Bachelorarbeit

Vitali Eylin

Konzeption und Entwicklung einer Post-Disaster-Analyse
für Linux mit NFC und Android

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Vitali Eylin

Konzeption und Entwicklung einer Post-Disaster-Analyse für Linux mit NFC und Android

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Thomas Lehmann

Abgegeben am 10.05.2012

Vitali Eylin

Thema der Bachelorarbeit
Konzeption und Entwicklung einer Post-Disaster-Analyse für Linux mit NFC und Android

Stichworte
Blackbox, Linux, Java

Kurzzusammenfassung
Konzeption und Entwicklung einer Post-Disaster-Analyse für Linux mithilfe eines Android-Smartphone und NFC Technik

Vitali Eylin

Title of the paper
Design and development of post-disaster analysis for Linux, NFC and Android

Keywords
NFC, Android, Blackbox, Linux, Java

Abstract
Design and development of post-disaster analysis for Linux using an Android smartphone and NFC technology

Inhaltsverzeichnis

1 Einführung **1**

2 Grundlagen **2**
2.1 Anwendungsentwicklung unter Android . . . 2
2.1.1 Allgemeine Information . . . 2
2.1.2 Android-Architektur . . . 3
2.1.3 Besonderheiten der Entwicklung für Android . . . 5
2.2 Near Field Communication . . . 5
2.2.1 Allgemeine Information . . . 5
2.2.2 Use Cases . . . 6
2.2.3 Sicherheit der Kommunikation . . . 7
2.3 STMicro M24LR64-R dual interface EEPROM . . . 7
2.4 Analyse der Logfiles . . . 8
2.4.1 Einführung in Log-Analytik . . . 8
2.4.2 Inhalt der Logs . . . 8
2.4.3 Logging-Level . . . 9
2.4.4 Schwierigkeiten bei der Auswertung . . . 9

3 Analyse **11**
3.1 Vision . . . 11
3.2 Existierende Lösungen . . . 11
3.3 Funktionale Anforderungen . . . 13
3.4 Nicht-funktionale Anforderungen . . . 13
3.5 Technische Voraussetzungen . . . 14
3.6 Anwendungsfälle . . . 15
3.6.1 Linux System ist in Betrieb . . . 15
3.6.2 Linux System ist defekt oder ausgeschaltet . . . 16

4 Entwurf & Architektur **19**
4.1 Architektur . . . 19
4.1.1 Entscheidungsgrundlage bei der Modellierung . . . 19

4.1.2 Kontextsicht . . . 20
4.1.3 Bausteinsicht . . . 22
4.1.4 Laufzeitsicht . . . 24
4.1.5 Verteilungssicht . . . 27
4.2 Design . . . 29
4.2.1 Motivation für Einsatz . . . 29
4.2.2 Observe Patter . . . 30
4.3 Vorgehen bei der Entwicklung . . . 30
4.3.1 Entscheidungsgrundlage für iterative Entwicklung . . . 31
4.3.2 Iterative Vorgehensweise . . . 31

5 Realisierung **33**
5.1 Implementierung . . . 33
5.1.1 Entwicklungswerkezeuge . . . 33
5.1.2 Herangehensweise . . . 34
5.1.3 JNotify . . . 36
5.1.4 Testing . . . 36
5.2 Methoden / Besonderheiten der Realisierung . . . 37
5.2.1 Schnittstellen der BlackBox . . . 37
5.2.2 Behandlung der Persistenz / EEPROM . . . 38
5.2.3 Auseinandersetzung mit Polling . . . 39
5.2.4 Schwierigkeiten bei der Implementierung . . . 40
5.3 Ausblick auf mögliche Erweiterungen . . . 41
5.3.1 Portierung auf ein anderes Betriebssystem . . . 42
5.3.2 Erweiterung der Reader-App . . . 42
5.3.3 Überwachung von anderen System-Parameter . . . 42

6 Abschluss **43**
6.1 Zusammenfassung . . . 43
6.2 Ausblick . . . 43

A Anhang **45**
A.1 CD . . . 45

Literaturverzeichnis **46**

Kapitel 1

Einführung

Man stelle sich ein folgendes Szenario vor: Ein wichtiger Computer, vorzugsweise ein Server, erleidet einen Datenverlust oder Totalausfall. In schlimmsten Fall ist die Festplatte unwiderruflich zerstört oder gelöscht. Für solche Fälle kann man mit regelmäßigen Sicherungen des Datenbestandes vorsorgen und die Folgen von solchen Ausfall lindern. Zum Schluss bleibt eine Frage bei einem unwiderruflichen Datenverlust inklusive wichtigen Systemlogs bestehen: was war die Ursache hierfür?

Es gibt sehr viele mögliche Szenarien, die zu einem solchen Desaster führen können: Defekt durch Überhitzung, Virusbefall, sehr grobe Fehlbedingung durch den Benutzer, etc. Bei einem privaten Rechner wird solch ein Ausfall nicht unbedingt Schaden in Millionenhöhe anrichten, bei mittel bzw. großen Unternehmen sieht es schon anders aus. Daher ist es für die Verantwortlichen im IT Bereich wichtig sehr zeitnah die Ursache für o.g. Ausfall zu identifizieren und in nahe Zukunft den Wiedereintritt vorzubeugen.

Das Ziel der Arbeit ist Entwicklung, Implementierung und Dokumentation einer NFC[1] basierten „Blackbox“ für den PC mit Linux OS. Die Entwicklung und Implementierung umfasst Entwicklung eines Linux Deamons um regelmäßig Systemlogs auf dem NFC-Tag[2] wegzusichern, sowie eine Android Applikation um die Log Daten von NFC-Tag mit einem NFC fähigen Android Smartphone auszulesen und darzustellen.

[1] Near Field Communication, kurz NFC ist ein internationaler Standard zur kontaktlose Übertragung der Daten auf sehr kurzen Strecken (bis 4 cm)

[2] NFC-Tag: ein Datenspeicher, dass mit NFC Fähigen Gerät gelesen und/ oder beschrieben werden kann. In diesen Fall ist eine per USB angeschlossene NFC Leiterplatine von STmicro.

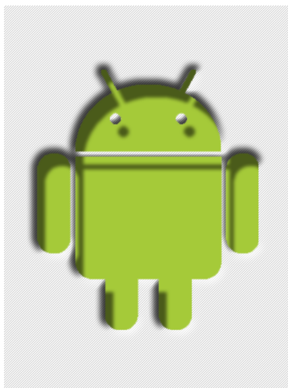
Kapitel 2

Grundlagen

2.1 Anwendungsentwicklung unter Android

In diesen Abschnitt wird das Betriebssystem namens Android vorgestellt. Folgend werden die Vor- und Nachteile des Systems aus dem Sicht der Anwendungsentwickler genauer aufgelistet. In späteren Verlauf wird die Android-Architektur mit ihren Eigenheiten beschrieben. Am Schluss werden die Besonderheiten der Anwendungsentwicklung unter Android-OS unter der Lupe genommen. Damit erhält man erstmals einen groben Überblick über Android OS. Die meisten feinen technischen Details bleiben für spätere Kapitel 4-5 vorenthaltet.

2.1.1 Allgemeine Information



Android ist ein freies und quellenoffenes Betriebssystem für mobile Geräte. Es wurde von Open Handset Alliance (OHA) unter der Leitung von Google entwickelt. Mittlerweile laufen die meisten weltweit verkaufte Smartphones[3] mit einem Android Betriebssystem. Dies stellt eine beachtliche Leistung für eine Betriebssystem, das erst seit 2008 eingeführt wurde.

Die zwei wichtigsten Vorteile von Android sind die Quellenoffenheit und kostenlose Verfügbarkeit. Jeder kann den Sourcecode von Android runterladen und selbst kompilieren. Anderseits verlangt Open Handset Alliance keine Lizenzgebühren für das nackte Android Betriebssystem.[4] Die OHA Mitglie-

[3] *heise.de Open Source. Jörg Wirtgen.* `http://www.heise.de/open/meldung/Smartphones-Android-ueberholt-Symbian-Apple-verliert-Marktanteile-1180547.html` *(Stand: 31.01.2011)*

[4] Android Licenses. Google Inc. `http://source.android.com/source/licenses.html` (Stand: 12.12.2011)

der verkaufen mehrheitlich Smartphones und Tablet PCs mit Android OS. Somit haben sie ein starkes Interesse an der Entwicklung und Fortführung der Android OS[5].

Die meisten Benutzer assoziieren Android mit einem Smartphone. Die möglichen Einsatzzwecken sind jedoch viel breiter. Es gibt mittlerweile eine breite Palette an Geräten, die Android als Produktgrundlage einsetzen. Beispiele: Autoradios, Netbooks, Smartphones, Tablet-PCs, Satelliten-Receiver usw. Es ist gut möglich, dass in der Zukunft Android OS auch unseren Fernseher und die Waschmaschine[6] steuert.

Für Entwickler hat Android mehrere Vorteile:

- große Reichweite an potenziellen Zielgeräten[7]
- kostenlose Android SDK
- gute Dokumentation der API
- sehr lebendige Community rund um die Entwicklung für Android
- Open Source[8]
- geringe Einstiegshürde für Java/C++ Entwickler in die mobile Programmierung

Folgende Nachteile dürfen auch nicht verschwiegen werden:

- starke Fragmentierung der Hard- und Software[9]
- Arbeit mit kryptischen xml Dateien ist manchmal unumgänglich[10]

2.1.2 Android-Architektur

Android basiert auf Linux Kernel 2.6. Das Betriebssystem ist für die meisten Mikrokontroller verfügbar. Hierzu zählen ARM, MIPS sowie X86 Prozessorarchitekturen.

Der Basis von Android bildet Dalvik Virtual Machine (DVM) ab. Es ist eine virtuelle Maschine. Jede Anwendung wird in einer eigenen DVM ausgeführt. Dieses Konzept ist nicht wirklich performant. Es hat aber viele Vorteile bezüglich Sicherheit und Stabilität. Da jede Anwendung in einer eigenen virtuelle Maschine läuft, wird es im Falle eines Absturzes nicht

[5] FAQ von OHA. `http://www.openhandsetalliance.com/oha_faq.html` (Stand 12.12.2011)

[6] silicon.de - Martin Schindler. Android erobert Smartphones, Netbooks und Tablet-PCs `http://www.silicon.de/technologie/mobile/0,39044013,41552866,00/android_4_0_steuert_selbst_waschmaschinen.htm` (Stand: 11.5.2011)

[7] (Becker und Pant, 2010, Vorwort)

[8] Sourcecode: `http://source.android.com/source/downloading.html`

[9] (Skogberg, 2010, Kap. Android SDK)

[10] (Becker und Pant, 2010, 1.3 Layout definieren)

die anderen Anwendungen gefährden. Die Sicherheit wird durch die Trennung der Speicherräume gewährleistet, d.h. die Anwendungen haben keinen Zugriff auf den gesamten Systemspeicher, sondern nur auf den für sie reservierten virtuellen Bereich.

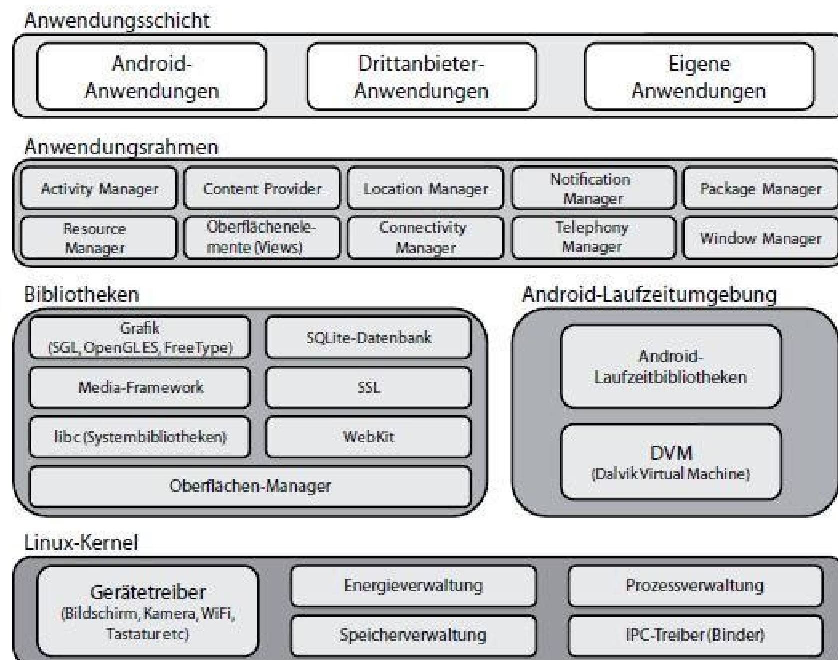


Abbildung 2.1: Architektur von Android
Quelle: Becker und Pant (2010)

Die Abbildung 2.1 zeigt wie DVM von dem System abgegrenzt ist. Die in der Android Laufzeitumgebung laufenden Anwendungen sind nicht in der Lage direkt auf die Hardware oder Systembestandteile zuzugreifen. Falls eine Anwendung eine bestimmte Funktionalität benötigt, muss es hierfür die von Android vorgegebene Application programming interface (API) nutzen. Diese Vorgehensweise hat mehrere Vorteile für Systemstabilität und Sicherheit.

- Das Betriebssystem behält die Kontrolle über die Hard- und Software Ressourcen
- Der Benutzer wird vor der Installation informiert auf welche Systembestandteil die Anwendung zugreifen möchte
- Android kann bestimmte Schnittstellen sperren, z.B. das Internetzugriff per Mobilfunk, um beispielsweise Energie zu sparen

2.1.3 Besonderheiten der Entwicklung für Android

Die Entwicklung einer Android-Anwendung erfolgt mit folgenden Werkzeugen: Android SDK, Eclipse IDE und Android-Plugin für Eclipse. Das Android SDK stellt das jeweilige Zielsystem zur Verfügung einschließlich entsprechenden APIs. Um ein Programm auszuführen, lädt man es mit Eclipse Plugin aufs Zielgerät hoch oder führt es im Android Virtual Device Manager aus (AVD).

Die Anwendungen werden in der Regel seitenweise aufgebaut. Jede, für den Benutzer, sichtbare Seite stellt eine s.g. Activity dar. Ein Activity ist für die Interaktion mit dem Benutzer zuständig. Alle Activitys müssen den System über Manifest-Datei bekannt gegeben. Eine Activity, die nicht in Manifest inkl. aller benötigten Berechtigungen eingetragen ist, wird von dem System nicht ausgeführt.

Die Activitys werden von Intents aufgerufen. Ein Intent heißt übersetzt „Absichtserklärung". Die Intents werden vom System gestartet, wenn die Anwendung aufgerufen wird. Sie rufen ihrerseits Activitys auf. Intents werden ebenfalls über die Manifest-Datei definiert. Die Intents werden für das Aufrufen von weiteren Activitys eingesetzt. Wenn eine Activity entweder fertig ist oder eine weitere Activity benötig, dann ruf sie es per Intent auf. Die Kommunikation der Activitys geschieht über die Intent Parameter.

Das Aussehen einer App wird durch die Layouts definiert. Die Layouts kann man entweder über den eingebauten Layout-Editor entwerfen und/oder direkt mit einem Texteditor über XML Dateien anpassen.

2.2 Near Field Communication

In diesen Abschnitt wird Near Fiel Communication (NFC) mit ihren vielfältigen Einsatzzwecken vorgestellt. NFC hat in den letzten Jahren fast unbemerkt einen Weg in viele Geräte und Gegenstände des täglichen Lebens gefunden. Immer mehr Hersteller und Dienstleistungsanbieter setzen NFC für drahtlosen Datenaustausch ein. Tendenz steigend. Wie in den anderen Abschnitten der Grundlagen bleiben die feinen technischen Details dem Leser vorerst erspart. Sie können in den späteren Kapiteln 4-5 nachgeschlagen werden.

2.2.1 Allgemeine Information

Near Field Communication ist ein internationaler Standard für die drahtlose Übertragung der Daten auf kurze Strecken (bis 4cm). NFC wurde im Jahr 2002 von NXP Semiconductors und Sony entwickelt. Beide Unternehmen sind Marktführer bei der Entwicklung von berührungslosen Chipkarten. Mittlerweile wird der Standard von NFC Forum weiterentwickelt. Hierzu gehört eine Reihe von namhaften Herstellern wie z.B. Motorola, Microsoft, NEC, MasterCard, Panasonic usw. Die Aufgabe des NFC Forums ist die Einheitliche Entwicklung der

NFC Technologie. Bei der Weiterentwicklung spielt das Mobiltelefon fast immer eine zentrale Rolle. Mögliche einsatzzwecke wären Mobile Payment, verschiedene Berechtigungskarten, Fahrscheine usw. (sehe Use Cases).

NFC basiert auf bewährten Technologien, beispielsweise RFID und Chipkarte. Bei RFID findet die Kommunikation nur in eine Richtung statt. Es ist immer jemand ein Sender oder Empfänger. Bei NFC kann dagegen jeder ein Sender und Empfänger gleichzeitig sein.

2.2.2 Use Cases

An diese Stelle werden einige Beispiele sowie Einsatzzwecke der NFC vorgestellt.

Die typischen Anwendungsfälle lassen sich gut in 3 Kategorien einteilen:

- papierlose Tickets
- mobiles Bezahlen
- Zutrittskontrolle / Identifizierung

Die ersten 2 Anwendungsfallgruppen haben eine Gemeinsamkeit: sie setzen in der Regel in NFC fähiges Mobiltelefon voraus. Es hat den Vorteil, dass so gut wie jeder ein Mobiltelefon in der Tasche hat und man anderseits nicht noch 3-4 zusätzliche Plastikkarten in der Portmonee einstecken muss.

Bei papierlosen Tickets lädt sich der Benutzer z.B. ein Ticket für das Konzert auf das Mobiltelefon herunter. Beim Eintritt in das Konzert wird das NFC fähiges Handy an einen NFC Leser gehalten um die einmalige Ticket-ID auszulesen und zu entwerten. Nach dem Auslesen wird die Ticket-ID in der Veranstalter-Datenbank entwertet.

Mobiles Bezahlen basiert auf Identifizierungstechnik. Damit wird es sichergestellt, dass kein Fremder Zugriff auf das Geldkonto erhält. Es läuft wie folgend ab: ein Benutzer identifiziert sich mit seinem NFC fähigen Mobiltelefon am Zahlterminal. Der Zahlterminal stellt eine Verbindung zum Bank her um festzustellen ob der Benutzer existiert und sein Konto für Wunschbetrag gedeckt ist. Wenn der Benutzer identifiziert und sein Konto gedeckt ist, wird der Zahlungsvorgang abgeschlossen. Die Ware oder Dienstleistung darf genutzt werden und der Verkäufer erhält das Geld per Lastschrift. Solche Zahl-Systeme gibt es bereits in Fernost im praktischen Einsatz. Aktuell dringt Google als Globalplayer mit Wallet[11] auf weitere Märkte. Somit ist es nur eine Frage der Zeit bis NFC basierte Bezahlsysteme sich in Europa verbreiten.

Bei der Zutrittskontrolle wird ein Mobiltelefon als drahtlose Zugangskarte eingesetzt. Der NFC Leser prüft ob das Gerät mit eine bestimmte ID zutrittsberechtigt ist.

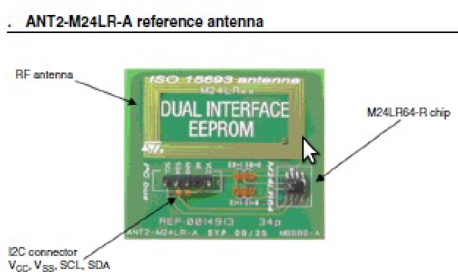
[11] NFC Bezahlsystem von Google

2.2.3 Sicherheit der Kommunikation

Die größte Errungenschaft von NFC gegenüber der RFID ist die Sicherheit des Standards. In der Vergangenheit gab es vielfach berechtigte Kritik aus der Bevölkerung, dass man die RFID Chips missbräuchlich aus einer sehr großen Entfernung auslesen und ggf. manipulieren kann. Daher war die Akzeptanz von RFID aus Sicherheitsgründen relativ gering. NFC stellt physikalisch sicher, dass es in der Praxis unmöglich ist den Chip aus der Ferne auszulesen. Dies wird dadurch erreicht, dass der NFC Sender nach der Spezifikation schon beim Senden der ersten Bits mindestens 30% der Energie verbraucht hat. Somit wird der Chip nach den ersten Bits physikalisch nicht mehr in der Lage sein den vollständigen Sendevorgang über eine lange Strecke durchzuführen.

Der Nachrichtenaustausch basiert auf dem bestehenden Standard ISO/IEC 14443, dieser definiert wie ein Leser kontaktlos von einer Chipkarte die Information auslesen kann.

2.3 STMicro M24LR64-R dual interface EEPROM



Quelle: STMicroelectronics

STMicroelectronics stellt das M24LR64-R Modul her. Bei M24LR64-R handelt es sich um einen Speicherchipkontroller jeweils mit eine Luft- und Kabelschnittstelle. Die Luftschnittstelle heißt RF und wird über RFID realisiert. Die Kabelschnittstelle I2C lässt sich mit einem Adapter auf klassischen USB umpatchen und demensprechend anschließen. Der Kontroller kann 4 verschiedene Betriebszustände einnehmen. Sie ergeben sich aus 2 verschiedenen Zuständen von I2C und RF. Bei der RFID Schnittstelle läuft die Kommunikation über ISO 15693 RFID Protokoll ab. Das Protokoll ist wegen der leicht erhöhten Sendeleistung (bis ca. 1,5 Meter) von der RF Schnittstelle nicht NFC konform. Es ist aber absehbar, dass die ISO 15693 Norm in der nächste Zeit als ein neues zusätzliches NFC Kommunikationsstandard von NFC-Forum aufgenommen wird. Das NFC-Forum beschäftigt sich gerade damit.

Der EEPROM verfügt insgesamt über 64 kBit Speicher. 8192 Bytes davon stehen an der I2C Schnittstelle zur Verfügung. Der Rest ist für RF reserviert. Damit kann der EEPROM mehr als 1 Million Schreibvorgänge vertragen. Das klingt auf dem ersten Blick hoch, ist es aber unter gewissen Umständen nicht. Falls beispielweise jede 10 Sekunden eine Textdatei in dem EEPROM abgelegt werden sollte, wird der millionste Schreibvorgang schon nach ca. 115 Tagen erreicht sein. Wenn man noch berücksichtig, dass beim Wiederbeschreiben von EEPROM zwei Vorgänge stattfinden (löschen und neu schreiben der Sektoren), dann fällt die Haltbarkeit noch kürzer aus. Dazu mehr in Kapitel (5.2.2).

2.4 Analyse der Logfiles

Alle modernen Betriebssysteme bieten in irgendeiner Form eine Möglichkeit an, nachträglich die Systemereignisse nachzuvollziehen. Am weitesten verbreitet ist die textuelle Beschreibung und Abspeicherung der Ereignisse, auch Logging genannt. Dieser Abschnitt beschäftigt sich mit dem Auslesen der Log-Einträge, sowie mit Schwierigkeiten, die dabei auftreten können.

2.4.1 Einführung in Log-Analytik

Jedes modernes Betriebssystem legt in regelmäßigen Abständen relevante Ereignisse textuell ab. Die Logeinträge sind zunächst systemspezifisch, sie enthalten meist sehr spezielle Daten und Statusmeldungen. Diese Informationen ermöglichen dem Anwender oder Administrator die aktuellen Ereignisse mit einem Blick nachzuvollziehen. An der Stelle ist es erwähnenswert, dass es durchaus Sinn macht auch bei einem auf dem ersten Blick lauffähigen System die Logs anzuschauen. Viele Fehler und Ausfälle treten in der Regel nicht unangekündigt auf [12]. Nach einem Ausfall oder Fehlverhalten lässt sich der Hergang der Ausfall anhand der Logs nachvollziehen.

2.4.2 Inhalt der Logs

Die Logs geben die für den gegebenen Logging-Level (2.4.3) relevante Informationen wieder. Die Inhalte der Logs variieren sehr stark je nach ihren Einsatzzweck. Üblicherweise enthält ein Log-Eintrag folgende Information: Datum, Uhrzeit, Host-Adresse, Name der Anwendung, Error-Level und Text.

Beispiel:

- *Mar 8 09:33:30 ve-lnx-nb NetworkManager[1342]: <info> Activation (eth0) Stage 5 of 5 (IP Configure Commit) started...*

Der o.g. Eintrag aus einem Ubuntu-Linux System ist nach *MM-DD HH:MM:SS <HOST> <APP>[PID]: <Log-Level> <Text>* Schema aufgebaut. Wenn ein Netzwerkadministrator nach einem Netzwerk-Fehler sucht, würde er den Log mit einem *Network* Wildcard durchsuchen lassen.

Es gibt leider keine Vorschriften oder Richtlinien wie ein Log aufgebaut sein sollte. Daher kann jeder Entwickler in Eigenregie entscheiden, wie und in welcher Form seine Anwendung Logs generiert, abspeichert bzw. welches Text wird mitausgegeben.

[12] Oliver u. a. (2012, S.56)

2.4.3 Logging-Level

In den Logs taucht nicht selten je nach Situation zu viel oder zu wenig Information. Das Linux liefert von Haus aus ein sehr mächtiges Werkzeug zum Filtern der Logs: feingranular einstellbare Logging-Level[13]. Man unterscheidet folgende Logging-Levels:

Level	**Bezeichnung**
0	Emergency (emerg)
1	Alerts (alert)
2	Critical (crit)
3	Errors (err)
4	Warnings (warn)
5	Notification (notice)
6	Information (info)
7	Debug (debug)

Level 7 schreibt alles sehr feingranular bis auf Variablenwerte in den Logs. So eine Informationsflut führt sehr schnell zu einem riesigen, kaum handhabbaren Datenberg. Man braucht den Level in der Regel nur bei der Einrichtung von Logging-Server oder bei der Entwicklung. Level 0 schreibt dagegen nur die systemkritische Einträge in dem Log, wie z.B. Kernel Panic. Damit sein aber kein effektive Überwachung vom System möglich. In der Praxis würde man die Levels 3-5 verwenden.

Die Logeinträge selbst lassen sich ebenfalls in verschiedene Kategorien einordnen:

Bezeichnung	Zweck
auth	authentication (login) messages
cron	messages from the memory-resident scheduler
daemon	messages from resident daemons
kern	kernel messages
lpr	printer messages (used by JetDirect cards)
mail	messages from Sendmail
user	messages from user-initiated processes/apps
local0-local7	user-defined
syslog	messages from the syslog process itself

Die Local0-7 Kategorie kann von dem Benutzer frei definiert und verwendet werden.

2.4.4 Schwierigkeiten bei der Auswertung

Während des Betriebs eines Rechners fallen in der Regel, je nach Log-Level, sehr große Log-Daten an. Bei der Suche nach einem bestimmten Ereignis in den großen Logs gibt es

[13]Debian `http://www.aboutdebian.com/syslog.htm` (Stand: 20.3.2012)

an einigen Stellen viele potenzielle Probleme. Wenn man die Stolperstellen in Blick behält, kann in der Regel relativ schnell zum Ziel kommen:

- In den meisten Fällen ist es jedoch unklar, nach welchen Kriterien gesucht werden soll

In diesem Fall kann man die letzte Log-Einträge kritisch ins Auge fassen oder je nach Ereignis z.B. Netzwerkausfall nach *NIC* Einträgen suchen.

- Die Logs sind systemspezifisch

Die Einträge enthalten oft verschiedene Parameter, Hardwarebezeichnungen, IPs, MAC-Adressen etc., die nur den jeweiligen Rechner zugeordnet sind. Daher sollte man sich nicht darauf verlassen, dass man an vielen verschiedenen Rechner gleiche oder ähnliche Einträge vorfindet.

- Es finden sich nicht selten anwendungsspezifische Debug-Einträge in den Logs

Die Programmierer setzen auch heute noch sehr oft Printf zum Debuggen ein. Teilweise werden die Debug-Einträge nach Syslog umgeleitet. Hier sollte man aufpassen und filtern.

- No Error Messages erschweren die Suche

Wenn man expliziert nach einem *error* Eintrag sucht, wird man in der Regel auch an die so genannte „no error” messages stoßen. Es handelt sich um Log-Einträge, die das Wort „error” in ungünstigen Kontext benutzen. Die Anwendung versucht den Benutzer mitzuteilen, dass es einwandfrei läuft oder eine bestimmte Aktion fehlerfrei abgeschlossen hat. In diesen Fall hilft es die „no error” messages zu filtern oder zu ignorieren.

- unvorsichtige Einsatz von „error” oder „failure” Level

Manche Entwickler setzen bei der Entwicklung sehr unvorsichtig solche kritische Log-Levels ein. Teilweise handelt es sich um harmlose Einträge wie z.B. fehlgeschlagene Zeitsynchronisierung. Bei den meisten Anwendungen wird eine kleine Uhrzeitabweichung nicht gleich zu falschen Ergebnissen oder gar zum Absturz führen. Trotzdem setzen es manche Programmierer gerne ein. In diesen Fall gibt es kaum praktikable Lösungen. Wenn man versuchen würde die wörter „error” oder „failure” auszufiltern, kann das Ergebnis schnell tatsächlich um wichtige Ereignisse gebraucht werden.

Kapitel 3

Analyse

3.1 Vision

Das fertige Projekt dient einer Post-Disaster Analyse eines Linux basierten Betriebssystem. Ein Disaster wäre z.B. ein nicht mehr funktionierendes System. Das Projekt wird in der Lage mit Linux-OS Computer zu kommunizieren und verschiedene Daten vor dem Ausfall persistent abzulegen. Es soll nach einem kompletten Hardware- oder Stromausfall möglich sein, die für den Ausfall relevanten Daten auszulesen. Die fertige Lösung wird dann die ausgelesenen Daten ohne externe Stromzufuhr auf einem Zeitraum von mindestens 1 Woche aufbewahren können. Denn es ist nicht auszuschließen, dass eine wichtige Speicher-Einheit wie z.B. Festplatte unwiderruflich ausfallen kann. Daher würden die Daten auf einem unabhängigen Speicher aufbewahrt. Dies macht auch bei einem Festplattenausfall die Disaster-Analyse möglich.

Das Auslesen und Analysieren von Daten soll mit einem geringen Aufwand innerhalb weniger Minuten erfolgen. Die Analyse der Daten wird dem Benutzer überlassen, weil nur ein Benutzer mit systemspezifischen Kenntnissen in der Lage ist den Desaster-Hergang anhand der Log Daten zu rekonstruieren.

3.2 Existierende Lösungen

Auf der Suche nach existierenden Lösungen wurde zuerst Microsoft Technet durchgesucht. Die Quelle bietet den Vorteil, dass die Information direkt vom Hersteller des Betriebssystems kommt. Fazit vorweg: in Microsoft-OS Welt gestaltet sich die Fehlersuche relativ aufwendig. Für die Fehlersuche nach einem Bluescreen bietet Microsoft einige Debugging-tools wie z.B. WinDbg sowie DebugView[14]. Die Schwierigkeit liegt in der Handhabung: WinDbg setzt

[14]Microsoft `http://msdn.microsoft.com/en-us/windows/hardware/gg581067` (Stand 20.2.2012)

einen Dump-File nach einem Bluescreen voraus (sehe vorige Fußnote). Der Rechner muss nach dem fehlerträchtigen Ereignis noch in der Lage sein den Dump-File auf eine Persistenz wegzusichern. Dies ist nicht in jeden Fall möglich. Bekannte Beispiele: oben erwähnte Festplattencrash, sowie Stromausfall. Ein weiteres Problem ist ein sehr niedriges Logging-Level auf Windows Ebene, dass sich nicht verändern lässt. Es gibt bei den großen Server-Hersteller wie HP umfangreiche Software Pakete z.B. StartSmart[15] für Fehleranalyse auf Hardware- sowie Software Ebenen. Bei einem vollständigen Persistenz-Ausfall stoßen auch solche Produkte auf ihre Grenzen.

Quelle: Microsoft

Bei Linux war die Suche auf nach Fehleranalyse-Lösungen bei den entsprechenden Distributionen sehr erfolgreich. Die Hersteller der Distributionen stellen in der Regel umfangreiche Wissensdatenbanken kostenlos zur Verfügung.

In Linux-OS Welt lässt sich der Logging-Level in vielen Stufen feingranular einstellen. Bei den Linux-basierten Servern ist es bei kleinen und Mittelständischen Firmen üblich mit Clustering und RAID die Redundanz sicherzustellen[16]. Bei solchen mehrfach abgesicherten Systemen inkl. regelmäßigen Backups stellt ein Ausfall einer einzelnen Maschine in der Regel kein nennenswertes Problem dar. Vereinzeln gibt es Lösungsansätze mit Batch Scripten. Diese Scripts sichern in festgelegten Zeitabständen die Systemlogs auf einen externen Datenträger.

Ein ausführliches Interview von einem erfahrenen Systemadministrator mit langer Berufserfahrung hat ergeben, dass es keine Produkte für die vollständige Lösung von gegebener Problemstellung gibt. Viel mehr werden partiell Insellösungen mit Batches und Backups praktiziert. Hier nochmal eine Auflistung der Verfahren mit ihren hauptsächlichen Einsatzorten, sowie ihre Vor- und Nachteilen:

[15] HP StartSmart `http://h18013.www1.hp.com/products/servers/management/smartstart/index.html`

[16] Mittelstand-Wiki `http://www.mittelstandswiki.de/wissen/Microsite:IT-Sicherheit_im_Mittelstand,_Teil_1` (Stand 20.2.2012)

Verfahren	Einsatzort	Vor-/Nachteile
Backup	Windows, Linux	+Datensicherheit -langsam -speicherplatzintensiv
WinDbg	Windows	+leicht bedienbar -vorausgesetzt PC lauffähig
StartSmart	Windows	+viele Funktionen -teuer -vorausges. PC lauffähig
RAID	Windows, Linux	+günstig +einfache Handhabung -SPOF: Kontroller
HW Redundanz	Windows, Linux	+gute zuverlässig -Einrichting&Datensync -teuer
Cron+Logbackup	Linux	+günstig +einfach +zuverlässig -Ausfallunsicherheit

3.3 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die Fähigkeiten, die das fertige Produkt erbringen soll. Sie sollen den Benutzer bei Lösung der Aufgaben unterstützen, hier bei einer Post-Disaster Analyse. **ISO 9126** dient als Grundlage hierfür. Die funktionalen Anforderungen von Disaster-Recovery Produkt lassen sich mit folgenden Punkten abdecken, weil mindestens 4 sofort gesicherte Logeinträge in der Regel auf die Ausfallursache schließen lassen:

- F1: Funktionalität, Angemessenheit: Das System soll mindestens die letzten 4 Ereignisse auf dem Ubuntu oder Debian Linux Rechner sichern. Damit kann man in den meisten Fällen die Fehlerursache herausfinden, auch dann wenn nicht das letzte Ereignis der Grund hierfür war.
- F2: Funktionalität, Richtigkeit: Die letzten Ereignisse müssen sofort nach dem Auftreten gesichert werden.
- F3: Funktionalität, Ordnungsmäßigkeit: Das System soll die letzte Ereignisse auch beim ausgeschalteten Zielrechner zum Auslesen anbieten. Falls der Rechner sich nicht mehr einschalten lässt, soll das Recovery-System immer noch in der Lage sein, die letzten Ereignisse ohne Verbindung zum Rechner darzustellen.

3.4 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen beschreiben die Produkteigenschaften, die von nicht-fachlicher Natur sind. Sie geben an „wie gut" ein System die geforderten Funktionalitäten umsetzten soll (Qualität). **ISO 9126** dient als Grundlage hierfür. Die nicht-funktionale Anforderungen von Disaster-Recovery fordern folgende Eigenschaften:

- NF1: Funktionalität, Interoperabilität: Das System soll auf einem aktuellen Ubuntu oder Debian Linux laufen. Der Grund hierfür ist die hohe Verbreitung[17] von Debian und

[17] DistroWatch `http://distrowatch.com` Stand: 18.03.2012

Debian basierten Ubuntu Linux, sowie ihre häufige Einsatz in Server- und Desktop Systemen.

- NF2: Zuverlässigkeit, Wiederherstellbarkeit: Die Daten müssen auch bei fehlender Stromversorgung für mindestens 1 Woche vorenthalten werden, weil in der Regel spätestens nach 1 Woche wird das ausgefalle Rechner / Server in kritische Umgebung ausgetauscht (Zielumgebung vom Produkt).
- NF3: Wartbarkeit, Stabilität: Die Anwendung darf nicht mit häufigen Abstürzen die Arbeit stark erschweren oder verhindern
- NF4: Effizient, Zeitverhalten: das Auslesen der Daten soll innerhalb wenigen Minuten lang dauern, oder falls es länger dauert mit einem Fortschritts-Balken deutlich kennzeichnen. Bei mehr als 10 Minuten wäre eine schnelle Fehlersuche beim Ausfall eines kritischen Systems deutlich erschwert.
- NF5: Kosten: die Realisierung inkl. Hardware soll maximal 5000 € kosten. Bei mehr als 5000 € ist es in vielen günstiger sein das komplette System doppelt anzuschaffen und für Desaster-Fall Vorzuenthalten. Nach einem Hot-Swap könnte man dann in aller Ruhe mit herkömmlichen Mitteln wie z.B. Austausch der Hardware-Teilen sich auf die Fehlersuche begeben
- Anmerkung: Einfachheit und Stabilität sind wichtiger als optisch aufwendiges UI mit vielen Optionen.

3.5 Technische Voraussetzungen

In diesen Abschnitt werden die technischen Voraussetzungen für die Umsetzung genauer unter der Lupe genommen um die Problemstellung möglichst optimal herangehen zu können.

Das Logging erfolgt auf einem Linux Betriebssystem. Hierfür wird eine aktuelle Linux basierte Distribution namens Ubuntu vorausgesetzt, weil Ubuntu eine der am weitesten verbreitete Linux Distribution ist. Alle aktuellen Distributionen inkl. Ubuntu bringen einem Syslogd Deamon mit. Syslog ist praktisch ein Standard Protokoll für die wichtige Systemmeldung in Linux-Welt. Die Umsetzung beschäftigt sich nicht mit Optimierung der Logging auf Systemebene, Verbesserung der Systemstabilität oder anderen Maßnahmen, die über das Sichern vom vorhandenen Logs hinausgehen. Daher würde sich ein Board mit einem beschränkten Speichervermögen gut für die Realisierung eignen. Bei der Auswahl von Hardware für die Umsetzung fiel die Entscheidung auf das STMicro M24LR64-R Modul, weil es 2 Schnittstellen aufweist, sehr günstig ist und ausreichend Speicherplatz für textbasierte Logs bietet. Auf

der Seite des Lesegerätes bietet sich ein Smartphone mit NFC Schnittstelle, weil mittlerweile sehr viele Leute Smartphone besitzen und mit sich herumtragen und NFC Technik in der letzte Zeit einer rasanten Verbreitung findet. Auf der Gerätseite wird Android 2.3.x vorausgesetzt, weil NFC erst ab Android 2.3 unterstützt wird. Als Hardware werden folgende Bestandteile als zwingend notwendig angesehen:

- Android Smartphone mit NFC
- Computer
- STMicro M24LR64-R Modul
- Aardvark I2C/SPI Adapter (sehe Kapitel 5.2.1)
- Mini-USB Kabel

Anforderungen an Software und Einstellungen:

- Ubuntu Linux
- Root Berechtigungen für die Einrichtung
- aktuelle Java Runtime Enviroment auf dem Linux-PC

Auf dem Linux-PC werden die USB Treiber für den Aardvark Adapter installiert. Daher sind die Root-Rechte erforderlich.

3.6 Anwendungsfälle

In folgenden werden 2 wichtigsten Einsatzszenarios vorgestellt: „Linux System ist in Betrieb" und „Linux System ist defekt oder ausgeschaltet". Das erste Szenario beschreibt das Verhalten von Überwachungssystem im normalen Betrieb. Im zweiten Anwendungsfall spielt das Überwachungssystem ihre Stärken aus.

3.6.1 Linux System ist in Betrieb

In diesem Szenario ist das System noch vollständig lauffähig und nicht von einem Ausfall betroffen. In diesen Fall soll das fertige Produkt parallel zum System laufen und die letzte wichtige Ereignisse aufzeichnen. Der Anwender soll in der Lage sein mit einem ihn vorgegebenen Verfahren/Gerätschaften die letzten aufgezeichneten Ereignisse zu verfolgen.

1. Linux System wird von Benutzer eingeschaltet und fährt hoch
2. Das Überwachungssystem wird gestartet (Autostart oder manuell)

3. Das Überwachungssystem analysiert letzte Ereignisse mit vordefinierten Verfahren
4. Die Ereignisse werden vom Überwachungssystem außerhalb von Zielsystem gesichert
5. Der Benutzer entscheidet sich für eine Laufzeitanalyse
6. Der Anwender interagiert mit den Überwachungssystem (Lesegerät oder Displayanzeige) nach festgelegten Schema
7. Das Überwachungssystem zeigt den Benutzer die letzte Ereignisse an
8. Der Anwender wertet die Daten aus
9. Das System läuft weiter und Zeichnet die Ereignisse auf

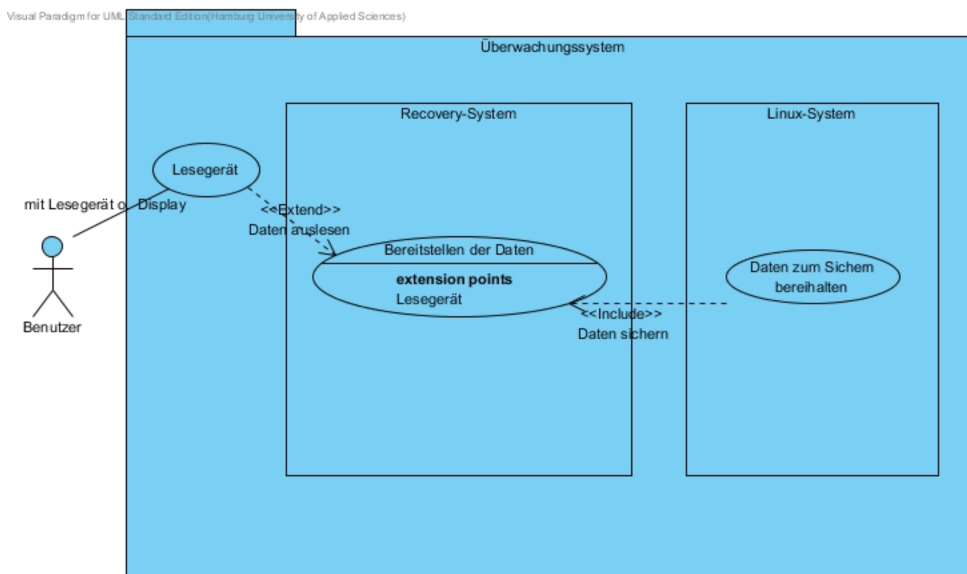


Abbildung 3.1: Use Case - lauffähiges System

Die Ereignisse für die Aufzeichnung sollen so gewählt werden, dass eine Disaster-Analyse anhand der letzten Ereignisse möglich sein soll.

3.6.2 Linux System ist defekt oder ausgeschaltet

Erst in diesem Szenario wird das System ihren Einsatzzweck vollständig erfüllen. Es tritt ein Disaster mit eine auf dem ersten Blick nicht erkennbare Ursache auf. Entweder wurde der Rechner von einem Benutzer ausgeschaltet oder es ist tatsächlich ein Ausfall aufgetreten.

Es wird angenommen, dass die Linux-Server üblicherweise rund um die Uhr laufen. Daher wäre es spannend zu erfahren was die Ursache für ein nicht mehr laufendes System ist. Mögliche Ursachen: das System wurde ordentlich heruntergefahren, defekt geworden oder es gab einem Stromausfall. So sieht es in einzelnen aus:

1. Linux System wird vom Benutzer eingeschaltet und fährt hoch
2. Das Überwachungssystem wird gestartet (Autostart oder manuell)
3. Das Überwachungssystem analysiert letzte Ereignisse
4. Die Ereignisse werden von Überwachungssystem extern gesichert
5. Disaster-Fall ist eingetreten: das Linux-System reagiert nicht mehr oder ist komplett ausgeschaltet.
6. Der Benutzer entscheidet sich für eine Ursachenanalyse.
7. Der Anwender liest die die Ereignis-Daten aus den Überwachungssystem nach festgelegten Verfahren aus
8. Der Benutzer wertet die Daten aus und entscheidet sich anhand der ausgewerteten Daten für weiteres Vorgehen
9. Der Anwender führt alle nötige Schritte aus um das Linux-System wieder lauffähig zu machen
10. Das Linux-System wird von Benutzer eingeschaltet und fährt hoch (weiter mit Punkt 1 aus vorigen Unterabschnitt)

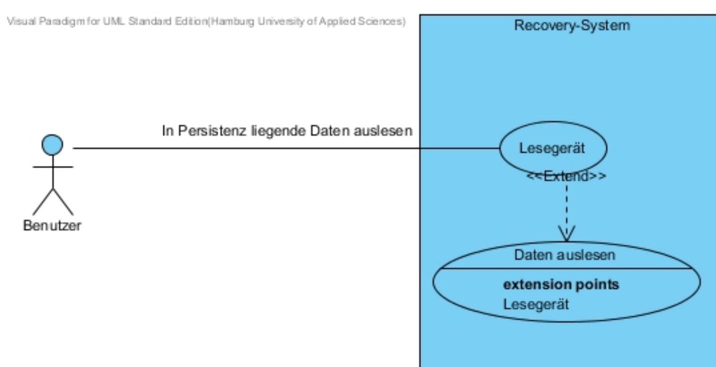


Abbildung 3.2: Use Case - Disaster

Die Ereignisse für die Aufzeichnung sollen so gewählt werden, dass eine Disaster-Analyse anhand der letzten Ereignisse möglich sein soll.

Bei einem Stromausfall wäre es z.B. von Vorteil vor dem regulären booten zu erfahren ob es beim Ausfall zum Datenverlust gekommen ist. Eine mögliche Reaktion auf einen Stromausfall wäre beispielweise ein Diskcheck/Repair der Festplatte vor dem ordentlichen Zugriff auf die Persistenz. Damit würde man sicherstellen, dass keine Daten beschädigt sind, bevor anschließend ein ordentliches Bootvorgang durchgeführt wird.

Kapitel 4

Entwurf & Architektur

4.1 Architektur

Die Architektur stellt den Entwicklungshergang schematisch dar. Sie enthält verschiedene Sichten auf das Projekt sowie eine Reihe von Begründungen für die getroffenen Entscheidungen.

Die Grundlage für die Auswahl bestimmter Soft- und Hardware Ausstattung in Kapitel „Technische Voraussetzungen” (3.5) stellen die funktionale, sowie nichtfunktionale Anforderungen dar. Die o.g. Entscheidungen erfühlen die funktionale Anforderungen (3.3) F1 und F3, weil für die Umsetzung gewählte EEPROM 64 kBit Speicher bietet, die Daten auch ohne Stromversorgung im Speicher monate- bzw. jahrelang bleiben und mit NFC Interface auch ohne PC-Verbindung zum Auslesen bereitstehen. Die funktionale Anforderungen F2, F3 und NF3 werden durch die entsprechende Implementierung sichergestellt. Die nichtfunktionale Anforderung NF1 wird durch die Voraussetzungen an der Software-Seite erreicht. NF2 „Zuverlässigkeit, Wiederherstellbarkeit” wird mit physischen Eigenschaften der EEPROM realisiert: EEPROM kann die Daten üblicherweise bis zu 40 Jahren vorenthalten[18]. Die nichtfunktionale Anforderung Nr.4 wird durch Verzicht auf Kabel mit NFC Technik erreicht. Von Startvorgang der App bis vollständige Anzeige aller Daten aus Eeprom vergeht in der Praxis maximal 1 Minute. Die NF5 ist ebenfalls erfühlt. Sehe Kostenaufstellung in Kapitel „(4.1.1) Entscheidungsgrundlage bei der Modellierung”.

4.1.1 Entscheidungsgrundlage bei der Modellierung

Bei der Modellierung der Infrastruktur für die Umsetzung wurde folgende Ziel verfolgt: die Sicherung von Linux-Logging auf einem externen EEPROM, sowie auslesen mit einem Android-Smartphone per NFC. Die Linux Logs geben bei einem feingranular eingestellten

[18] EEPROM Hersteller STMicro `http://www.st.com/internet/mcu/class/1276.jsp` Stand: 26.3.2012

Logging-Level sehr ausführliche Informationen über die letzten Systemereignisse. Daher eignen sie sich sehr gut für die Disaster-Hergang Rekonstruierung. Android-Smartphone mit NFC wurde auf Grund der Verbreitung ausgewählt: mittlerweile haben sehr viele Leute einen Smartphone in der Tasche und NFC Technik eine rasante Verbreitung findet. Auf den meisten Smartphones läuft Android-OS[19]. Daher ist es einfacher und kostengünstiger ein alltägliches Gegenstand wie z.B. Smartphone zu benutzen als ein proprietäres Anzeigegerät einzusetzen und gegeben falls zu konfigurieren. Als BlackBox Komponente wurde ein M24LR Dual Interface Board von STMicro ausgewählt, weil es genug Speicherplatz für die Logdaten bietet, sehr kostengünstig ist und zwei Schnittstellen für wire- & wireless Kommunikation bietet.

Hier ein kleines Übersicht der Kosten für Hardware-Komponenten:

- Totalphase Aardvark I2C Host-Adapter: ca. 170 €
- STMicroelectronics M24LR Dual Interface EEPROM: ca. 1 €
- Mini-USB Kabel: 2 €
- (falls nicht bereits vorhanden) Android Smartphone mit NFC z.B. Google Nexus S: ca. 240 €

Wie man in der obere Aufstellung sieht, sind die Hardware-Kosten für Server-Bereich (Zielumgebung) relativ überschaubar. Zum Vergleich: SAS 2TB Server Festplatte - 450 €[20].

Die Kommunikation zwischen der Blackbox und Benutzer-Lesegerät soll über NFC stattfinden, weil es ohne Kabel schnell und einfach durchführbar ist. In eine kritische Situation wie z.B. ein Desaster ist es von großem Vorteil wenn der Benutzer sich nicht mit Kabel und Treiber beschäftigen muss.

4.1.2 Kontextsicht

Die Kontextabgrenzung grenzt das System von Nachbarsystemen ab. Dabei wird das Zusammenspiel mit den angrenzenden Systemen dargestellt. Es werden die Schnittstellen festgelegt und beschrieben[21].

[19] Spiegel `http://www.spiegel.de/netzwelt/netzpolitik/0,1518,761434,00.html` Stand: 25.3.2012

[20] Amazon `http://www.amazon.de/gp/product/B0054RF1L2/` Stand: 25.3.2012

[21] Sarstedt (2011)

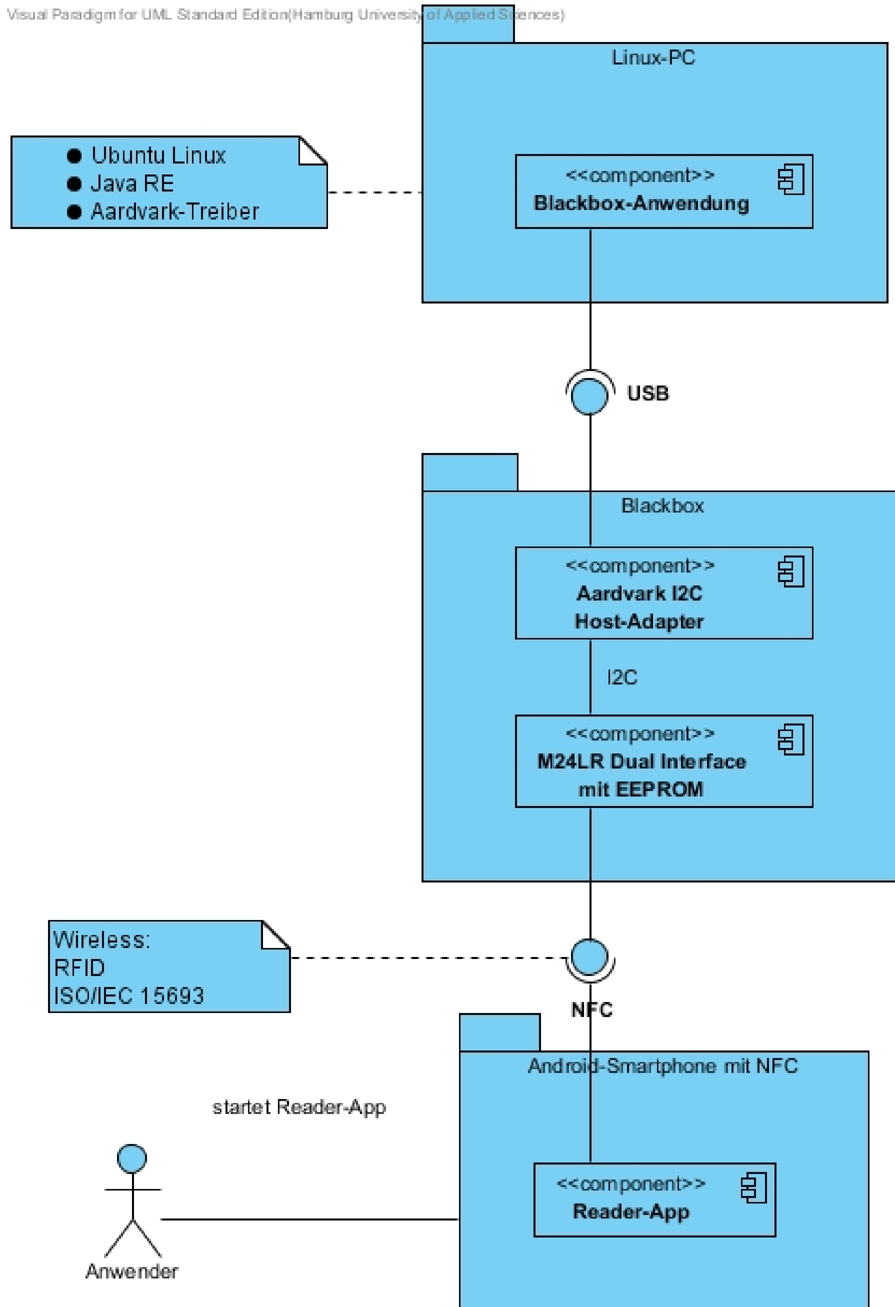


Abbildung 4.1: Blackbox-Kontextabgrenzung

Als eine zentrale Komponente des Blackbox Systems wurde das M24LR Dual Interface Modul mit EEPROM ausgewählt. Sehe dazu „Entscheidungsgrundlage bei der Modellierung" (4.1.1). Wie der Name schon sagt, handelt es sich um einen Modul mit zwei Schnittstellen: eine kabelgebundene I2C Schnittstelle und eine kabellose mit NFC kompatible RFID Schnitt-

stelle nach ISO 15693. Da ein handelsüblicher PC kein I2C Anschluss besitzt, wird ein Aardvark I2C -> USB Host-Adapter eingesetzt. Es übernimmt die Kommunikation mit dem M24LR Modul über eine normale USB 2.0 - Schnittstelle und stellt somit eine Brücke zwischen den PC und BlackBox dar.

Auf dem Linux-PC läuft eine Blackbox-Anwendung, die dafür sorgt dass die neusten Log-Einträge sofort nach dem Auftreten über die von Blackbox bereitgestellte USB Schnittstelle im EEPROM von M24LR landen. Hierfür benutzt die Blackbox-Anwendung die Linux USB-Treiber von Aardvark Host-Adapter.

4.1.3 Bausteinsicht

In der White-Box Sicht wird die innere Struktur von Blackbox System genauer dargestellt. Dabei wird das Zusammenspiel der Komponenten untereinander und mit den Schnittstellen dargestellt[22].

[22] Sarstedt (2011)

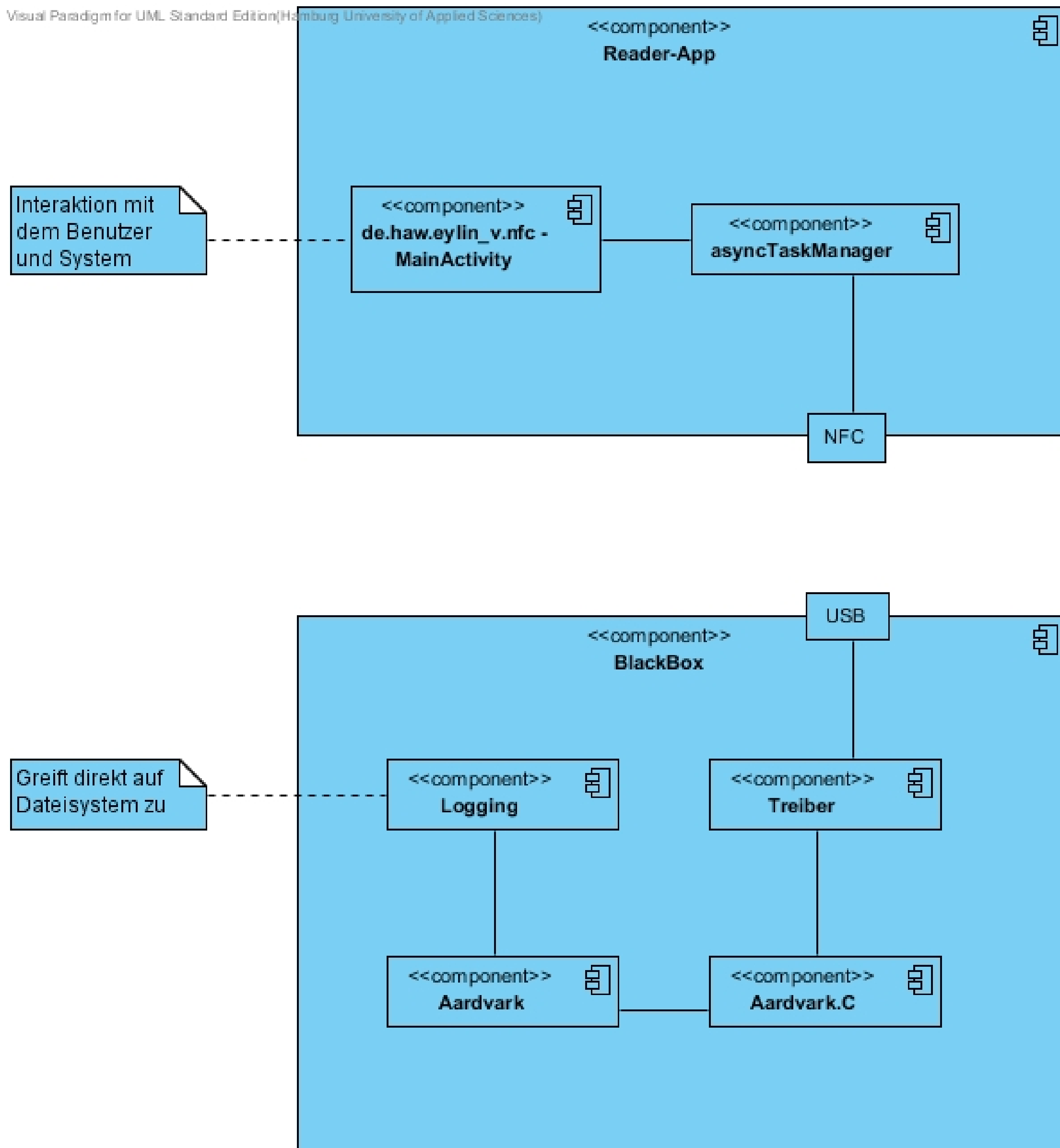


Abbildung 4.2: WhiteBox

Die Logging-Komponente spielt auf dem zu überwachenden Rechner eine zentrale Rolle. Für die Überwachung wurde JNotify-Framework verwendet. Bei der Wahl von JNotify handelt es um eine Entscheidung technischer Natur. Sehe dazu Kapitel „Implementierung" (5.1). Die Komponente überwacht mit Hilfe von JNotify[23] alle Änderungen auf den Logfiles. So bald

[23]JNotify `http://jnotify.sourceforge.net`

eine Änderung z.B. durch einen neuen Eintrag in den Logfiles eintritt, liest die Komponente eine vorweg definierte Anzahl an letzten Einträgen und schickt sie an Aardvark-Komponente.

Die Aardvark-Komponente kümmert sich um die Speicherverwaltung von EEPROM der M24LR. Sie bereitet die Log-Einträge so auf, dass sie schnell, speicherplatzsparend und effizient in EEPROM abgelegt werden. Dafür werden solche Techniken wie z.B. Page-Writing und Roundrobin eingesetzt (sehe Kap „4.3 Vorgehen bei der Entwicklung”). Die Konfiguration und Handling von dem Gerät übernimmt die Aardvark.C Komponente. Sie enthält vom Hersteller vorgegebene Flags und Bytecodes, die man absetzen muss um mit dem M24LR über I2C zu kommunizieren. Der Aufruf von Methoden und Systemcalls aus der Aardvark.C Komponente erfolgt über Java Nativ Interface (JNI). Die hardwarenahe Befehle werden dann mit Hilfe von USB-Treibern von dem Aardvark an das M24LR gesendet. Das M24LR interpretiert sie und setzt sie um, in dem er z.B. die Daten in bestimmte EEPROM Bereiche ablegt. Beim Entwurf wurde bewusst die technische Komponente Aardvark zwischen der Logging-Komponente und Aardvark.C eingeführt um fachliche und technische Ebenen voneinander zu trennen.

Die Reader-App auf dem Android-Smartphone von dem Benutzer besteht aus 2 Komponenten: MainActivity und asyncTaskManager. Die MainActivity ist für die optische Darstellung von User UI auf dem Telefon-Display zuständig. Sie bietet dem Benutzer eine übersichtliche Darstellung von ausgelesenen Daten sowie eine Aktivitätsanzeige. Die eigentliche T-Komponente für die Kommunikation mit dem NFC-Interface von dem Android-Smartphone ist die asyncTaskManager Komponente. Sie greift auf die Android OS 2.3.x zur Verfügung gestellte NFC Schnittstelle zu um mit dem M24LR über NFC Interface zu kommunizieren. Sie liest die EEPROM Daten von M24LR aus in dem sie die von dem Hersteller definierte Byte Kommandos sendet und die Antworte auswertet. Der asyncTaskManager stellt somit eine technische Komponente dar um NFC Kommunikation von dem Anwendungskern, in diesen Fall MainActivity, zu trennen.

4.1.4 Laufzeitsicht

Der Laufzeitsicht beschreibt welche Teile von der System zum welchen Zeitpunkt existieren und wie sie untereinander interagieren. In diesem Fall wird ein wichtiger Anwendungsfall „Linux System defekt” kurz vor seinem Eintritt aufgezeigt um einen besseren Überblick über die Interaktion der Komponenten vor dem Desaster Eintritt zu erhalten.

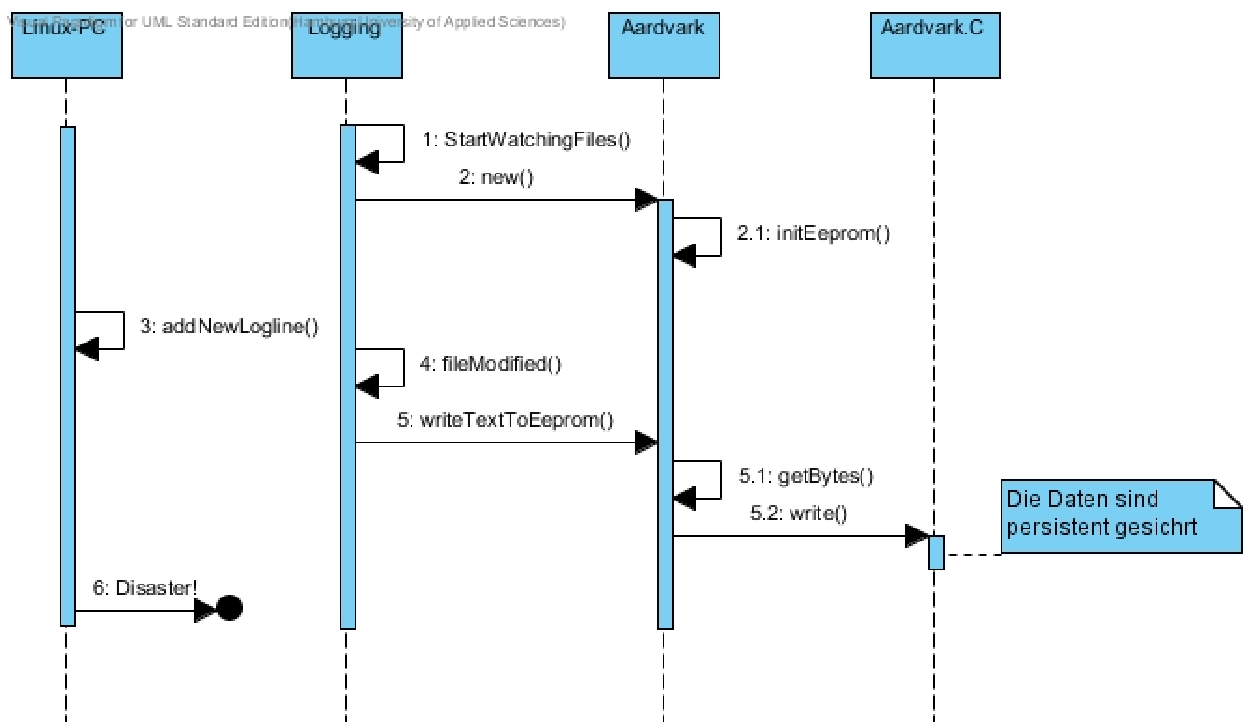


Abbildung 4.3: BlackBox Laufzeitsicht

Am Anfang startet die Logging-Komponente die Überwachung von einem vorhin festgelegten Dateipfad. Die Logging-Komponente initialisiert unmittelbar nach dem Start die Aardvark Komponente um alle Logging-Ereignisse sofort nach dem Eintreten abspeichern zu können. Die Aardvark-Komponente initialisiert am Anfang ihrer Lebenszyklus den Eeprom-Handler. Ab diesem Punkt ist das Blackbox-System einsatzbereit.

Nun wird ein neues Ereignis in dem Log abgespeichert. Dadurch wird die fileModified() Funktion *sofort* ausgelöst. Sie stoßt ihrerseits den Schreibvorgang an. Die Logging-Komponente extrahiert die letzten Logeinträge und übergibt sie an writeTextToEeprom(). Ab diesen Punkt übernimmt die die T-Komponente Aardvark den „Persistenz-Write” Vorgang. Sie wandelt hierfür die Textstrings in Byte-Arrays um, teilt sie in Pages[24] auf und generiert Schreibbefehle, die das M24LR auch „verstehen” kann. Direkt danach setzt die Aardvark.C Komponente an das Aardvark gesendete Befehle mittels Java Nativ Call (JNI) um, weil die Aardvark-Treiber sich nicht direkt mit Java ansteuern lassen.

[24]Ein „Page” ist eine Zusammenfassung von mehreren zusammengehörenden Datenelementen in Blöcke um ihre Verarbeitung zu beschleunigen

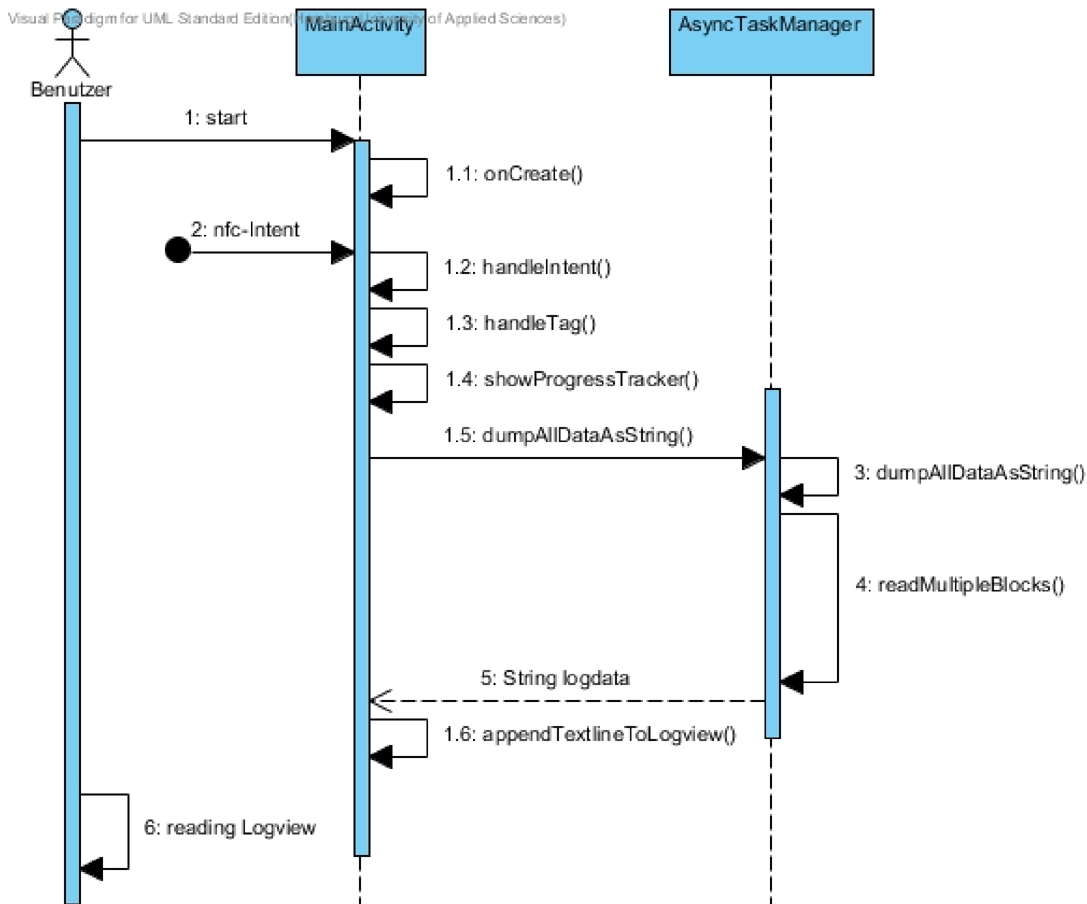


Abbildung 4.4: Reader-App Laufzeitsicht

Die Reader-App stellt eine Schnittstelle zwischen den Benutzer und BlackBox System. Sie stellt per NFC eine drahtlose Verbindung zum BlackBox mit dem Smartphone, auf dem sie läuft, her. Danach liest sie alle in EEPROM gespeicherte Byte-Array Daten aus und wandelt sie in menschenlesbaren Text um.

Am Anfang startet der Benutzer die Reader-App auf seinem Smartphone. Sofort nach dem Starten begrüßt die Reader-App den Benutzer mit „Ready to read" Nachricht auf dem Display. Dies geschieht sofort nach dem die onCreate() Funktion von MainActivity den User Interface aufgebaut hat. Die Benutzeroberfläche bietet alle wichtigen Funktionen auf einem Blick an. Sobald der Benutzer sein Smartphone in der Nähe von Blackbox hält, wird ein NFC Intent[25] von Android-OS ausgelöst. Das Programm fängt den Intent ab und behandelt es mit handleIntent(). Bei der Bearbeitung von Intent werden die NFC Tag Daten mit handleTag() behandelt. Die Tag-Daten enthalten alle wichtigen technischen Eigenschaften von der Ge-

[25] Intent ist eine Aktion mit zugehörigen Daten in Android-OS. Ein Intent wird durch ein Ereignis oder einer Anwendung ausgelöst und kann eine Aktion wie z.B. Activity starten.

genstelle um die Kommunikation über die Luftschnittstelle herzustellen. Fürs erste hat die MainActivity ihre Arbeit getan und ruft den Progress Tracker mit showProgressTracker() an. Dadurch erhält der Benutzer ein Gefühl, dass die App im Hintergrund arbeitet. Es ist aus Benutzbarkeitsgründen notwendig, weil ein komplettes Auslesevorgang teilweise bis zu 2 Minuten dauern kann.

Im nächsten Schritt übernimmt die AsyncTaskManager Komponente nach dem Aufruf von dumpAllDataAsString() alle weitere Aktionen auf T-Ebene. Sie generiert eine Reihe von Read-Befehlen um den Speicher von BlackBox auszulesen und sendet sie nacheinander ab. Als Ergebnis bekommt die Komponente ein Byte-Array. Daraus generiert sie ein String und gibt an die MainActivity wieder zurück. Die MainActivity blendet den Progress Tracker wieder aus und zeigt den Inhalt der BlackBox mit appendTextlineToLogview() auf dem Smartphone Display an.

Jetzt kann der Benutzer mit dem Lesen und Auswerten von BlackBox Inhalt beginnen.

4.1.5 Verteilungssicht

Verteilungssicht (deployment diagramm) ist ein Synonym für Infrastruktursicht. Es beschreibt die Ablaufumgebung des Systems in Form von Hardwarekomponenten. Außerdem beschreibt es eingesetzte Betriebssysteme, Leistungsdaten, Nachbarsysteme etc[26]. Das Ziel von Verteilungssicht ist die Kommunikationsmechanismus, Zuordnung von Bausteinen, Protokoll und Zusammenspiel mit Nachbarsystemen besser zu verstehen.

Bei der Erstellung von Verteilungssicht fiel die Entscheidung auf freien Notationsstiel um einen starren Korsett von wenigen UML Symbolen zu umgehen und das Gesamtsystem besser zum Ausdruck zu bringen.

[26] Sarstedt (2011)

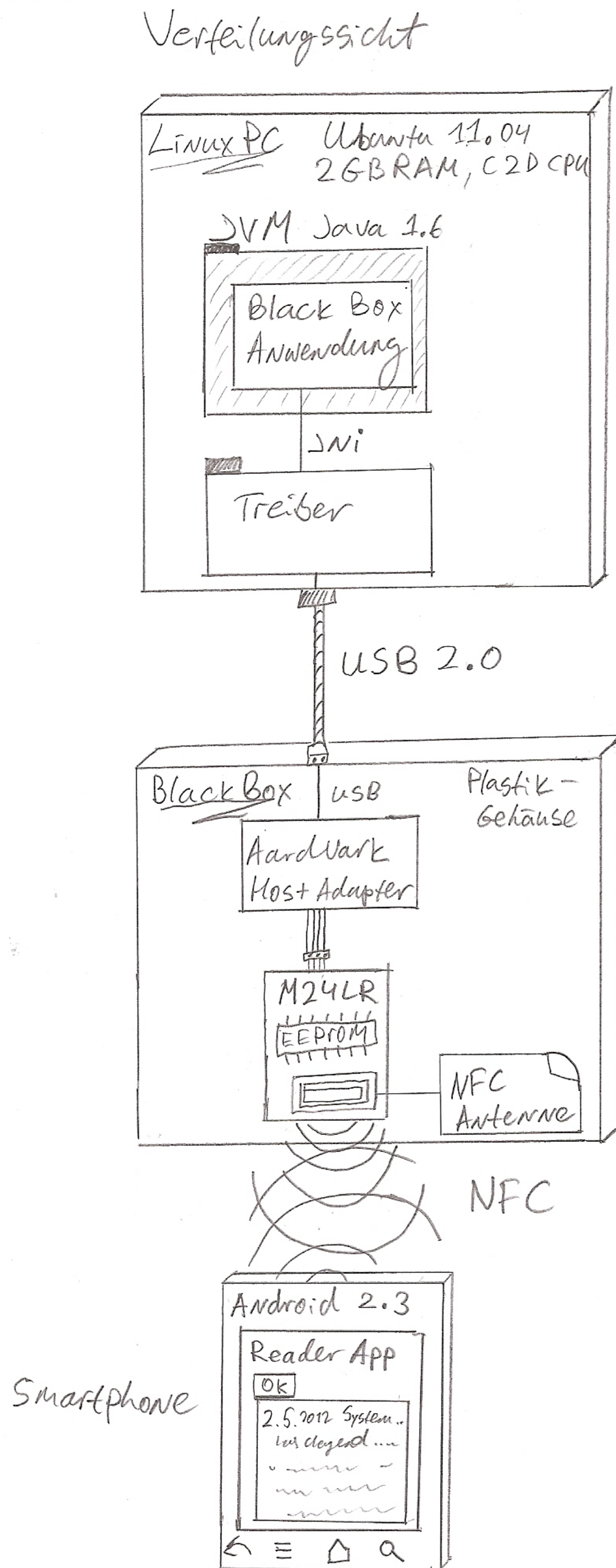


Abbildung 4.5: Blackbox Verteilungssicht

Als Zielsystem dient ein Linux-Rechner. Bei der Umsetzung wurde ein Ubuntu 11.04 Rechner mit handelsüblicher Ausstattung wie z.B. Core 2 Duo CPU oder 2 Gigabyte Ram-Speicher verwendet. Nach der Einrichtung von Linux-Rechner sind noch die Aardvark-Treiber und Oracle Java 1.6 Laufzeitumgebung nachinstalliert worden. Die Blackbox Anwendung selbst läuft in eine Java Virtual Machine Umgebung, wie es bei Java-Anwendung üblich ist. Die Kommunikation zwischen Treiber und Blackbox-Anwendung findet mit Hilfe von JNI statt (sehe „4.1.3 Bausteinsicht" Kapitel).

Das BlackBox System ist per USB 2.0 an das Zielsystem angebunden. Die Entscheidung das BlackBox System außerhalb des Rechners aufzustellen hat mehrere Gründe: zunächst hat nicht jeder PC-Gehäuse genügend Platz zum Aufstellen der BlackBox (z.B. ein Notebook) und zum zweiten würde es den Datenverlust in Falle eines Brandes in die Gehäuse verhindern. Außerdem lässt sich so eine BlackBox deutlich einfacher per USB anschließen und einrichten, statt mit einer proprietäre Schnittstelle, weil USB so gut wie alle modernen Rechner aufweisen.

Ein Kabel führt die Daten von Linux-Rechner zum Aardvark Host-Adapter. Das Aardvark Adapter überträgt die Daten über eine I2C Verbindung zum M24LR Platine, wo sie im EEPROM landen. Die M24LR Platine hat neben der Kabelschnittstelle noch eine weitere Schnittstelle: NFC-Antenne. Sie ermöglicht die kabellose Kommunikation mit dem Android-Smartphone über NFC.

Ein Android-Smartphone mit Android OS 2.3 stellt eine Schlüsselkomponente zwischen Benutzer und BlackBox System dar. Die Reader-App Anwendung ist in der Lage über die NFC Schnittstelle von Android-Smartphone mit der BlackBox zu kommunizieren und die Daten auf dem Display darzustellen.

4.2 Design

Im vorigen Kapitel wurde der Entwurf von Architektur beschrieben. Das Design der Software stellt dagegen einen wichtigen Schritt zwischen den Architekturentwurf und tatsächliche Implementierung. Bei der Design-Erstellung wurde keine Software Design Description (SDD) verwendet, weil es sehr umfangreich ist und sich besser für große Projekte eignet. Stattdessen wird in diesem Abschnitt erklärt welche Design-Patter bei der Architekturumsetzung zum Einsatz gekommen sind.

4.2.1 Motivation für Einsatz

Es empfiehlt sich bei der Wahl von Software-Design auf Design-Patter zurückzugreifen, weil die Entwurfsmustern ein erprobtes Lösungskonzept für bestimmte Problemstellungen darstellen. Mit einer geschickten Wahl eines Entwurfsmusters kann man praxiserprobte Lösungswege anwenden ohne das Rad neuerfinden zu müssen. In diesen Fall würde der zuge-

hörige Lösungsweg viel Aufwand für die Umsetzung sparen. Die Herausforderung besteht also bei der Wahl eines Entwurfsmusters oder gegeben falls eine komplette Ablehnung davon, falls es einen viel besseren und eleganteren Lösungsweg gibt. Ein Anwendungsentwickler mit viel Erfahrung wäre schnell in der Lage ein passendes Entwurfsmuster zu finden und ihn umzusetzen. Bei weniger Erfahrung empfiehlt es sich das Problem in ihre Grundbausteine zu zerlegen und Standardlösungen dafür zu finden. Eine mehrfache Lösung der Probleme würde auf einen Lösungsmuster hindeuten. Damit könnte man sich auf einem Design Patter stützen. Mit dem Einsatz von einem Entwurfsmuster würde man den Erfahrungsschatz der Profis auf ihrem Fachgebiet nutzen. Diese Vorgehensweise entspricht der Wiederverwendung Philosophie von objektorientierte Entwicklung.

Man sollte also immer versuchen den Lösungsweg über Design-Patter anzustreben, aber man muss gleichzeitig in der Lage sein in einigen wenigen Fällen sich dagegen zu entscheiden.

4.2.2 Observe Patter

Das BlackBox System überwacht die Logfiles und schreibt alle neuste Änderungen in das EEPROM. So lange es keine neuen Einträge gibt, befindet sich das BlackBox System im Ruhezustand. Die Logging-Komponente wird erst dann aktiv, wenn ein Ergebnis z.B. ein neuer Logeintrag eintritt. Das **Observe Patter** Design würde sich gut für die Umsetzung eignen, weil es auf dem Abo-Prinzip basiert (Subscribe). Die Logging-Komponente würde als Beobachter bestimmte Änderungen an den Logdateien abonnieren und erst dann aktiv werden, wenn eine Änderung eintritt. Damit werden die Systemressourcen geschont und die BlackBox kann nach einem Ereignis schneller die Logeinträge auf die Persistenz kopieren. Das System implementiert ein so genanntes Observer (Beobachter), um bei einem vordefinierten Ereignis die Funktionskette der (3.6) Anwendungsfälle in Bewegung zu setzen. Der Observer abonniert beispielsweise alle Änderungen bei einem Subjekt und wird erst dann tätig, wenn sich der Zustand von dem Objekt ändert.

4.3 Vorgehen bei der Entwicklung

Bei der Entwicklung von Architektur und Design wurde im Vorfeld eine umfangreiche Recherche durchgeführt. Das Ziel davon war rauszufinden, wie man die funktionale und nichtfunktionale Anforderungen erfüllen könnte. Außerdem wurde die Recherche genutzt um herauszufinden welche Design Patter sich am besten für die Umsetzung dieser Architektur eignet. Dabei hat sich der Vorlesungsskript „Architektur von Informationssystemen”[27] als eine gute Informationsquelle für das Entwurf und weitere Recherchen ergeben.

[27] Sarstedt (2011)

4.3.1 Entscheidungsgrundlage für iterative Entwicklung

Bei der Entwicklung kamen sehr viele neue unterschiedliche Komponenten wie z.B. Aardvark und M24LR Board zum Einsatz. Da es bisher keine vergleichbare Kombination aus Hard- und Software für die Lösung es Problemveröffentlicht wurde (sehe 3.2 „Existierende Lösungen"), muss man sich in mehreren Schritten an das Ergebnis vorantasten. Mit jedem Schritt wäre zu klären ob das eine oder andere Komponente sich wie vorgestellt verhält und sich für den Einsatzzweck eignet, weil man bei einem neuen Umfeld nicht davon ausgehen kann, dass der Lösungsansatz auch in der Praxis funktionieren wird. Es gibt im Bereich der Software-Entwicklung einen vergleichbaren Ansatz: das inkrementelles Vorgehensmodell. Das Modell enthält das iterative Entwicklungsmuster.

Bei dem iterativen Entwicklungsmuster versucht man das Problem schrittweise heranzugehen und zu lösen. Dabei teilt man das gesamte Projekt in viele einfache, aufeinander aufbauende Iterationsschritte auf. Es werden die Ergebnisse aus den vorigen Iterationsschritten in den kommenden Schritten einzusetzen. Dabei wird es zwischen Iteration für die Implementierung und Verbesserung des Entwurfes unterschieden.

Das iterative Modell würde sich daher ganz gut als Vorgehensmodell bei der Entwicklung eignen, weil es mit jedem Schritt möglich ist das eine oder andere neue Komponente auf ihre Eignung zu prüfen und anschließenden in die Architektur einzubauen. Damit umgeht man das Risiko, dass eine in die Architektur eingeplante aber bisher noch nicht ausreichend erprobte Komponente im Nachhinein den erfolgreichen Projektabschluss gefährdet.

4.3.2 Iterative Vorgehensweise

Die iterative Entwicklung besteht aus einer Vielzahl aufeinander aufbauender Iterationsschritte. Am Anfang eines Schrittes steht die Planung. In der Planungsphase wird festgelegt was, wie und wie schnell erreicht werden soll. Danach werden die Anforderungen an das Ergebnis festgelegt. Nach dem die Anforderungen festgelegt sind, muss die genauere Umsetzung und Herangehensweise definiert werden. Im nächsten Schritt werden die Anforderungen implementiert. Sobald die Implementierung abgeschlossen ist, müssen die Ergebnisse getestet und verifiziert werden. Die aus dem aktuellen Iterationsschritt gewonnene Erkenntnisse und Ergebnisse fließen in den nachfolgenden Schritt.

Ein Beispiel für einen Iterationsschritt ist die Kommunikation von Android-Smartphone mit NFC Gegenstelle: die technische Datenblätter und Beschreibungen von STMicro definieren einige Eigenschaften von M24LR Chip, die geprüft werden müssen. Darunter zählen essenzielle Write- und Read Befehle. Google Inc. hat zwar in ihren Android-API Dokumentation die NFC Schnittstelle definiert, aber nicht genau NFC-Spezifische Befehle erläutert und ihre Einsatzmöglichkeit bestätigt. Daher müsste man im Vorfeld mit einem Iterationsschritt klären ob ein Nexus S Smartphone sich mit dem M24LR Chip versteht und die über die Android 2.3 NFC API gesendete Commands auch tatsächlich korrekt gesendet und emp-

fangen werden können. Damit waren Anforderungen für den Iterationsschritt definiert. Die Vorgehensweise bestand darin ein Testprogramm zu implementieren, die nachweist, dass das M24LR Board die Read- und Write-Befehle von einem Android Smartphone versteht und richtig umsetzt. Nach dem eine Testanwendung implementiert wurde, wurde es sofort auf Korrektheit der Ergebnisse getestet. Wenn die Tests erfolgreich abgeschlossen waren, konnte man das M24LR Board und Sourcecode aus der Testanwendung in den nächsten Iterationsschritt übernehmen. Parallel dazu wurden die Ergebnisse in die Architektur übernommen. Da bei der iterativen Entwicklung kein Ende vorgesehen ist, muss der Projektleiter selbst entscheiden, wann das Projekt die endgültige Reife erreicht hat.

Kapitel 5

Realisierung

5.1 Implementierung

In diesem Abschnitt wird es genauer auf die technische Umsetzung des BlackBox Systems eingegangen. Die Hauptprogrammiersprache ist Java, weil es um eine Multiplattform Programmiersprache handelt. Die in Java geschriebenen Anwendungen sind grundsätzlich multiplattformfähig und lassen sich mit wenig Aufwand beispielsweise von Linux auf Windows übertragen. Damit wäre es z.B. möglich die BlackBox Anwendung auch auf einem Microsoft Windows Server laufen zu lassen.

5.1.1 Entwicklungswerkezeuge

Folgende Entwicklungs- und Planungswerkzeuge kamen dabei zum Einsatz:

- Java 1.6 SDK
- Eclipse RCP
- Subversion
- Android SDK
- Logic Analyser
- Visual Paradigm for UML

Java 1.6 Software Development Kit stellt ein Grundbaustein für die Entwicklung dar. Es ist eine Laufzeitumgebung für das gesamte Projekt, das fast ausschließlich in Java geschrieben wurde. Die Entscheidung für Java fiel aus 3 Gründen:

1. es ist möglich mit wenig Anpassungsaufwand die BlackBox Anwendung auch unter anderen Betriebssystemen laufen zu lassen.

2. Auf dem NFC-Smartphone läuft Android OS. Android Anwendungen müssen in Java geschrieben werden, weil Google als Plattform-Entwickler es mit den APIs vorgeschrieben hat.

3. Widerverwendbarkeit der Ergebnisse: einige Komponente, die für Android geschrieben sind, können auch unter Linux in BlackBox Anwendung verwendet werden. Zum Beispiel die technische Komponente NfcConnector.

Eclipse Rich Client Plattform ist ein der mächtigsten Werkzeuge für die Java Entwicklung. Eclipse IDE ist Open Source und kostenlos. Es sind unzählige Plugins und Communitys vorhanden, die Eclipse unterstützen. Ein K.O. Kriterium bei der Auswahl von Entwicklungsumgebung war die Verfügbarkeit von Android SDK. Google stellt Android SDK Erweiterung aktuell nur für Eclipse RCP zur Verfügung.

Subversion ist eine freie und moderne Software für die Versionsverwaltung der Dateien. Es eignet sich hervorragend für die Versionsverwaltung von Software-Sourcecode. Es wird oft in Entwicklermannschaften eingesetzt um gemeinsam an einem Projekt zu entwickeln und den Source-Code zu versionieren. Obwohl an der Entwicklung nur eine Person beteiligt war, war es trotzdem sehr nützlich SVN einzusetzen. Einerseits war es sehr einfach alle Änderungen auf einem Server zu sichern und notfalls zu einer stabilen Version von der Anwendung zurückkehren zu können. Anderseits hat es die Synchronisierung von Sourcecode unter mehreren Notebooks und Rechner deutlich vereinfacht.

Android SDK ist eine Sammlung aus Programmen, Plugins und Dokumentation die Entwicklung unter Android OS ermöglicht. Android SDK basiert auf Eclipse und bietet einige sehr hilfreiche Werkzeuge wie z.B. LogCat für die Betrachtung Android Systemlogs oder ein Android Virtual Device (AVD) um verschiedene Laufzeitumgebungen zu simulieren. Ohne Android SDK ist die Entwicklung unter Android OS praktisch unmöglich, daher stellt es ein Pflichtwerkzeug dar.

Logic Analyser ermöglicht den gesamten Traffic auf einen Datenbus wie beispielsweise I2C aufzuzeichnen und komfortabel grafisch auszuwerten. Damit gelangt man in die Lage bitweise die gesamte Kommunikation zwischen den PC und M24LR Board auszuwerten und festzustellen ob ein Problem auf Hard- oder Software Ebene angesiedelt ist. Einen groben Überblick über die Auswertung erhält man in Kapitel (5.2.3).

Visual Paradigm for UML von gleichnamige Firma ist ein sehr mächtiges Werkzeug für die Erstellung von UML Diagrammen. Es ermöglicht einfach und schnell gut aussehende Diagramme zu erstellen. Fast alle im Kapitel (4.1) verwendeten Grafiken sind mit Visual Paradigm erstellt worden.

5.1.2 Herangehensweise

Wie im Kapitel (4.3) „Vorgehen bei der Entwicklung" beschrieben, verlief die Entwicklung iterativ. Das erste Iterationsschritt bestand darin rauszufinden wie die NFC Kommunikati-

on zwischen den Smartphone und eine NFC Gegenstelle funktioniert. Als NFC Gegenstelle kam ein LRiS64K Large-memory RFID IC vom STMicro zum Einsatz, weil es alle benötigten NFC Kommandos von M24LR Dual Interface Board zur Verfügung gestellt hat. Bei LRiS64K handelt es sich um einen kleinen NFC Tag in Plastikkarte-Form. Das LRiS64K ist auch von Eeprom Speicherplatz her vergleichbar mit dem M24LR Board: beide haben 64 kBit Eeprom Speicher zur Verfügung. Nach dem Studium von Android-API für NFC und Analyse von einigen Code-Beispielen auf diversen Entwicklerportalen wie z.B. Stackoverflow wurde als erstes eine einfache Android App ohne User Interface geschrieben. Im Anschluss daraufhin wurde eine T-Komponente namens NfcConnector entwickelt. Sie übernimmt die Kommunikation auf Low-level mit den NFC Chip. Android API bietet nur ein Möglichkeit mit dem Chip zu kommunizieren: in dem man mittels trancive() ein Byte-Collection an die NFC Gegenstelle sendet. Der Programmierer ist selbst dafür zuständig ein passendes Byte-Array mit nötigen Flags zu generieren, an's Chip zu senden, mögliche Sendefehler abzufangen und eine Antwort in Byte-Array Form auszuwerten. Solche Low-level Vorgehensweise stellt einen sehr unüblichen Vorgang in objektorientierten Java-Welt, weil alle restlichen Android Komponenten und Funktionalitäten objektorientiert sind. An diese Stelle darf nicht unerwähnt bleiben, dass die Auseinandersetzung mit Byte-Arrays, Flags, Bit-Masken und Byte-shifting sehr viel Aufwand mit mehreren Wochen Vorlaufzeit gekostet hat.

```
/**
 * generating an write-command for the LRiS64K EEPROM Flags: DATA_rate_Flag:
 * 1: High data rate is used Protocol_extension_flag: 1: Protocol format
 * extension
 *
 * @param Integer
 *            address to write an data. block-range: 0-2047
 * @param data
 *            array with 4 bytes to write
 * @return single block write command with given data
 */
private byte[] singleBlockWritecommand(int adressToWrite, byte[] data) {
    // 0x0A,0x21 = write command, byte[2] = lower Block-Index, byte[3] =
    // higher Block-Index, byte[4-7] = data
    return new byte[] { 0x0A, 0x21, (byte) adressToWrite,
            (byte) (adressToWrite >>> 8), data[0], data[1], data[2],
            data[3] };
}
```

Abbildung 5.1: singleBlockWritecommand()

So sieht beispielsweise eine Funktion um ein Byte-Array Command zu generieren, dass 4 Byte Daten mittels Page-Writing[28] Verfahren auf einem beliebigen Block XY schreiben kann. 0x0A und 0x21 sind Bit-Muster, die den Chip vorkonfigurieren und ihn sagen, was er zu tun hat.

[28]Page Writing ist ein Verfahren um mehrere Dateneinheiten auf einem Schlag mit einem Block (Page) abzuarbeiten. Dadurch erhält man i.d.R. einen großen Geschwindigkeitsvorteil.

5.1.3 JNotify

Auf der Seite der BlackBox Anwendung hat sich die Frage nach technische Umsetzung der Logfiles Überwachung gestellt. Prinzipiell wäre es möglich mit busy waiting (aktives Warten) Verfahren zu realisieren, weil es am einfachsten zu implementieren ist. Bei busy waiting würde der Überwachungs-Thread jede x-Sekunden die Logfiles auf Änderungen abtasten. Wenn sich die Logdatei geändert hat, dann wird eine Sicherungsaktion durchgeführt. Anderseits ist es ein sehr ressourcenhungriges Verfahren und würde das System so stark auslasten, dass die systemseitige Sicherung der Logfiles schlimmstenfalls verzögert wäre. Da die Logeinträge so schnell wie möglich nach dem Logereignis gesichert werden müssen, würde eine Verzögerung durch busy waiting nicht in Frage kommen. Eine weitere Möglichkeit für die Realisierung wäre ein Watch auf Linux-Ebene auf eine Datei zu setzen, damit die Änderungen vom Linux-System überwacht werden. Dieser Lösungsansatz hat mehrere Nachteile:

- Um ein Watch auf Systemebene zu setzen müsste man die Java-Welt verlassen und ein C-Projekt nur für diese Zwecke anlegen
- Eine spätere Portierung auf einen anderen Betriebssystem, beispielsweise Microsoft Windows, wäre nicht mehr möglich

Letztendlich kam ein Open Source Projekt namens **JNotify**[29] zum Einsatz. Es ist, wie der Name schon sagt, ein Java Projekt um genau das Problem zu lösen. JNotify ermöglicht die Dateiüberwachung in den Java Source-Code einzubinden. Es ist unter Linux, Windows und MacOS lauffähig. Außerdem ist es mit Watches realisiert. D.h. es werden keine Systemressourcen während der Überwachung verschwendet. Damit sind die Nachteile aus den ersten beiden Realisierungsansätzen kompensiert.

5.1.4 Testing

Bei der Entwicklung von BlackBox System wurde auf das klassische Testverfahren mit JUnit verzichtet, weil die Implementierung teilweise viel Low-Level Programmcode enthielt. Außerdem war die Gesamtlösung auf mehrere Systeme verteilt: BlackBox Anwendung unter Linux und Reader-App auf Android-Smartphone mit M24LR Board dazwischen. Ein verteiltes System kann man nur ganz schwer mit automatisierten JUnit Tests prüfen, daher war es sinnvoll sich ein anderes Verfahren zu überlegen. Da die Entwicklung iterativ verlief, fiel die Entscheidung auf kleine Tests der jeweiligen Komponenten nach ihrer Implementierung, sowie Testing der Gesamtlösung nach Integration der Komponenten.

Beispiel: In Kapitel (5.1.2) „Herangehensweise" wurde bewiesen, dass das Android-Smartphone wie erwartet mit LRiS64K Chipkarte kommuniziert in dem es beliebige Byte-Ketten auf den Eeprom ablegen und wieder auslesen kann. Die T-Komponente NfcConnector wurde wiederverwendbar ausgelegt, damit sie nicht nur die Daten über den Smartphone

[29] JNotify von Omry Yadan: `http://jnotify.sourceforge.net`

auslesen, sondern auch die BlackBox Anwendung beim Schreiben der Daten ins Eeprom unterstützt.

Nach der Implementierung und erfolgreichen Test von NfcConnector wurde geprüft ob das M24LR Board sich wie eine LRiS64K Chipkarte verhält. Der Test bezüglich M24LR Funktionalität wurde erfolgreich abgeschlossen. Damit war noch ein Iterationsschritt erfolgreich abgeschlossen. In diesem Schritt wurde bewiesen, dass die angestrebte Lösung mit vorgeschlagene Hard- und Softwarearchitektur funktionieren kann.

5.2 Methoden / Besonderheiten der Realisierung

In diesen Abschnitt werden die Methoden der Realisierung, sowie ihre Besonderheiten vorgestellt. Zuerst werden die Schnittstellen und ihre Eigenschaften von dem M24LR Board genauer beschrieben. Danach wird die „Behandlung der Persistenz” beleuchtet, um die technische Realisierung der EEPROM Verwaltung besser nachvollziehen zu können. Im Abschnitt „Auseinandersetzung mit Polling” wird erklärt wieso Polling bei der Implementierung nicht zum Einsatz kam. Am Ende der Kapitel werden die größten Schwierigkeiten bei der Realisierung der Arbeit beschrieben.

5.2.1 Schnittstellen der BlackBox

Bei der Kommunikation zwischen den Benutzer-Lesegerät und BlackBox System wird eine kabellose NFC-Verbindung eingesetzt. Die NFC-Verbindung zwischen den Nexus S Smartphone und M24LR Board ist relativ langsam. Es liegt an dem in Nexus S eingesetzten NFC-Chip. Da die Daten der BlackBox werden in der Regel erst nach einem Ausfall ausgelesen, handelt es sich um einen sehr seltenen Vorgang. Nichtdestotrotz bleibt die NF4 (Zeitverhalten) Anforderung erfühlt, weil die Daten unter ungünstigen Bedingungen nach maximal einer Minute ausgelesen werden.

Die Verbindung zwischen den PC und BlackBox wird über ein USB Kabel realisiert. Die Kernkomponente M24LR der BlackBox kann entweder per I2C oder NFC Schnittstelle angesteuert werden. Ein handelsüblicher PC hat weder eine NFC, noch I2C Schnittstelle. Da nur die I2C Schnittstelle funktionale Anforderung F2 (Funktionalität, Richtigkeit) wegen ihrer Schnelligkeit erfüllen kann, müsste ein USB-I2C Adapter her um die BlackBox an den Computer per Kabel anzuschließen. Nach Rücksprache mit den Entwicklern der Stollmann GmbH fiel die Entscheidung auf das Aardvark Host-Adapter von Totalphase, weil es um einen Entwicklerwerkzeug mit sehr guter Dokumentation der Schnittstellen und Multiplattformfähigkeit handelt. Nach der Installation der USB Treiber war der Adapter sofort einsatzbereit. Totalphase liefert neben den Gerät-Treibern auch noch eine kleine Sammlung C-Klassen um den Adapter zu steuern.

5.2.2 Behandlung der Persistenz / EEPROM

Nachdem die die C-Klassen per Java Nativ Call an das Projekt gebunden waren, bestand die Herausforderung darin, die Daten persistent abzulegen. Das Eeeprom von M24RL kann die Daten nur in Form von einzelnen Bytes abspeichern. Daher war es nötig die Logfiles in Row-Text umzuwandeln und jedes einzelnes UTF8 Zeichen in ein Byte umzuwandeln. In Form von Byte-Arrays ließ sich der Text prima auf dem Eeprom ablegen. Umgekehrt kann es genauso wieder ausgelesen und zurück in Text Zeichenketten umgewandelt werden.

Da der Chip von M24LR mindestens 8 ms für jeden einzelnen Lese- und Schreibvorgang benötigt, würde das komplette Auslesen bzw. Beschreiben von dem Eeprom deutlich mehr als eine Minute: 8192 Bytes Eeprom * 8 ms = 65536 ms = ca. 66 Sekunden. Dies würde gegen die funktionale Anforderung F2 (3.3) verstoßen: „Die letzten Ereignisse müssen sofort nach dem Auftreten gesichert werden". Das Studium von Datenblatt des M24LR brach einen entscheidenden Tipp: M24LR unterstützt Page-Writing und Page-Reading. Bei einem Page Schreib- bzw. Lesevorgang kann ein Block (Page) an Daten in einem Rutsch verarbeitet werden. Die maximale Page-Größe von M24LR Dual Interface Board beträgt 4 Byte. Damit wäre eine theoretische Beschleunigung von Faktor 4 möglich. Ein kurzer Test hat die Vermutung bestätigt: mit Page-Writing und Reading hat sich die Schreib- bzw. Lesegeschwindigkeit tatsächlich beinahe vervierfacht. Diese Erkenntnis wurde dann entsprechend in der Implementierung umgesetzt. So sieht beispielsweise Page-Writing aus (ein klein Ausschnitt):

```
public boolean write (int offset, byte[] data)
{
  // this code should do ack-polling and page-writes!
  byte[] command = new byte[6];

  for (int i=0; i<data.length; i +=4)
  {
    command[0] = (byte)(((i+offset) >> 8) & 0xff);
    command[1] = (byte)(((i+offset) >> 0) & 0xff);
    command[2] = data[i];
    command[3] = data[i+1];
    command[4] = data[i+2];
    command[5] = data[i+3];


    try
    {
            mI2c.i2c_write (M24LR64_MEM_ADDRESS, aardvark.AA_I2C_NO_FLAGS, command);
    }
```

Abbildung 5.2: Page-Writing

Wie man den Codeausschnitt entnehmen kann, wird ein Array mit Daten in jeweils 4 Kilobyte große Blöcke geteilt und in einem Schritt auf dem Eeprom gesichert. Die ersten 2

Bytes bilden die Eeprom Adresse in 4-er Schritten ab. Die restlichen Bytes 2 bis 5 enthalten 4 KB Daten, die in die Persistenz geschrieben werden.

Eine weitere Eigenheit bei der Umsetzung von Persistenz Behandlung stellt Round-Robin Schreibverfahren dar. Das Eeprom Speicher verträgt ca. 1 Million Schreibvorgänge pro 4K Sektor. Deswegen musste man sicherstellen, dass einzelne Sektoren z.B. am Anfang des Eeprom nicht übermäßig beansprucht werden. Mit Round-Robin stellt man sicher, dass alle Sektoren gleich oft beschrieben werden. Dies geschieht in dem das Eeprom erstmals fortlaufend bis zum vollen Speicherkapazität beschrieben wird und dann wieder von vorne beginnt. Dabei werden die älteste Daten mit neusten überschrieben, damit man möglichst große Bandbreite an neuesten Daten vorenthalten kann.

5.2.3 Auseinandersetzung mit Polling

Im Abschnitt (5.2.2) wurden die Eigenheiten von dem M24LR Chip beschrieben. Dies ist nötig um funktionale Anforderung F2 zu erfüllen. Neben den bereits implementierten Page-Writing existiert eine weitere Möglichkeit um den Lese- und Schreibvorgängen eine zusätzliche Beschleunigung beizubringen: Polling, auch als „aktives Warten” genannt. Dabei wird es an dem M24LR Chip in einem Abstand von 1-2 Millisekunden die nächste Anfrage gestellt. Die Anfrage wird in vordefinierten Zeitabständen an den Chip so oft geschickt bis der Chip die Anfrage mit einem OK Flag beantwortet. Dann gilt sie als abgeschlossen.

Bei der Abarbeitung von Lese- und Schreibbefehlen ist die Persistenz der Flaschenhals. Sie benötigt bis zu 8 Millisekunden um eine Byte-Sequenz auszulesen oder zu schreiben. In diese Zeit hat der Chip selbst keine weiteren Aufgaben. Daher würden mehrere Anfragen in einem kurzen Zeitabstand das M24LR Board nicht überlasten. In den Spezifikationen von dem M24LR Board sind 8 ms als eine Verzögerung für die Abarbeitung von einem Befehl angegeben. Die 8 ms stellen einen Worst Case dar. In der Praxis ist der Chip in der Regel viel schneller mit Verarbeitung eines Befehls fertig. Man kann nur mit mehreren Anfragen in 1-2 ms Abständen den tatsächlichen Dauer von dem Vorgang rauszufinden, weil es keine andere Möglichkeit gibt um rauszufinden ob das Eeprom fertig mit dem Lesen oder Schreiben ist.

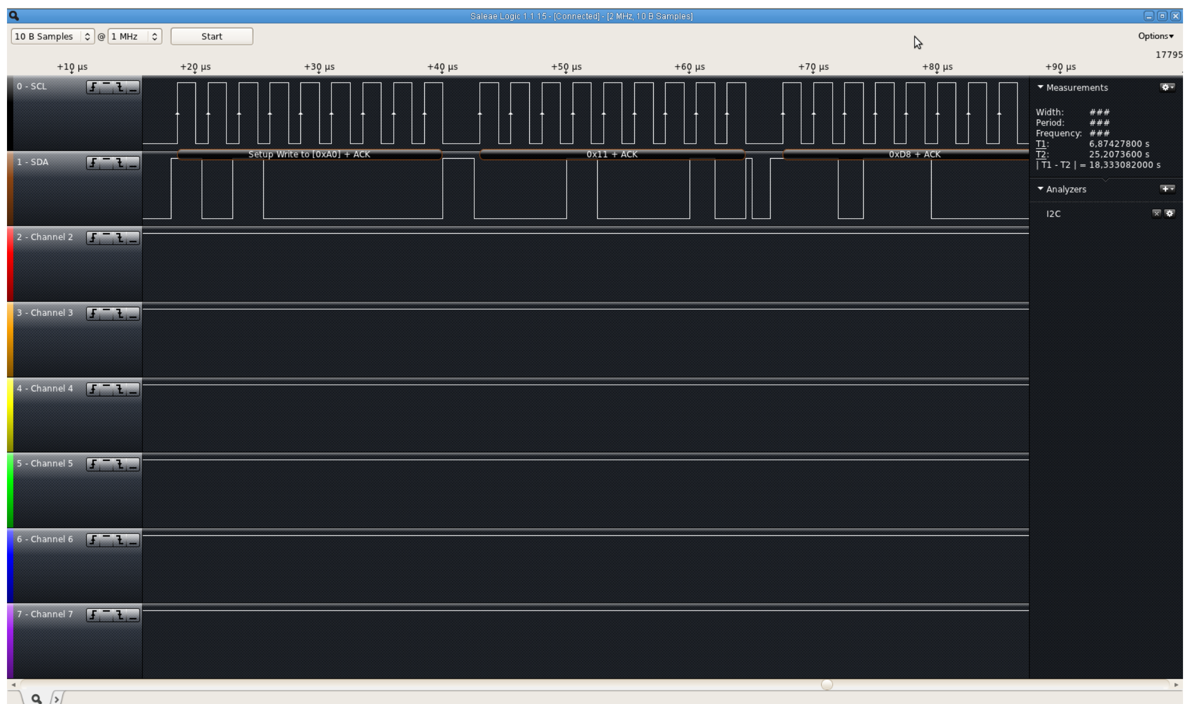


Abbildung 5.3: I2C Bus Traffic

An diese Stelle eine mit Logic-Analyser erstellter Analysis des Datenverkehrs auf dem I2C Bus während eines Schreibvorganges. Man kann gut erkennen, dass auf dem 1 SDA Bus die meiste Zeit Funkstille herrscht. Mit Polling könnte man die Zeitabstände zwischen den Anfragen verkürzen.

Weitere Recherchen haben ergeben, dass die Minimierung von Zeitbedarf für einen Schreibvorgang einer Zeile Text nur um wenige Millisekunden verkürzen würde. Daher war die Idee mit Polling verworfen, weil der Zeitaufwand für die Implementierung in kein Verhältnis zum Nutzen gestanden hätte.

5.2.4 Schwierigkeiten bei der Implementierung

Die Entwicklung von der Endlösung verlief, wie üblich, nicht komplett reibungslos. Dieses Unterkapitel beschreibt die größte Stolpersteine, die auf dem Weg zur fertigen Lösung zum größten Zeitvernichter wurden.

Nach der Einrichtung von JNotify (5.1.3) unter Linux lief das Framework nicht wie erwartet. Ein Grund hierfür ist, dass es keine Dokumentation zum Installationsvorgang existiert. Es gibt lediglich ein Paar Code-Samples sowie eine eher kurz gehaltene FAQ Sammlung. Bei jedem Versuch JNotify Thread zu starten, brach es mit eine unverständliche Meldung ab. Durch eine lange Recherche in verschiedenen Foren wurde ein Ansatzpunkt für die Feh-

lersuche gefunden: ein Jnotify.so Treiber soll nicht sauber laufen. Der Treiber war allerdings im Installationsverzeichnis vorhanden und intakt. Daher kam es zur Vermutung, dass JNotify den Treiber an eine ganz andere Stelle sucht. Auch nach dem die o.g. Datei ins Systemordner für Treiber /var/lib kopiert wurde, gab es keine Besserung. Erst die Änderung von der Berechtigung der Datei auf 755 (read & execute mode unter Linux) brauch einen lang erwarteten Durchbruch. Danach lief JNotify einwandfrei.

Eine weitere Reibungsstelle gab es bei der Einbindung von Aardvark C-Bibliotheken und Aardvark-Treiber. Um die C-Funktion für die Java-Anwendung zur Verfügung zu stellen müssen die Treiber-Dateien bei Java VM eingebunden sein. Hierfür darf kein klassisches Java-Classload oder Einbindung in die Projektbibliotheken genutzt werden. Stattdessen müssen die Treiber mit System.load geladen werden. So sieht ein System.load aus:

```
00    private native boolean NativeOpen    (int deviceId);
01    private native boolean NativeClose   (int handle);
02    static private native int[] NativeFindDevices ();
03
04    static
05    {
06      System.load("/home/vitali/i2c-java-2/libaa_jni.so");
07    }
```

Abbildung 5.4: Java System.load Ausschnitt

Die Implementierung von NFC Kommunikation hat ebenfalls einige Tücken vorzuweisen. Die NFC Schnittstelle über die Luft wird auf sehr hohen, für Störungen anfälligen Frequenzen realisiert. Daher kann es schon sein, dass bei der Übertragung ein oder mehrere Pakete verloren gehen. Der Programmierer ist für die Realisierung der Fehlerbehandlung und Korrektur selbst zuständig. Das NFC Protokoll bleibt dabei außen vor. In diesem Projekt war es ausreichend eine vierfache Wiederholung der Schreib- bzw. Lesevorganges pro Paket in Fehlerfall einzubauen. Falls den NFC-Chip nach 4 Leseversuchen ein Sektor immer noch nicht auslesen kann, fehlt am Ende eventuell ein Buchstabe im Logtext. So ein Fehler ist durchaus verzeihbar, weil die Lesbarkeit der Logfiles dadurch nicht merklich erschwert wird.

Im großen Ganzen verlief die Entwicklung doch relativ problemlos nach den o.g. Anlaufschwierigkeiten.

5.3 Ausblick auf mögliche Erweiterungen

Bei der Realisierung dieser Arbeit waren nicht alle mögliche Funktionen und Features implementiert, weil es die vorgegebenen zeitlichen Rahmen überschreiten würde. Daher findet man an diese Stelle einen kurzen Überblick über die möglichen Erweiterungen der Arbeit.

5.3.1 Portierung auf ein anderes Betriebssystem

Der durchgehende Einsatz von Java als Programmiersprache erlaubt es relativ einfach das bestehende BlackBox System auf ein anderes Betriebssystem zu portieren. Hierfür muss eine passende Version von JNotify installiert werden (Windows, MacOS oder Linux) und einige Systempfade angepasst. Ferner müssen ebenfalls die Aardvark in eine passende Version in das System eingebunden. Dann wäre das BlackBox System auch unter MacOS und Windows einsetzbar.

Im Fall der Portierung wäre beispielsweise keine Anpassung der Reader-App notwendig, weil das BlackBox System unabhängig von Betriebssystem die Log-Daten im gleichen Format abspeichert. Eine grafische Benutzeroberfläche für die Bedienung von der BlackBox Anwendung wäre ebenfalls eine wünschenswerte Erweiterung, weil die Anwendung derzeit über die Entwicklungsumgebung gesteuert wird.

5.3.2 Erweiterung der Reader-App

In der aktuellen Version weist die Reader-App nur rudimentäre Funktionen auf. Es wäre möglich die App um einige praktische Funktionen zu erweitern. Beispielsweise Speicherung von den ausgelesenen Logs auf die SD Karte von dem Smartphone als Datei oder ein Filter für die Logs um bestimmte Ereignisse schneller zu finden. Außerdem ist es vorstellbar die Benutzeroberfläche auf ihre Benutzbarkeit (Usability) zu testen um in Disaster-Fall eine schnellere Bedienung von der App zu ermöglichen.

5.3.3 Überwachung von anderen System-Parameter

Die Blackbox Anwendung in ihre aktuelle Version stütz sich ausschließlich auf die Protokollierung von den letzten Logereignissen. Eine sinnvolle Änderung um noch mehr Indizien für eine mögliche Ausfallursache zu lokalisieren würde die Überwachung von einigen Systemparametern darstellen. Bei so einer Erweiterung würde das BlackBox System andere Systemparameter wie beispielsweise Hardware-Temperatur überwachen und persistent ablegen. Falls so einer Erweiterung in Zukunft implementiert sein sollte, wäre eine Vergrößerung der Persistenz unumgänglich, weil es dann deutlich mehr Daten anfallen würde. Aktuell stehen 64 kBit Eeprom Speicher zur Verfügung.

Kapitel 6

Abschluss

6.1 Zusammenfassung

Das Ziel von der Bachelorarbeit war ein Konzept für die Post-Disaster Analyse eines Linux Systems zu erstellen und umzusetzen. Dabei sollte bei der Realisierung die neuste Technologie Near Field Communication, sowie das weit verbreitete Mobile OS Android verwendet werden. Die ausgiebigen Recherchen haben ergeben, dass es bisher kein vergleichbares Konzept für Post-Disaster Analyse auf dem Markt gibt. Das Konzept für die Realisierung konnte aus diesem Anlass frei festgelegt und anschließend entwickelt werden ohne unfreiwillig ein bereits existierendes Produkt oder Konzept zu kopieren.

In der Analyse-Phase wurden die essenziellen Anforderungen gestellt um bei den wichtigsten Praxis relevanten Anwendungsfällen eine funktionierende Analyse zu erhalten und anschließend erfolgreich umgesetzt.

Die im Entwurf & Architektur eingesetzte Architekturen und Design Pattern haben sind bei der Umsetzung als geeignet erwiesen. Die für die Umsetzung vorgeschlagene Bausteine wie z.B. M24LR oder das Nexus S Smartphone haben sich bewährt, auch wenn die Entwicklung nicht immer reibungslos verlief, wie man dem Kapitel (5.2.4) „Schwierigkeiten bei der Entwicklung” entnehmen kann.

Mit der Realisierung wurde bewiesen, dass man mit dem BlackBox Konzept die letzte Log-relevante Systemereignisse sehr zeitnah in eine nicht-flüchtige Persistenz sichern kann um sie später bequem mit einem handelsüblichen Android Smartphone per NFC auszulesen und zu analysieren

6.2 Ausblick

Das durch die Bachelor-Arbeit erlangte Wissen kann durchaus zur Erstellung eines kommerziellen Produktes genutzt werden. Allerdings kann es bei einem kommerziellen Einsatz

zu Schwierigkeiten kommen, weil während der Umsetzung dieser Bachelor-Arbeit die Firma Apple Inc. Ende 2011 ein Patent auf eine ähnliche Lösung zum Auslesen von PC-Daten mit einem NFC-Smartphone angemeldet hat. Im wissenschaftlichen Bereich kann ein funktionierendes Konzept für die NFC Kommunikation sowie der Ansatz wichtige Daten auf eine externe Persistenz, auf die ein Betriebssystem in der Regel kein Zugriff hat, von Nutzen sein.

Die Firma Stollmann E+V GmbH, die diese Bachelor-Arbeit betreut und gefördert hat, kann sich vorstellen das Ergebnis aufzuarbeiten und bei Messen, Ausstellungen oder Kundenkontakten zur Demonstration eines möglichen Einsatzzweckes von NFC Technik zu zeigen.

Anhang A

Anhang

A.1 CD

Inhalt

- Sourcecode von BlackBox Anwendung
- Sourcecode von Reader-App
- readme.txt
- Bachelor-Arbeit in PDF Form

Literaturverzeichnis

[Becker und Pant 2010] BECKER, Arno ; PANT, Marcus: *Android2 - Grundlagen und Programmierung*. 2. Auflage. dpunkt.verlag GmbH, 2010

[Langer und Roland 2010] LANGER, Josef ; ROLAND, Michael: *Anwendungen und Technik von Near Field Communication (NFC)*. Springer Verlag, 2010

[Mosemann und Kose 2009] MOSEMANN, Heiko ; KOSE, Matthias: *Android - Anwendungen für das Handy-Betriebssystem erfolgreich programmieren*. Carl Hanser Fachbuchverlag, 2009

[Oliver u. a. 2012] OLIVER, Adam ; GANAPATHI, Achana ; ; XU, Wei: Advances and Challanges in Log Analysis. In: *Published on ACM Queue Digital Library* (2012)

[Sarstedt 2011] SARSTEDT, Prof. Dr. S.: *Architektur von Informationssystemen*. 2011. – Architektursichten zur Kommunikation

[Schlosser 2009] SCHLOSSER, Joachim: *Wissenschaftliche Arbeiten schreiben mit LATEX*. 3. Auflage. mitp, 2009

[Skogberg 2010] SKOGBERG, Benny: *Android Application Development*. skogberg.eu/android/ : -, 2010

[STMicroelectronics 2010] STMICROELECTRONICS: *How to manage simultaneous I2C and RF data transfers. Rev. 3*, 2010

[STMicroelectronics 2011] STMICROELECTRONICS: *Simple NDEF Exchange Protocol*, 2011

[Wolf 2009] WOLF, Jürgen: *Linux-UNIX Programmierung*. 3. Auflage. Galileo Computing, 2009

Abbildungsverzeichnis

2.1 Architektur von Android 4

3.1 Use Case - lauffähiges System 16
3.2 Use Case - Disaster 17

4.1 Blackbox-Kontextabgrenzung 21
4.2 WhiteBox 23
4.3 BlackBox Laufzeitsicht 25
4.4 Reader-App Laufzeitsicht 26
4.5 Blackbox Verteilungssicht 28

5.1 singleBlockWritecommand() 35
5.2 Page-Writing 38
5.3 I2C Bus Traffic 40
5.4 Java System.load Ausschnitt 41

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 10.05.2012 Vitali Eylin