwendunghülle als echte *Android*-Applikation ausgeführt werden zu können. Anhand der Anforderungen, die der *SEC*-Client an die *Android*-Laufzeitumgebung stellt, wurden verschiedene Möglichkeiten zur Umsetzung diskutiert und schließlich ein Anwendungsdesign erstellt. Auf Basis dieses Entwurfes wurde die *Anwendungshülle* implementiert, die eine *JNI*-Schnittstelle zur Kommunikation zwischen *Java* und *C/C++*-Code ermög-

licht. Vor der Integration des *SEC-Clienten* in die *Java*-Hülle, mussten einige Hürden überwunden werden, um die Anwendung zu stabilieren, so dass diese die gewünschten Laufzeiteigenschaften vorwies.

Für die Kommunikation zwischen dem *C++*- Code des *SEC-Clienten* und der *Java*-Hülle mussten Anpassungen im Quelltext vorgenommen werden. Die dazu entwickelt *C++* Klasse *OSHelper* sorgt dabei für einen plattformunabhängien, transparenten Code, der sich für *Linux*- und *Android*-Geräte kompilieren lässt, ohne vorher Änderungen vornehmen zu müssen. Des Weiteren mussten einige Änderungen für die Verwendbarbeit mit *Android*-Applikationen am Experimentbeschreibungsprotokoll vorgenommen werden. Auch diese wurden plattformunabhäng gestaltet, so dass bei der Ausführung eines Experiments das verwendete *Client*-Betriebssytem keine Rolle spielt.

Analog dazu wurde das bereits für Android portierte *RMF* in die entwickelte Anwendungshülle eingesetzt und konnte erfolgreich mit dem *Android SEC-Clienten* getestet werden.

Der *Overhead*, den die Anwendungshülle verursacht, ist bei der Verwendung von Experimenten, bei den die automatisierte Steurung im Vordergrund steht, als vernachlässigbar klein zu werten.

8.1 Ausblick

Das stetig weiterentwickelte *Application-Framework* der *Android*-Plattform bietet eine Unmenge an Funktionen, die sich sinnvoll mit *SEC* verwenden lassen. Beispielsweise wurde in Rahmen dieser Arbeit eine *SMS*-Funktion implementiert, die Statusmeldungen über den Versuchsablauf versendet.

Für Experimente, in denen das Timing wichtig ist, in welchen Intervall-Abständen das *RMF* hintereinander vom *SEC-Client* gestartet werden sollen, könnten solche Messungen helfen, die benötigte Programmladezeit bei der Berechnung der nächsten Ausführungszeit zu berücksichtigen.

Literaturverzeichnis

[Amf12] AMFT, Tobias: *Eine zustandsbasierte Experimentsteuerung über einen fehleranfälligen Kommunikationskanal*. Juni 2012.

[Bra13] BRAIN MCFLY: *Partitionen im Androidsystem*. Website, November 2013. Online verfügbar unter http://www.brutzelstube.de/2012/partitionen-im-androidsystem-boot-system-recovery-data-cache-misc/

[C++13a] C++0X/C++11 SUPPORT IN CLANG: *GPS*. Website, November 2013. Online verfügbar unter http://clang.llvm.org/cxx_status.html

[C++13b] C++0X/C++11 SUPPORT IN GCC: *GCC C++11 Features*. Website, November 2013. Online verfügbar unter http://gcc.gnu.org/projects/cxx0x.html

[Cin12] CINAR, Onur: *Android Apps with Eclipse* -. 1. Aufl. New York : Apress, 2012. ISBN 978–1–430–24435–6

[Dav13] DAVE BORT: *Android is now available as open source*. Website, November 2013. Online verfügbar unter http://web.archive.org/web/20090228170042/http://source.android.com/posts/opensource

[Goo13] GOOGLE: *Dashboards - Android Developer*. Website, November 2013. Online verfügbar unter http://developer.android.com/about/dashboards/index.html

[Hei13] HEISE: *Marktforscher: Windows Phone explodiert*. Website, November 2013. Online verfügbar unter http://heise.de/-1931586

[Off13a] OFFICIAL CRYSTAX NDK PROJECT WEBSITE: *CrystaX NDK*. Website, November 2013. Online verfügbar unter http://www.crystax.net/en/android/ndk

[Off13b] OFFICIAL GOOGLE NDK WEBSITE: *Google NDK*. Website, November 2013. Online verfügbar unter http://developer.android.com/tools/sdk/ndk/index.html

[Olf13] OLFEN, Malte: *Ende-zu-Ende-Messungen von Mobilfunkparametern mit Smartphones*. Juni 2013.

[Ope13] OPEN HANDSET ALLIANCE: *OHA*. Website, November 2013. Online verfügbar unter http://www.openhandsetalliance.com/

[Prz13] PRZEMYSLAW SZYMANSKI: *Eric Schmidt: Bald mehr als eine Milliarde Android-Geräte - Computer Base*. Website, November 2013. Online verfügbar unter http://www.computerbase.de/news/2013-04/eric-schmidt-bald-mehr-als-eine-milliarde-android-geraete/

[Rat12] RATABOUIL, Sylvain: *Android NDK - Discover the Native Side of Android and Inject the Power of C/C++ in Your Applications : Beginner's Guide*. Birmingham : Packt Publishing Ltd, 2012. ISBN 978–1–849–69153–6

[Rob13] ROBIN DAVIES: *Foreground Service Being Killed By Android*. Website, November 2013. Online verfügbar unter http://stackoverflow.com/questions/6645193/foreground-service-being-killed-by-android

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 04.Dezember 2013 Daniel Sathees Elmo

Diese CD enthält:

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- **SEC** und **RMF** Quelldateien der im Rahmen der Bachelorarbeit erstellten Android Anwendungshüllen
- **CSV-Dateien** den zur Auswertung verwendeten Datensatz
- die Websites der verwendeten Internetquellen