

**Fachhochschule
Münster University of
Applied Sciences**



**Totzeitfreies Roamingprotokoll zur Fernsteuerung von
Mobilmaschinen in WLAN-Netzen**

Dennis Borgmann

Diplomarbeit

im Studiengang *Maschinenbauinformatik*

am Fachbereich Maschinenbau

der Fachhochschule Münster/Steinfurt

bei Prof. Dr.-Ing. Harald Bösche

Die Arbeit wurde erstellt in Kooperation mit
Embigence GmbH
49549 Ladbergen

Betreuer: Dipl.-Ing. Tom Stöveken (Embigence)

Ladbergen, März 2007

Danksagung

Mein herzlicher Dank geht an die Herren Prof. Dr.-Ing. Harald Bösche, dem Betreuer seitens der Fachhochschule, und Dipl.-Ing. Tom Stöveken als Betreuer seitens der Firma Embigence. Besonders Herr Stöveken stand mir täglich mit Rat und Tat vor Ort zur Seite und brachte mir die faszinierende Welt der Funknetzwerke näher.

Des Weiteren gebührt Herrn Dipl.-Ing. Christoph Müller Dank, der es mir ermöglichte, diese Diplomarbeit bei Embigence zu absolvieren. Nicht zu vergessen sind Herr Dipl.-Ing. Jürgen Gröniger vom Laserzentrum der FH Münster, sämtliche Mitarbeiter von Embigence und vor allem meine Freundin Stefanie, die mich von Anfang bis Ende dieser Diplomarbeit unterstützt hat, treibende Kraft bei der Fehlersuche war und jederzeit zu mir stand.

Sperrvermerk

Diese Diplomarbeit enthält interne Informationen und Betriebsdaten, die ohne Einwilligung nicht an Dritte weitergegeben werden dürfen. Es ist darauf zu achten, dass die Arbeit nicht an Unbefugte ausgehändigt und nicht vervielfältigt wird.

Anmerkung: Verwendete Fachbegriffe sind neben den Erläuterungen im laufenden Text in Kurzform in Anhang A erläutert und bei der ersten Verwendung im laufenden Text **fett** gedruckt. Linux-Systembefehle sind durch

eine folgende eingeklammerte Ziffer, die den Abschnitt innerhalb des Linux-Onlinehandbuches kennzeichnet, hervorgehoben.

Inhaltsverzeichnis

1 Einleitung	7
2 Problemstellung	8
2.1 Stand der Technik	8
2.2 Lösungsansatz	9
3 Verwendete Hardware	11
3.1 Embedded PCs	11
3.1.1 WRAP 1e203	11
3.1.2 mycable XXS1500	12
3.1.3 Lippert Cool FrontRunner	12
3.1.4 EmbiCube	13
3.2 WLAN-Netzwerkkarten	13
3.2.1 Intel PRO/Wireless 2200BG	14
3.2.2 Atheros Karten	14
3.3 Accesspoints	15
3.3.1 Linksys WRT	15
3.3.2 OpenWRT	16
4 qemu - der Computer im Computer	17
4.1 Motivation für den Einsatz einer virtuellen Maschine	17
4.2 qemu in der Anwendung	18
5 WLAN - ein kurzer technologischer Überblick	21
5.1 Sendefrequenzen	21
5.2 Einfache Managementroutinen	23
5.2.1 Authentifikation	24
5.2.2 Assoziation	24
5.2.3 Deauthentifikation	25
5.2.4 Disassoziation	25
5.3 Beacons	26

INHALTSVERZEICHNIS

5.4	Der IEEE802.11-Header	27
6	Userland-Software mit ipw2200 und MadWiFi	32
6.1	Unzulänglichkeit des Linuxtreibers ipw2200	32
6.2	Roaming mit MadWiFi	33
6.2.1	Umassoziation mit ioctl(2)-Befehlen	34
6.3	Die Netzwerkkarte im Monitormodus	36
6.3.1	Vorteile des Monitormodus	36
6.3.2	Nachteile trotz Monitormodus	37
6.3.3	Aktivierung des Monitormodus	39
7	Einrichten eines Tunnels zwischen Netzwerkkarte und WLAN-Karte	41
7.1	Injizieren von Datenframes	41
7.2	Tunneln von Datenframes	42
8	Roaming - der prinzipielle Vorgang	45
8.1	Technische Erläuterungen	45
8.2	Totzeitfreies Roaming?	47
9	ROAMEO	48
9.1	Lesen auf der virtuellen Netzwerkkarte	52
9.2	Schreiben auf der WLAN-Karte	53
9.3	Lesen auf der WLAN-Karte	54
9.4	Schreiben auf der virtuellen Netzwerkkarte	55
10	Testergebnisse	58
10.1	Höherer Aufwand pro Frame im WLAN	58
10.2	Messung von Rundlaufzeiten ohne Roaming	59
10.2.1	Round-Trip-Time Messsoftware	59
10.2.2	RTT-Ergebnisse lastfrei	60
10.2.3	RTT-Ergebnisse belastetes Medium	62
10.3	Messung von Rundlaufzeiten mit Roaming	64
10.3.1	Roaming mit MadWiFi-0.9.2.1	64
10.3.2	Roaming mit roameo	66
11	Implementierung von Treiberfunktionen	71
11.1	Treibererweiterung in der Theorie	71
11.2	Die Schritte zur Treibererweiterung	73
12	CAD-Zeichnungen von Präsentationsmustern	76

INHALTSVERZEICHNIS

13 Ausblick - was in Zukunft noch getan werden kann	77
A Glossar	78
B Quellcode	84
C CAD-Zeichnungen	102

Kapitel 1

Einleitung

In der vorliegenden Diplomarbeit wird ein Verfahren zur nahtlosen Datenübertragung in Wireless Local Area Networks (WLANs) beim Wechsel von einem Empfangsbereich in den nächsten vorgestellt. Zur erfolgreichen Implementation dieses sogenannten **Roamings** werden die Techniken der Netzwerk tunnel, einer im Monitormodus betriebenen WLAN-Netzwerkkarte und hiermit verbundenes **Frameinjizieren**, virtuelle Maschinen und das eigentliche Verfahren eines Roamings erläutert. Verwendung finden hierbei der Linux ipw2200-Treiber für **Intel**-Karten und der für **Atheros** WLAN-Netzwerkkarten entwickelte MadWiFi-Treiber in einer Linux-Umgebung. Abschließend wird darüber hinaus auf die Implementierung eines Treiberbefehls für MadWiFi eingegangen.

Das Ziel dieser Arbeit bestand darin, die bestehenden Möglichkeiten hinsichtlich der Übergabe von **Stationen** in einem Funknetzwerk nach **IEEE802.11**-Standard zu erweitern und die Totzeit hierbei zu minimieren. Im konkreten Anwendungsfall fährt eine mobile Maschine in einem Bergwerk entlang einer mit **Accesspoints** ausgestatteten Strecke und übermittelt zu Überwachungszwecken via WLAN das Bild einer ammontierten Kamera. Gibt es hier zu große Ausfallzeiten in der Übertragung der Daten, stoppt die mobile Maschine.

Anwendung finden wird das beschriebene Verfahren in Präsentationen für die Firma LKAB, welche in ein groß angelegtes Projekt in Kiruna/Schweden involviert ist und für die Firma Atlas Copco, die bereits seit längerem Kunde von Embigence ist. Bei diesen Präsentationen werden EmbiCubes eingesetzt werden, für deren Auslegung insbesondere bei der Wahl der entsprechenden WLAN-Komponenten Sorge getragen werden musste.

Kapitel 2

Problemstellung

2.1 Stand der Technik

Bewegt sich eine mobile Maschine, welche mit einem Rechner inklusive WLAN-Karte ausgestattet ist, entlang einer Strecke von Accesspoints, so bewegt sie sich in Empfangsbereiche von Accesspoints hinein und auch aus diesen Bereichen wieder heraus. Eine Übergabe des Clientrechners von einem Accesspoint zum anderen, mit dem Ziel, die Datenkommunikation über ein drahtloses Medium fortbestehen zu lassen, bezeichnet man als „roamen“ oder „roaming“. Wird das Signal des Accesspoints, über den die Datenübertragung abgewickelt wird, schwächer, so wäre es prinzipiell sinnvoller, einen Wechsel auf den stärksten verfügbaren Accesspoint vorzunehmen: Es sollte zu dem stärksten Accesspoint „geroamt“ werden.

Dieses Vorgehen in der zuvor beschriebenen Art und Weise ist in der zum Zeitpunkt der Diplomarbeit vorliegenden Version 0.9.2.1 des MadWiFi-Treibers nicht implementiert. Das übliche Verhalten dieses Treibers sieht es vor, seine alte Assoziation so lange beizubehalten, bis gar keine Datenübertragung mehr möglich ist. Erst in diesem Fall wird die alte Assoziation abgebaut, ein neuer Scan auf allen Kanälen durchgeführt und dann ein gefundener Accesspoint neu assoziiert. Dieses Vorgehen ist langsam, es tritt ein hoher Datenverlust auf und die Datenrate wird mit schwächer werdendem Signal verringert, um somit noch möglichst lange die Assoziation des alten Accesspoints aufrecht zu erhalten. Da hier die Datenraten auf ein Minimum abgesenkt werden und sogar Daten verloren gehen, ist kein zufriedenstellendes Roaming mit diesem Treiber möglich.

2.2 Lösungsansatz

Um dieses Verhalten zu verbessern, muss permanent bekannt sein, welche Accesspoints sich im Empfangsbereich befinden und wie stark ihr jeweiliges Signal ist. Befindet sich ein stärkerer Accesspoint in Funkreichweite, so ist es als sinnvoll anzusehen, zu diesem zu wechseln und ab diesem Zeitpunkt den gesamten Datentransfer nur noch über diesen Accesspoint abzuwickeln. Hierbei ist zu berücksichtigen, wie lange es dauert, einen Accesspoint zu wechseln. Gefährdet diese Zeit nicht die verlustfreie Datenübertragung, so ist zusätzlich zu überprüfen, ob es Sinn ergibt, zu diesem neuen Accesspoint zu wechseln oder ob das alte Signal noch gut genug ist, um die Daten weiterhin über den alten Accesspoint zu senden. Nur in dem Fall, dass ein solcher Wechsel des Accesspoints schnell genug vonstatten geht und weder Datenverlust noch unvertretbar hohe Roamingzeiten auftreten, soll ein sofortiger Wechsel des Accesspoints vollzogen werden, sobald ein Accesspoint in seiner Signalstärke einen anderen übertrifft.

Dass allein das Kriterium, ob Datenverlust auftritt oder nicht, kein ausschlaggebendes Kriterium für ein Roaming sein darf, lässt sich anschaulich an dem Beispiel einer Videoübertragung erläutern. Auf einer mobilen Maschine sei eine Kamera platziert, die ihre Bilder über das WLAN an eine Überwachungsstation sendet, wo die Fahrt der Maschine per Video überwacht wird. Angenommen, ein Roamingvorgang dauert zwei Sekunden und es gibt einen Puffer für die Daten, die während der zwei Sekunden anfallen, die der Client im schlimmsten Fall ohne Verbindung ist. Roamt der Client nun zwischen zwei Accesspoints, so werden nach abgeschlossenem Roaming die Daten aus dem Zeitraum des Roamings an den neuen Accesspoint aus dem Puffer übertragen, allerdings zu spät. Die Applikation, in diesem Fall also eine videoabspielende Software für das Bild der Kamera auf der Maschine, würde möglicherweise die Daten der letzten zwei Sekunden im Schnelldurchlauf anzeigen (dieses Verhalten ist abhängig von der entsprechenden Applikation), effektiv aber würde das Bild während der zwei Sekunden des Roamings „stehen“, da keine Daten am Client auflaufen. Fährt in diesen zwei Sekunden die Maschine vor ein Hindernis, kann der Bediener zu dem Zeitpunkt, da er die Bilder über diesen Unfall erhält, keine Gegenmaßnahmen mehr einleiten: der Unfall ist bereits geschehen.

In solchen Fällen ist es notwendig für die Bedienung und die Funktion der Maschine, dass die Daten rechtzeitig ihr Ziel erreichen.

KAPITEL 2. PROBLEMSTELLUNG

Es gibt also zwei verschiedene Anforderungen an ein Roaming: die Daten müssen

- verlustfrei und
- rechtzeitig, d.h. ohne störende Verzögerung

übermittelt werden.

Kapitel 3

Verwendete Hardware

3.1 Embedded PCs

3.1.1 WRAP 1e203

Zu Beginn der Diplomarbeit stand ein **WRAP** 1e203-System Board der Firma PC-Engines zur Verfügung. Dieses als Routerplattform (WRAP = Wireless Router Application Platform) konzipierte **Motherboard** zeichnet sich durch Stabilität, Kompaktheit und für den Anwendungszweck optimierte Bauelemente, wie beispielsweise eine hinreichend schnelle CPU (233 MHz **AMD** Geode SC1100) oder eine zweckorientierte Hauptspeichergröße (128 MB SDRAM) aus. Daraus resultiert eine geringe Leistungsaufnahme, die bei solchen möglichst wartungsfrei betriebenen Geräten wünschenswert ist.

Diese Plattform wurde gewählt, weil hier ein x86-Prozessor verbaut wurde. Da die eigentliche Softwareentwicklung auf einem Intel Celeron Rechner verrichtet wurde, war eine einfache Portierung möglich. Beide CPUs entsprechen der gleichen **Maschinenarchitektur**, was den großen Vorteil mit sich brachte, dass ein **Crosscompilen** nicht notwendig war.



Abbildung 3.1: Das PC Engines WRAP1e203-Board[20]

Die Abbildung zeigt das PC Engines WRAP1e203-Board. Dieses Board stellte die Grundlage für erste Tests mit der für **Flash-Speicherkarten** ausgelegten Linux-distribution „Voyage Linux“[14] dar. Aufgrund eines bereits im zum Download

verfügbarer Paket enthaltenen Scripts stellt sich hier die Installation als sehr einfach dar. Sämtliche einstellbaren Optionen sind selbsterklärend und ein funktionsfähiges Linux ist somit binnen Minuten auf eine **CF-Card** gebannt.

3.1.2 mycable XXS1500

Neben dem WRAP-Board stand zu Beginn dieser Arbeit ein kompaktes mycable XXS1500-Gerät für Tests bereit. Dieses Modell ist ebenfalls als Kleinstsystem ausgelegt, arbeitet jedoch mit einer **MIPSel** AMD Alchemy Au1500 CPU.

Hierin verbirgt sich bereits die erste kleinere Hürde, da für die weitere Entwicklung auf dieser Plattform immerzu eine Crosscompiler-toolchain notwendig gewesen wäre. Außerdem stand die MIPSel-Architektur einer eventuell angedachten Portierung auf andere Betriebssysteme im Wege, da beispielsweise unter den BSDs lediglich NetBSD diese Architektur unterstützt. Der Hauptgrund nicht weiter verfolgter Entwicklung auf dieser Plattform war jedoch unzureichende Zusammenarbeit zwischen dem Hersteller und Embigence.



Abbildung 3.2: Der X XS1500 von mycable[10]

3.1.3 Lippert Cool FrontRunner

Das Lippert Cool FrontRunner-Modul ist das Herzstück des letztendlich verwendeten EmFiCubes. Es besteht aus einer **PC/104**-konformen Modulkarte mit einem AMD Geode GX500@1.0W Prozessor (366MHz), 10/100BaseT Ethernet, AC97 on-board Sound, 256MB **DDR SDRAM** und einem **ATA-5 (UDMA-66) EIDE** Interface für CF-Cards.

Für dieses Modul gelten die gleichen Vorteile hinsichtlich der Prozessorarchitektur wie beim WRAP-Board. Auch hier ist in der Regel auf dem Entwickler-PC programmierte Software direkt lauffähig.

Des Weiteren bietet dieses Board noch zwei serielle Schnittstellen, die zwischen RS232 und RS465 umschaltbar sind und eine 10/100BaseT Ethernet-Schnittstelle. Da auch ein direkter SVGA-Monitoranschluss vorhanden ist, eignet sich dieses Modul ebenfalls zur direkten Wartung am Rechner selbst. Es handelt sich praktisch um einen vollwertigen, kompakten, robusten PC.



Abbildung 3.3: Der „Cool Front-Runner“ mit AMD Geode GX500@1.0W[10]

3.1.4 EmbiCube

Innerhalb des EmbiCubes sind sowohl das Lippert Cool FrontRunner-Modul als auch diverse andere PC/104 Plus Komponenten verbaut. So findet sich ein PC/104 Plus-Netzteil V104-5-16, welches eine Eingangsspannung von 8–30 V auf 5 V transformiert. Des Weiteren ist ein PCM-3116 Mini PCI-Adapter mit zwei Steckplätzen der Firma Advantech verbaut, welches Platz für eine Gigabyte GN-WI01HT WLAN-Karte mit Atheros AR5006XS-Chipsatz bietet. Als Gehäuse dient ein Tri-M CT6 Aluminium-„Can-Tainer“.

Je nach Projekt können andere Komponenten gewählt werden. Äußerst robuste WLAN-Karten des Herstellers Elcard [7] mit erweitertem Temperaturbereich finden Anwendung in den LKAB [11] und Atlas-Copco [5] Projekten von Embigence. Sie sind kompatibel zu den hier genutzten Mini PCI WLAN-Karten.

Die Version, welche für die Präsentationen bei LKAB und Atlas Copco im Anschluss an diese Diplomarbeit verwendet wurde, ist in Abbildung 3.4 und Abbildung 3.5 abgebildet.

3.2 WLAN-Netzwerkkarten

Sämtliche genutzte Netzwerkkarten lagen in der Bauform Mini PCI vor, um im EmbiCube eingesetzt werden zu können.

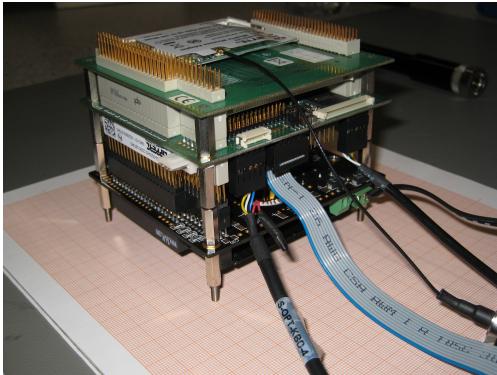


Abbildung 3.4: Die PC/104 Plus-Platinen des EmbiCube



Abbildung 3.5: Der EmbiCube von außen mit Anschlüssen

3.2.1 Intel PRO/Wireless 2200BG

Die Intel 2200BG-WLAN-Netzwerkkarte steckte ursprünglich in einem Fujitsu Siemens Amilo 1425-Laptop, wurde jedoch zu Testzwecken sehr früh ausgebaut und in das WRAP-Board eingesteckt. Es handelt sich, wie der Name schon sagt, um eine IEEE802.11 b/g konforme Karte.

Sie war Bestandteil des ersten Roaming-Tests, welcher in Kapitel 6 näher erläutert wird, wurde jedoch im Anschluss an Versuche mit der Software aus genanntem Kapitel durch Atheros-Karten ersetzt, da der Intel-Treiber laut aircrack-ng [19] keine Frameinjizierung ohne Probleme vom Userland aus ermöglicht.



Abbildung 3.6: Intel PRO/Wireless 2200BG Netzwerkkarte[10]

3.2.2 Atheros Karten

Im Test befanden sich insgesamt drei verschiedene Karten mit Atheros-Chipsatz:

- Wistron CM9
Atheros AR5213A Chipsatz
- Senao EMP-8602
Atheros AR5006X Chipsatz

- Gigabyte GN-WI01HT

Atheros AR5414 aka AR5006XS Chipsatz

Bei der Senao-Karte handelt es sich um eine sogenannte Highpower-Karte. Diese Karten haben eine größere Leistung als herkömmliche Karten und versprechen somit eine größere Sendereichweite. Leider fiel diese Karte bei einem Dauertest aus der Wahl heraus, weil sich nach etwa vier Stunden ein kurzzeitiges Zischen bemerkbar machte und der Rechner sich selbst neu startete. Zurückzuführen ist dies vermutlich auf ein zu schwaches Netzteil, welches die Eingangsspannung von 5V heruntertransformiert auf die für Mini PCI üblichen 3,3V. Die Karte benötigt jedoch 5V, weshalb die Funktion dauerhaft nicht sichergestellt werden konnte. Weitere Tests mit der Wistron Karte verliefen wesentlich stabiler, jedoch wurde diese Karte sowohl in Datendurchsatzraten als auch in der Dauerstabilität von der Gigabyte Karte übertroffen. Folglich fiel letztendlich die Entscheidung für den Dauereinsatz im EmbiCube auf die Gigabyte-Karte.

3.3 Accesspoints

3.3.1 Linksys WRT

Sämtliche Tests wurden mit Accesspoints der Linksys WRT54G-Serie durchgeführt. Es handelt sich hierbei um eine hochwertige Plattform der Cisco Tochterfirma Linksys, die mit Broadcom-Chipsatz bestückt ist.

Im Grunde genommen ist der WRT54GL ein WRT54G (Abbildung 3.7), dessen Betriebssystem ein angepasstes Linux ist. Er beinhaltet 16MB RAM und 4MB Flash-Speicher. Als Prozessor ist ein Broadcom BCM5352EK Mipsel-Chip verbaut, der auf 200MHz getaktet ist. Für den Funkverkehr dieses Gerätes zeichnet sich ein Broadcom BCM2050KML-Gerät verantwortlich, das an zwei RP-TNC Antennenstecker angeschlossen ist. Ein 5-Port-Switch ist standardmäßig verbaut und ermöglicht so die problemlose Integration in bestehende LANs.[13]



Abbildung 3.7: Der Linksys WRT54G[9]

3.3.2 OpenWRT

Als Firmware der Wahl kann bei den benutzten Linksys-Routern OpenWRT genannt werden. Es handelt sich hierbei um ein speziell für Router und Accesspoints entwickeltes Linux. Eine erstklassige Beschreibung dieser frei verfügbaren Linux-Distribution liefern die Entwickler selbst auf Ihrer Homepage:

„What makes OpenWrt really unique though is the fact it employs a writable filesystem, so the firmware is no longer a static compilation of software but can instead be dynamically adjusted to fit the particular needs of the situation. In short, the device is turned into a mini Linux PC with OpenWrt acting as the distribution, complete with almost all traditional Linux commands and a package management system for easily loading on extra software and features.“[13]

Auf sämtlichen Accesspoints wurde die OpenWRT-Version „White Russian RC5“ installiert.

Kapitel 4

qemu - der Computer im Computer

4.1 Motivation für den Einsatz einer virtuellen Maschine

Virtuelle Maschinen werden durch Emulation von Hardware erreicht. Emulieren ist ein Begriff aus dem Lateinischen und bedeutet soviel wie „nachahmen“. Emulierte Maschinen sind nachgebildete Maschinen in dem Speicher einer anderen Maschine. Durch die Technik des Emulierens können mehrere virtuelle Computer in einem Leitrechner, dem sogenannten Host, koexistieren.

Da bei manchen Tests ein WRAP-Board mit einer 256MB-Flashcard verwendet wurde, bei anderen der später ausgelieferte EmbiCube mit 512MB-Flashcard, war der vorhandene Platz für Kompilierungen rein physikalisch begrenzt. Hinzu kommt die begrenzte Leistungsfähigkeit der Prozessoren auf dem WRAP-Board oder dem EmbiCube. Bei kleinen Programmen ist es kein Problem, Code auf einem **embedded** Board selbst zu kompilieren (die platzraubende Installation eines Compilers vorausgesetzt), bei größeren Kernelarbeiten oder Treiberänderungen ist es jedoch nicht empfehlenswert, die vergleichsweise kleine CPU zu belasten.

Das Erstellen von Software bietet sich daher auf einem Desktop-Rechner mit weitaus mehr Rechenleistung an. Zur Verfügung stand ein ACERPower mit einer Intel® Celeron® CPU, getaktet auf 2.80GHz und ausgestattet mit 512MB RAM. Um jedoch den Umweg des eher aufwändigen Crosscompilens zu umgehen, besteht die Möglichkeit, sich eine Kopie der Flashcard als Image zu erstellen und so ein softwaretechnisch exakt gleiches System zu schaffen wie auf dem Zielsystem, das dann in einem virtuellen PC ablaufen kann. Für

diese Vorgehensweise bietet sich das frei verfügbare und unter Linux, Microsoft Windows und MAC OSX lauffähige qemu [16] an. Alternativen zu qemu wie etwa Bochs [6] oder Virtual Box [22] wurden nicht näher betrachtet, da bereits umfangreiche Erfahrungen mit qemu gemacht wurden.

Auf diese Art und Weise hat man sowohl die Möglichkeit, alle hardwareunabhängigen Programme, die man auf dem embedded Gerät einsetzen möchte, vorher in sicherer, leistungsfähiger Umgebung zu testen oder eben auch Programme, die auf dem embedded Gerät später ausgeführt werden sollen, zu kompilieren. Sinnvoll ist das Kompilieren in der virtuellen Maschine vor allem deshalb, weil hier eine konfigurierbare Größe des Arbeitsspeichers vorgegeben werden kann und man Zugriff auf die physikalisch vorhandene Festplatte hat. Die Installation von Compilern, Debuggern oder anderer hilfreicher Software stellt somit kein Problem dar.

Ist ein Vorgang auf dem virtuellen System erfolgreich abgeschlossen, so besteht die Möglichkeit, auf den emulierten Computer via Netzwerkinterface zuzugreifen. So ist auf einfache Art und Weise Datentransfer sowohl auf das emulierte System als auch vom emulierten System herunter möglich. Eventuell erstellter Binärkode kann also problemfrei in dem virtuellen Computer gebaut und anschließend auf das Zielsystem kopiert und dort genutzt werden.

4.2 qemu in der Anwendung

Um qemu zu starten, müssen einige Parameter im Programmaufruf mitgegeben werden, die beispielsweise den zu emulierenden Rechner spezifizieren. Aus praktischen Gründen wurde ein kleines Script geschrieben, das die vorzunehmenden Schritte automatisiert ablaufen lässt und so den Start von qemu vereinfacht.

Listing 4.1: Ein qemu-Startscript für die Shell

```

1 #!/bin/sh
2
3  # Script to start developer machine
4  # adjust the networking options to your needs (eth1, tun0 masquerading)
5
6  # use sudo one time, it caches the password for the following commands
7  sudo -p "need password for sudo (root rights):" echo ok
8
9  # load kqemu accelerator, resize shared mem, load module for masquerading
10 # sleep a second
11 # set rights to allow access to accelerator module and network device
12 sudo umount /dev/shm
13 sudo modprobe kqemu major=0
14 sleep 1
15 sudo chmod 0666 /dev/kqemu
16 sudo chmod 0666 /dev/net/tun
17 sudo mount -t tmpfs -o size=272m none /dev/shm
18
19 # start qemu
qemu -m 256 -hda debian_cube_qemu.img -kernel-kqemu -net nic -net tap &

```

Interessant ist in Listing 4.1 unter Anderem die Zeile 13. Hier wird das kqemu-Modul gestartet, ein Beschleuniger für qemu selbst. Fabrice Bellard, federführender Programmierer in der Entwicklung von qemu, schreibt zu diesem Modul auf der qemu-Homepage:

„The QEMU Accelerator Module increases the speed of QEMU when a PC is emulated on a PC. It runs most of the target application code directly on the host processor to achieve near native performance.“[16]

So kann also unter Nutzung dieser speziellen Software ein enormer Leistungsgewinn innerhalb der emulierten Maschine erlangt werden, was vor Allem dann von Interesse ist, wenn beispielsweise ein Treiber oder größere Software kompiliert werden müssen.

Vergleichbare Bedeutung kommt der letzten Zeile innerhalb des Scriptes in Listing 4.1 zu. Hier erfolgt der eigentliche Aufruf von qemu mit seinen Parametern, welche sich in diesem Falle wie folgt aufschlüsseln:

- **-m 256**

Das **-m**-Flag kennzeichnet die gewünschte Größe des emulierten Hauptspeichers, in diesem Fall 256MB.

- **-hda debian_cube_qemu.img**

Ein eingefügtes **-hda** spezifiziert die erste Festplatte, die im System eingebunden werden soll. Die Datei *debian_cube_image.img* ist ein vergrößertes Abbild der CF-Karte für den EmbiCube, liegt im lokalen Verzeichnis und stellt im Prinzip die gesamte physikalische Speicherkapazität für den simulierten PC dar.

- **-kernel-kqemu**

Hier wird schlicht die Nutzung von kqemu als Parameter mitgegeben, was den vollen Virtualisationsmodus für die beste Performance einschaltet. Möglich ist der Einsatz dieses Moduls nur unter der x86-Plattform für emulierte x86-Maschinen.

- **-net nic**

Diese Option erstellt eine virtuelle NE2000-kompatible Netzwerkarte, welche anschließend vom System genutzt werden kann.

- *-net tap*

Abschließend wird hiermit die Nutzung des TAP-Interfaces des Hostrechners mit in die emulierte Maschine einbezogen. Über dieses Interface kann die virtuelle Netzwerkkarte mit einer realen Netzwerkkarte gebrückt („gebridged“) werden und gelangt so in ein möglicherweise angeschlossenes Netzwerk.

Zu allen obigen Punkten sind wesentlich ausführlichere Informationen in der manpage von qemu(1) [15] oder auf der entsprechenden Homepage [16] zu finden.

Kapitel 5

WLAN - ein kurzer technologischer Überblick

5.1 Sendefrequenzen

Mit zunehmender Verbreitung von Netzwerken sowohl in Haushalten als auch in der Industrie wuchs auch der Bedarf an mobiler Kommunikation mit dem Laptop, dem PDA oder gar einer kabellosen Verbindung eines stationären PCs mit einem Netzwerk. Weitere Anwendung finden in der Industrie beispielweise Mobiltelefone, Kameras, Pager oder Handscanner.

1997 wurde die Technik des Funkens über gewisse Frequenzen mit dem Ziel, Daten zu übertragen, von dem Institute of Electrical and Electronics Engineers (IEEE) unter dem Namen IEEE 802.11 standardisiert. Wenig später, im Jahre 1999, kamen die ersten Erweiterungen dieses Standards hinzu: 802.11a und 802.11b. Sie ermöglichen Übertragungsgeschwindigkeiten von 54 Mbit/sec (802.11a) bzw. 11 Mbit/sec (802.11b). Weitere Entwicklungen folgten hierauf und ein Ende der Forschung auf diesem Gebiet ist nicht absehbar.

Generell benutzen IEEE 802.11-kompatible WLANs den Mikrowellen- oder den Infrarot-Bereich zur Datenübertragung. Während sich Mikrowellen aufgrund höherer Reichweite für Anwendungen innerhalb und außerhalb von Gebäuden eignen, werden WLANs im Infrarot-Bereich hauptsächlich innerhalb von Gebäuden für kurze Kommunikationsstrecken genutzt. Handelsübliche WLAN-Karten nutzen den in Abbildung 5.1 abgebildeten Mikrowellenbereich zwischen 2,412 GHz und 2,472 GHz (IEEE 802.11b/g) oder 5,18 GHz und 5,70 GHz (IEEE 802.11a). Jedermann darf ohne besondere Genehmigung auf diesen Frequenzbändern Kommunikation betreiben, solange

KAPITEL 5. WLAN - EIN KURZER TECHNOLOGISCHER ÜBERBLICK

gewisse Beschränkungen beispielsweise hinsichtlich der Sendeleistung beachtet werden.

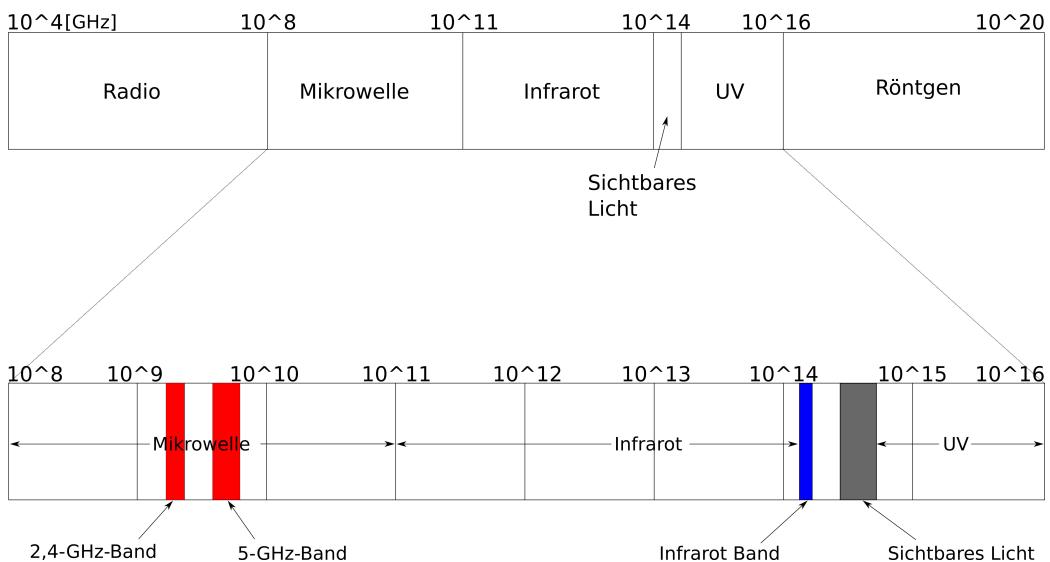


Abbildung 5.1: Frequenzbänder für WLANs im elektromagnetischen Spektrum

Um in der Lage zu sein, Kommunikation zwischen Computern über ein WLAN zu realisieren, kann nicht davon ausgegangen werden, dass permanent eine Verbindung zwischen ihnen besteht. Nutzt man beispielsweise eine Ethernet-Verbindung zwischen ihnen, so ist aufgrund der physikalischen Verbindung der Rechner eine Übertragung grundsätzlich möglich. Ein grundlegendes sicherheitstechnisches Problem tritt hier ebenfalls nicht auf, denn wenn für eine Person der physikalische Zugang zu einem Rechner oder zu einem Netzwerkkabel, was an einen Rechner angeschlossen ist, möglich ist, so ist der Zugriff im Grunde als autorisiert zu betrachten. Weitere Abschottungsmaßnahmen hinsichtlich Firewalls oder geeigneter Passwortwahl sind hier zunächst außer Acht gelassen. Bei WLANs ist diese Autorisierung nun wesentlich schwerer abzusichern, da ein einfaches Schloss an der Zugangstür nicht ausreicht: Funkwellen durchdringen auch gemauerte Wände, was zur Folge hat, dass prinzipiell jeder mit einer WLAN-Karte bestückte Rechner Zugang zum Netzwerk erlangen kann.

Sicherungsmaßnahmen wie **WEP** oder **WPA** können zwischen zwei WLAN-Funkstationen vereinbart werden, was durch die Managementroutinen des 802.11-Standards ermöglicht wird.

5.2 Einfache Managementroutinen

Innerhalb des IEEE802.11-Standards gibt es sogenannte Managementframes, die als Grundgerüst immer die gleiche Struktur haben, lediglich der Daten- teil des Frames, der Framebody, unterscheidet sich bei den verschiedenen Managementframes. Wie aus Tabelle 5.2 ersichtlich, gehören insgesamt elf verschiedene Managementframes zum IEEE802.11-Standard.

Die in Tabelle 5.2 fett gedruckten Managementframes sind Bestandteil der hier etwas näher erläuterten Managementroutinen. Ihnen wird größere Aufmerksamkeit zuteil, da sie wesentlicher Bestandteil dieser Diplomarbeit sind. Das generelle Format eines Managementframes ist aus Tabelle 5.2 ersichtlich.

Das physikalische Kabel und das Anklemmen eines solchen Kabels wie bei Ethernet muss also auf irgendeine Art und Weise über das drahtlose Medium realisiert werden. Zu diesem Zweck gibt es die Vorgänge der Authentifikation, Assoziation, Deauthentifikation und Disassoziation sowie das Acknowledgementframe.

Fachbegriffe

Zum Verständnis der folgenden Erläuterungen sollen an dieser Stelle einige Fachbegriffe näher erklärt werden.

- Station

Eine Station ist eine Komponente mit IEEE802.11-Fähigkeiten, also prinzipiell jeder Computer, der über eine WLAN-Schnittstelle verfügt.

- Station Service

Die Dienste, die von allen verfügbaren IEEE802.11-Stationen, Accesspoints eingeschlossen, angeboten werden, heißen „**Station Services**“. Im Einzelnen gehören von den hier genannten die Authentifikation und die Deauthentifikation zu den „Station Services“.[17]

- IBSS

IBSS ist die Kurzform für „**Independant Basic Service Set**“ und bildet die einfachste Variante eines IEEE802.11-Netzwerkes. Schon zwei Stationen können ein solches Netzwerk aufbauen, in dem sie nur so lange Daten austauschen können, wie sie sich in der gegenseitigen Empfangsreichweite befinden. Da dieses 802.11-Netzwerk oft ohne grundlegende vorherige Planung aufgebaut wird, wird es oft auch als „AdHoc Netzwerk“ bezeichnet.[17]

- Distributionssystem

Für größere Funknetzwerke reicht die direkte Verbindung von einer Komponente zur anderen nicht aus. Anstelle nebeneinander unabhängig zu existieren, können mehrere **Basic Service Sets** miteinander verbunden werden. Die architektonische Komponente, die eine Verbindung mehrerer Basic Service Sets ermöglicht, ist das *Distributionssystem*. Ein Accesspoint ist eine Station, die Zugang zu einem solchen Distributionssystem anbietet, indem sie zusätzlich zu ihrer Funktionalität als Station Distributionsdienste anbietet.[17]

- ESS

ESS steht für „**Extended Service Set**“ und stellt das Netzwerk dar, das durch mehrere Basic Service Sets erstellt wird. Höheren Bekanntheitsgrad hat diese Form eines IEEE802.11-Netzwerke unter dem Namen „Infrastruktornetzwerk“ erlangt.

5.2.1 Authentifikation

In verkabelten LANs kann physikalische Sicherheit zum Unterbinden von unautorisiertem Zugang genutzt werden. Dies ist in drahtlosen LANs unmöglich, da sie ein Medium ohne definierte Grenzen nutzen.

IEEE802.11 bietet die Fähigkeit, LAN-Zugang über den Authentifikationsdienst zu kontrollieren. Der Dienst wird von allen Stationen genutzt, um ihre Identität auf den Stationen, mit denen sie kommunizieren wollen, bekannt zu machen. Dies gilt sowohl für ESS- als auch für IBSS-Netzwerke. Wenn kein gegenseitig vereinbarter Grad der Authentifizierung aufgebaut werden kann, kann keine Assoziation aufgebaut werden. Die Authentifizierung ist ein Station Service.

Eine Station darf zu jedem gegebenen Zeitpunkt mit vielen anderen Stationen authentifiziert sein.[17]

Zur Authentifikation gehört die *Authentication* aus Tabelle 5.2.

5.2.2 Assoziation

Um eine Nachricht innerhalb eines Distributionssystems zu versenden, muss im Distributionsdienst bekannt sein, welcher Accesspoint für die gefragte Station anzusprechen ist. Diese Information wird vom Distributionssystem durch das Konzept der Assoziation geliefert. Die Assoziation ist notwendig, aber nicht hinreichend, um BSS-Beweglichkeit zu bieten. Eine Assoziation ist hinreichend, um ohne Beweglichkeit arbeiten zu können.

Bevor eine Station eine Datennachricht über einen Accesspoint senden darf, muss sie mit einem Accesspoint assoziiert sein. Der Vorgang des Assoziierens ruft den Assoziationsservice auf, welcher das Zuordnen von Stations in Accesspoints zum Distributionsystem bietet. Das Distributionsystem benutzt diese Information, um seinen Nachrichtenverteilungsdienst zu verrichten. Zu keinem Zeitpunkt darf eine Station zu mehr als einem Accesspoint assoziiert sein. Dies stellt sicher, dass das Distributionsystem eine eindeutige Antwort auf die Frage „Welcher Accesspoint betreut Station X?“ liefern kann. Wenn eine Assoziation vervollständigt ist, darf eine Station ein Distributionsystem über einen Accesspoint voll nutzen. Eine Assoziation wird immer von einer Station initiiert, nicht vom Accesspoint. Ein Accesspoint darf zu einem Zeitpunkt mit vielen Stationen assoziiert sein.[17]

Dieser Routine sind das *Association request* und die *Association response* aus Tabelle 5.2 zuzuordnen.

5.2.3 Deauthentifikation

Der Deauthentifikationsdienst wird einbezogen, wenn eine bestehende Authentifikation beendet werden muss. Die Deauthentifikation ist ein System Service.

Da Authentifikation eine Bedingung für die Assoziation ist, muss der Vorgang der Deauthentifikation eine Station in einem ESS dazu bringen, dissoziiert zu werden. Der Deauthentifikationsdienst darf von jeder authentifizierten Einheit durchgeführt werden. Eine Deauthentifikation ist keine Anfrage, sie ist eine Mitteilung. Sie darf von keiner Seite abgelehnt werden. Wenn ein Accesspoint eine Deauthentifikationsnachricht an eine assoziierte Station schickt, muss die Assoziation auch beendet werden.[17]

Der Deauthentifikation ist die *Deauthentication* aus Tabelle 5.2 eindeutig zuzuordnen.

5.2.4 Disassoziation

Der Disassoziationsdienst wird immer dann einbezogen, wenn eine bestehende Assoziation beendet wird. Eine Disassoziation ist ein **Distribution System Service**.

In einem ESS erteilt die Disassoziation dem **Distribution System** den Auftrag, bestehende Assoziationsinformationen zu verwerfen. Jeder Versuch, Nachrichten über das Distribution System an eine dissoziierte Station zu

senden, ist erfolglos.

Der Dissoziationsdienst darf von jeder Instanz aufgerufen werden. Er ist keine Anfrage, sondern vielmehr eine Benachrichtigung, die von keinem Teilnehmer einer Assoziation abgelehnt werden kann.

Accesspoints müssen möglicherweise Stationen dissoziieren, um einen Accesspoint aus dem Netzwerk herauszunehmen.

Stationen sollten immer versuchen, sich zu dissoziieren, wenn sie ein Netzwerk verlassen. Dennoch ist das **MAC** Protokoll nicht darauf angewiesen, dass Stationen sich dissoziieren, da es bereits so angelegt ist, um den Verlust einer Station zu verkraften.[17]

Die IEEE802.11-Implementation der Dissoziation ist die *Disassociation* aus Tabelle 5.2.

5.3 Beacons

Wörtlich übersetzt liefert der englische Begriff die deutsche Bedeutung „Lichtsignal“, „Leuchtstrahlsender“ oder „Signalfeuer“. Beacons liefern in WLANs prinzipiell genau diese Funktion. Dauerhaft von Accesspoints ungefragt ausgesendet liefern Beacons jederzeit Informationen über den Accesspoint und das von ihm angebotene Netz.

So finden sich in einem solchen Frame Informationen wie die MAC-Adresse des Accesspoints, die BSSID seiner zugehörigen Koordinationseinheit und weitere, aus Tabelle 5.3 entnehmbare Informationen.

Auf die Begriffe *frequency-hopping*, *direct-sequence* und *PCF (point coordination function)* wird in diesem Dokument nicht näher eingegangen.

Defaultmäßig sind bei der auf den Accesspoints installierten Version von OpenWRT Intervalle von etwa einer Zehntelsekunde für den Versand von Beacons eingestellt.

Einige Accesspoints ermöglichen das „Verstecken der SSID“, was einigen Endnutzern trügerische Sicherheit vorgaukelt, da nun die Beacons entweder mit einer leeren SSID verschickt wurden oder gar ganz abgeschaltet werden konnten. Da bei einer Assoziationsanfrage der Accesspoint aber standardkonform mit seiner eingestellten SSID antwortet, ist diese Hürde für einen Angreifer sehr leicht zu nehmen. In dieser Diplomarbeit wird grundsätzlich davon ausgegangen, dass Accesspoints immerzu Beacons aussenden.

5.4 Der IEEE802.11-Header

WLAN-Frames unterscheiden sich signifikant von Ethernet-Frames. Während bei der Kabelverbindung im Ethernet lediglich die Ziel- und die Sende-MAC Adresse eindeutig sein müssen, gibt es im WLAN ungleich mehr variable Komponenten, um ein **Paket** eines beliebigen Protokolls durch ein Netzwerk zu schicken.

Als Header bezeichnet man den Kopf eines Netzwerkframes, welcher ein fester Bestandteil jedes Frames in einem solchen Netzwerk ist und jederzeit den prinzipiell gleichen Aufbau hat. Hier sind grundlegende Informationen auf OSI-Ebene 2, wie die oben genannten notwendigen Adressen oder der nach dem Header folgende Protokolltyp, abgelegt. Typischerweise sieht ein Ethernetheader aus wie in Tabelle 5.4 schematisch dargestellt.

Ein WLAN-Header trägt deshalb weitaus mehr Daten, weil weitaus mehr Informationen zu übermitteln sind. Es muss beispielsweise angegeben werden, in welcher BSSID sich das Frame bewegen soll. Dazu gibt es eine eindeutige Quelladresse und eine Zieladresse. Falls ein Wireless Distribution System(**WDS**) verwendet wird, erweitert sich diese Adressierung noch um eine weitere Adresse. Zusätzlich ist noch die Information von Belang, ob ein Frame im aktuellen Distributionsystem verbleiben soll oder es verlassen muss, von einer Station kommt und direkt für eine andere Station adressiert ist oder eben WDS benutzt.

Es gehören also wesentlich vielfältigere Informationen in einen solchen WLAN-Header, dessen grundlegender Aufbau den ersten sieben Einheiten aus Tabelle 5.2 entspricht.

Im Rahmen dieser Diplomarbeit sind nicht alle Informationen, die im WLAN-Header einstellbar sind, von Interesse. So sollen hier nur die Adressfelder und die mit Ihnen in Verbindung stehenden Bits „To DS“ und „From DS“ innerhalb des „Frame Control“-Teils erläutert werden.

Die möglichen Kombinationen der „To DS“- und „From DS“-Bits ergeben die Belegung der in Tabelle 5.2 erwähnten Adressen. Die beiden Bits selbst finden sich im „Frame Control“-Teil des WLAN-Headers wieder, welcher die ersten zwei Bytes des WLAN-Headers belegt.

Im Detail stellen sich die beiden „Frame Control“-Bytes wie in Tabelle 5.5 dar. Mögliche Kombinationen der Bits „To DS“ und „From DS“ illustriert Tabelle 5.6.

Entsprechend der gesetzten Bits aus Tabelle 5.6 ergibt sich die Belegung der Adressen im WLAN-Header. Da im Rahmen dieser Diplomarbeit mehrere verschiedene Versionen dieser Kombinationen auftreten, soll hier kurz auf

KAPITEL 5. WLAN - EIN KURZER TECHNOLOGISCHER ÜBERBLICK

diese eingegangen werden. Tabelle 5.7 listet die möglichen Bedeutungen auf.

Für detailliertere Informationen bezüglich der Inhalte einzelner Teile des Headers sei auf den IEEE802.11-Standard [17] verwiesen.

Typ Wert b3 b2	Typ Beschreibung	Subtyp Wert b7 b6 b5 b4	Subtyp Beschreibung
00	Management	0000	Association request
00	Management	0001	Association response
00	Management	0010	Reassociation request
00	Management	0011	Reassociation response
00	Management	0100	Probe request
00	Management	0101	Probe response
00	Management	0110-0111	Reserved
00	Management	1000	Beacon
00	Management	1001	ATIM
00	Management	1010	Disassociation
00	Management	1011	Authentication
00	Management	1100	Deauthentication
00	Management	1101-1111	Reserved
01	Control	0000-1001	Reserved
01	Control	1010	Power Save-Poll
01	Control	1011	Request-to-send
01	Control	1100	Clear-to-send
01	Control	1101	Acknowledgement(ACK)
01	Control	1110	Contention-Free(CF)
01	Control	1111	CF-End + CF-ACK
10	Data	0000	Data
10	Data	0001	Data + CF-ACK
10	Data	0010	Data + CF-Poll
10	Data	0011	Data + CF-ACK + CF-Poll
10	Data	0100	Null function
10	Data	0101	CF-ACK
10	Data	0110	CF-Poll
10	Data	0111	CF-ACK + CF-Poll
10	Data	1000-1111	Reserved
11	Reserved	0000-1111	Reserved

Tabelle 5.1: Zulässige Typ/Subtyp Kombinationen [17]

Bytes	2	2	6	6	6	2	6	0- 2312	4
Meaning	Frame Control	Duration	Address 1	Address 2	Address 3	Sequence Control	Address 4	Frame Body	FCS

Tabelle 5.2: Das grundlegende Format von Managementframes [17]

Reihenfolge	Information	Bemerkung
1	Zeitstempel	
2	Beacon-Intervall	
3	Fähigkeiten des APs	
4	SSID	
5	Unterstützte Datenraten	
6	FH Parameter Satz	Das FH-Parameter-Satz-Informationselement ist in Beaconframes zu finden, die von STAs erstellt wurden, die frequency-hopping nutzen.
7	DS Parameter Satz	Das DS-Parameter-Satz-Informationselement ist in Beaconframes zu finden, die von STAs erstellt wurden, die direct-sequence nutzen.
8	CF Parameter Satz	Das CF-Parameter-Satz-Informationselement ist in Beaconframes zu finden, die von APs erstellt wurden, die PCF unterstützen.
9	IBSS Parameter Satz	Das IBSS-Parameter-Satz-Informationselement ist in Beaconframes zu finden, die von STAs erstellt wurden, die sich in einem IBSS befinden.
10	TIM	Das TIM-Informationselement ist nur in Beacons zu finden, die von APs erstellt wurden.

Tabelle 5.3: Mögliche Inhalte des Frame-Bodys in einem Beacon-frame [17]

Bytes	6	6	2
Bedeutung	Ziel-Adresse	Quell-Adresse	Protokolltyp

Tabelle 5.4: Grundlegender Aufbau eines Ethernet-Headers [17]

Bits	2	2	4	1	1	1	1	1	1	1	1
Meaning	Protocol Version	Type	Sub-type	To DS	From DS	More Frag	Retry	Pwr Mgt	More Data	WEP	Order

Tabelle 5.5: Bitweiser Aufbau der „Frame Control“-Bytes im WLAN-Header [17]

ToDS-Wert	FromDS-Wert	Bedeutung
0	0	Daten Frame von einer STA zu einer anderen STA im selben IBSS; alle Management Frames oder Control Frames
1	0	Daten Frame, das für das DS bestimmt ist.
0	1	Daten Frame, das das DS verlässt.
1	1	WDS-Frame von einem AP zu einem anderen AP.

Tabelle 5.6: ToDS und FromDS Werte mit deren Bedeutung [17]

ToDS-Wert	FromDS-Wert	Address1	Address2	Address3	Address4
0	0	Ziel-addresse	Quell-addresse	BSSID	N/A
1	0	Ziel-addresse	BSSID	Quell-addresse	N/A
0	1	BSSID	Quell-addresse	Ziel-addresse	N/A
1	1	Empfänger-addresse	Sender-addresse	Ziel-addresse	Quell-addresse

Tabelle 5.7: Mögliche Adressfeldinhalte abhängig von den ToDS und FromDS Bits aus Tabelle 5.6 [17]

Kapitel 6

Userland-Software mit ipw2200 und MadWiFi

6.1 Unzulänglichkeit des Linuxtreibers ipw2200

Der Stand der Technik von Intel Wireless Karten unter Linux hinsichtlich Roaming stellt sich vergleichsweise dürftig dar. Schaut man sich mit iwlist(8) die Feldstärken verschiedener Accesspoints an, so wechseln diese erwartungsgemäß bei Bewegung des Clients, im Testfalle ein Notebook mit Intel® PRO/Wireless 2200BG-Wireless-Chipsatz. Schwächt nun ein Signal ab, beispielsweise indem zuvor montierte Antennen abgeschraubt werden, so wird dies direkt in den ausgegebenen Werten von iwlist(8) angezeigt. Das Signal des entsprechenden Accesspoints wird schwächer. Ist nun ein stärkerer Accesspoint in Reichweite, so sollte im idealen Roamingfall auf diesen Accesspoint gewechselt werden. Wird ein solcher Wechsel vollzogen und das Signal des zuerst assoziierten Accesspoints anschließend wieder erhöht, beispielsweise durch Neumontage der Antennen oder lokale Annäherung des Accesspoints an den Client, so wird das Signal nicht besser, sondern bleibt auf seinem alten Wert stehen. Dies ändert sich ausschließlich in dem Fall, dass der neu assoziierte Accesspoint durch Deaktivierung der WLAN-Karte, komplettes Abschalten des Accesspoints oder verlassen des Empfangsbereiches dieses Accesspoints nicht mehr genutzt werden kann. Nur in diesem Fall wird das Ergebnis von iwlist(8) aktualisiert und die reale Feldstärke des Accesspoints angezeigt.

Zugrunde liegt diesem Vorgang innerhalb von iwlist(8) ein ioctl(2)-Befehl. Um einen Scan anzustoßen wird innerhalb der Software der Befehl

```
iw_set_ext(socket_fd, device, SIOCSIWSCAN, &wrq);
```

abgesetzt. Dieser Befehl beinhaltet den ioctl(2)-Befehl `SIOCSIWSCAN`, welcher laut `/usr/include/wireless.h` definiert ist als

```
#define SIOCSIWSCANOx8B18 /* trigger scanning (list cells) */.
```

Um die Ergebnisse dieses Scans anschließend abzurufen, wird der iwlib-Befehl

```
iw_get_ext(socket_fd, device, SIOCGIWSCAN, &wrq);
```

benutzt, welcher den ioctl(2)-Befehl `SIOCGIWSCAN` beinhaltet.

```
#define SIOCGIWSCAN0x8B19 /* get scanning results */
```

Selbst eine Automatisierung dieser Suche nach neuen Accesspoints mit expliziter Anwendung dieser Befehle verbessert die Ergebnisse nicht. Laut der manpage von `iwlist(8)` ist die Information, die aus einem Scan gezogen werden kann, stark treiberabhängig:

„The type of information returned depends on what the card supports. (...), the way scanning is done (the scope of the scan) will be impacted by the current setting of the driver.“[2]



Abbildung 6.1: Auch dieser Versuchsaufbau änderte nichts an den angezeigten schwachen Signalen des Intel Treibers

Offensichtlich bestand für die Entwickler des Linuxtreibers keine Notwendigkeit, Scan-Ergebnisse eines anderen Accesspoints neu auszuwerten, solange ein Accesspoint der gleichen SSID assoziiert ist, gleich mit welcher Signalstärke. Selbst ein Versuchsaufbau wie in Abbildung 6.1 erhöht die Signalstärke des Accesspoints nicht. Erst ein Lösen der Verbindung mit dem aktuell assoziierten Accesspoint aktualisiert die gescannte Signalstärke des in Abbildung 6.1 dargestellten Accesspoints.

Diese Ergebnisse zeigen einen unzulänglichen Linuxtreiber hinsichtlich Roaming.

6.2 Roaming mit MadWiFi

Wesentlich besser verhalten sich von Anfang an Atheros Karten. Hier werden durchgehend die scan-Ergebnisse aktualisiert, auch bei oben genanntem

Aufbau, der der Intel-Karte Probleme bereitete. Zum Umassoziiieren mit Karten dieses Herstellers muss die WLAN-Karte einmal heruntergefahren und dann wieder aktiviert werden. Erste Messungen der Zeit für ein Herunterfahren des Interfaces, Neuassoziiieren mit einem Accesspoint und Aktivieren der Wireless-Karte ergaben mit voller Datenlast $\sim 94,6$ ms und ohne Datenlast $\sim 82,3$ ms. Diese Zeitwerte wurden bei einer Interfacebehandlung mit linuxüblichen Tools erzielt. Das Herunterfahren und wieder Hochfahren der Netzwerkkarte wurde also mit dem Befehl

```
ifconfig ath0 down bzw.
```

```
ifconfig ath0 up
```

durchgeführt. Hinzu kommt die Zeit, die das kompilierte Programm benötigt, einen system(3)-Befehl abzusetzen.

6.2.1 Umassoziation mit ioctl(2)-Befehlen

Grundlegende Beschleunigung dieses Umassozierens ist ohne den Befehl system(3) erreicht worden. Nutzt man ioctl(2)-Befehle sachgemäß, so stellen sich enorme Verbesserungen hinsichtlich des Zeitverbrauchs eines solchen Umassozierens ein. In Listing 6.1 sind die entsprechenden Befehle zu finden.

Listing 6.1: Umassozieren mit ioctl(3)-flags

```

1  struct ifreq my_ifr;
2  memset(&my_ifr,0,sizeof(my_ifr));
3  strcpy(my_ifr.ifr_ifrn.ifrn_name,device);
4
5  if (ioctl(socket_fd, SIOCGIFFLAGS, (caddr_t)&my_ifr) < 0) {
6      perror("ioctl (SIOCGIFFLAGS)");
7      exit(1);
8  }
9  //get start-time
10 gettimeofday(&start,&here);
11 //set flag for interface DOWN
12 my_ifr.ifr_ifru.ifru_flags = 0x0302;
13 //put interface down
14 if (ioctl(socket_fd, SIOCSIFFLAGS, (caddr_t)&my_ifr) < 0) {
15     perror("ioctl (SIOCSIFFLAGS)");
16     exit(1);
17 }
18 //Actually set the device to the new AP
19 if( iw_set_ext(socket_fd, device, SIOCSIWAP, &wrq) < 0) {
20     perror("Could not set device");
21 } else {
22     //set flag for interface UP
23     my_ifr.ifr_ifru.ifru_flags = 0x0301;
24     //put interface back UP
25     if (ioctl(socket_fd, SIOCSIFFLAGS, (caddr_t)&my_ifr) < 0) {
26         perror("ioctl (SIOCSIFFLAGS)");
27         exit(1);
28     }
29     //get finish-time
30     gettimeofday(&stop,&here);
31     result = stop.tv_usec - start.tv_usec;
32     printf("Elapsed: %i usec\n",result);
33     printf( "New chosen AP:\n" \
34             "MAC:    %s\n" \
35             "ESSID:   %s\n" \
36             "Quality: %i\n\n", \

```

```

38     APs_with_correct_ssid[chosen].MAC, \
39     APs_with_correct_ssid[chosen].ssid,\ \
40     APs_with_correct_ssid[chosen].qual);
}

```

So wird zunächst eine Variable vom Typ `struct ifreq` definiert. Ihr Aufbau ist in Listing 6.2 zu ersehen. Dieser Datentyp ist für Zugriffe auf ein Netzwerkinterface konzipiert, was auch leicht an den Namen der Bestandteile dieses `structs` zu erkennen ist. So werden in diesem `struct` unter anderem Spezifikationen wie die HardwareAdresse, Netzwerkmaske oder IP-Adresse vereinbart.

Listing 6.2: Der Datentyp struct ifreq

```

1  struct ifreq
2  {
3      union
4      {
5          char ifrn_name[IFNAMSIZ]; /* if name, e.g. "ath0" */
6      } ifr_ifrn;
7
8      union {
9          struct sockaddr ifru_addr;
10         struct sockaddr ifru_dstaddr;
11         struct sockaddr ifru_broadaddr;
12         struct sockaddr ifru_netmask;
13         struct sockaddr ifru_hwaddr;
14         short ifru_flags;
15         int ifru_ivalue;
16         int ifru_mtu;
17         struct ifmap ifru_map;
18         char ifru_slave[IFNAMSIZ]; /* Just fits the size */
19         char ifru_newname[IFNAMSIZ];
20         char * ifru_data;
21         struct if_settings ifru_settings;
22     } ifr_ifru;
23 };

```

Um nun Befehle an ein Netzwerkinterface mittels `ioctl(2)`-Befehlen abzusetzen, muss ein solches `struct ifreq` mit einem gewissen Grundstock an Informationen gefüllt werden. In diesem speziellen Beispiel beginnt das mit dem Kopieren des Interfacenamens per `strcpy(3)` in Zeile 3. Weiterhin ist es notwendig, die aktuell gesetzten Flags der Netzwerkkarte in das genannte `struct` zu schreiben, was der Aufruf in Zeile 5 durchführt. Anschließend beginnt die Zeitmessung, die Karte wird heruntergefahren (das Besondere an dieser Stelle ist nun, dass dies mit einem `ioctl(2)`-Befehl geschieht), und in Zeile 16 wird ein zuvor ausgesuchter Accesspoint neu assoziiert. Dieser Befehl aus der `iwlib`-Bibliothek setzt einen `ioctl(2)` ab, der wie in Listing 6.1 angegeben deklariert werden muss. In diesem speziellen Fall wäre das also der Aufruf von `SIOCSIWAP`, was laut `/usr/include/linux/wireless` definiert ist als

```
#define SIOCSIWAP 0x8B14 /* set access point MAC addresses */
```

Somit kann durch diese Zeile ein Wechsel des Accesspoints vollzogen werden.

Abschließend wird das Interface wieder hochgefahren und die Stoppzeit genommen, um einen Indikator für die Geschwindigkeit dieses Roamings zu erhalten.

Wird nun die Zeit an den entsprechenden Stellen in Listing 6.1 gemessen, so ergibt sich eine Zeit von durchschnittlich etwa 20 ms.

Wie sich nach einiger Zeit herausstellte, sind diese Zahlenwerte jedoch für die Anwendung von keinerlei Bedeutung. Das erscheint bei näherem Hinsehen auch logisch, da die Zeit, die das Programm für dieses Umsetzen benötigt noch nicht die Übergabe der Frames an einen neuen Accesspoint, dessen Handhabe dieser Frames inklusive Verarbeitung und Pflege einer neuen Station und die Laufzeit der Frames beinhaltet. Um hier wirklich aussagekräftige Zahlen zu erlangen, ist ein Test auf Rundlaufzeiten von IP-Paketen aussagekräftiger, worauf in Kapitel 10 näher eingegangen wird.

Das eigentliche Problem bei all diesen Versuchen bestand darin, dass jedes Roaming auf Daten beruhte, die aus Scans gewonnen wurden. Die Software suchte also aktiv nach neuen Accesspoints. Diese Scans funktionierten zwar wie bereits erwähnt bei Atheroskarten wesentlich besser als bei Intelkarten. Wurde die WLAN-Karte jedoch gleichzeitig mit großer Datenlast beaufschlagt, so konnten Scans kaum noch abgearbeitet werden und es dauerte eine beträchtliche Zeit (bis zu 15 s), bis ein signalstärkerer Accesspoint gefunden wurde. Dies stellt ein großes Manko dar, da in Randbereichen des Empfangsbereiches immer noch hohe Datenlast auf dem drahtlosen Medium liegen kann und das auch bis kurz vor dem Zeitpunkt des Verbindungsabisses. Wurde in einem solchen Fall nicht rechtzeitig ein neuer Accesspoint gefunden, weil wegen hoher Datenlast kein Scan abgearbeitet werden konnte, so stellt sich hier schnell ein Problem ein: es wird hohe Totzeiten geben und Daten erreichen möglicherweise ihr Ziel überhaupt nicht mehr. Beweise hierfür liefert Kapitel 10.

6.3 Die Netzwerkkarte im Monitormodus

6.3.1 Vorteile des Monitormodus

Alle bisher abgehandelten Umassozierungsmethoden setzten eine WLAN-Karte voraus, die im Managed-Modus arbeitete. Der große Nachteil dieses Kartenmodus ist zusätzlich zu dürftig implementierten Scaneigenschaften die Tatsache, dass keine Daten sichtbar sind, die auf der OSI-Schicht 2 zu finden sind. Es können also keinerlei Beacons abgefangen werden. Die Möglichkeit, Beacons auszuwerten eröffnet jedoch die Möglichkeit, die Signalstärke von Accesspoints zu bekommen, ohne einen Scan zu starten. Dies ist von großem Vorteil, da bei einem passiven Evaluieren der ohnehin präsenten Beacons kei-

ne Bandbreite durch einen selbst initiierten, aktiven Scan verloren geht. Nutzt man den MadWiFi-Treiber und wertet die Beacons aus, welche standardmäßig einen **Prism-Header** angehängt bekommen, so findet sich an einer bestimmten Stelle in diesem Prism-Header die von der Karte empfangene Signalstärke. Diese kann anschließend ausgelesen werden und somit ist ohne jeden Aufwand und vollkommen passiv die aktuelle Feldstärke aller Accesspoints, die sich in Reichweite befinden, verfügbar.

Zusätzlich ergibt sich durch den Monitormodus die Möglichkeit, selbst Frames zu bauen und diese nahezu bitgleich in die Luft abzugeben. Das eröffnet die Möglichkeit, das Management von Accesspoints durch eigene Frames selbst zu handhaben. Automatisch findet in diesem Modus also keine Assoziation oder Authentifikation statt. Nutzbar ist dies vor Allem vor dem Hintergrund der passiv stets empfangenen Beacons, auf Grundlage derer die Suche nach neuen Accesspoints mit besserer Signalstärke durchgeführt werden kann.

Beacons werden bei einer Accesspoint-Standardinstallation von OpenWRT [13] im Interval von ~ 10 Beacons pro Sekunde gesendet. Der Empfang dieser Beacons ist auch bei voller Datenlast nicht merklich eingeschränkt. Selbst bei einem Testaufbau mit zehn übereinander gestapelten Accesspoints, die allesamt auf dem gleichen Kanal arbeiteten, wurden noch 90% der Beacons von einem direkt daneben stehenden Client empfangen.

6.3.2 Nachteile trotz Monitormodus

Eine grundlegende Annahme zu Beginn dieser Diplomarbeit bestand darin, dass eine Netzwerkkarte im Monitormodus ausschließlich die Signale sendet, die man ihr per Befehl an die Hand gibt. Erste Zweifel in diese Richtung kamen zum Vorschein, als im Rahmen des Roamings Authentifikationen und Assoziationen versendet wurden. In einer frühen Version der verwendeten Userlandsoftware wurden schlicht drei Authentifikationsanfragen und anschließend drei Assoziationsanfragen herausgeschickt, um sicherzustellen, dass auch eine dieser Anfragen am Ziel ankommt und somit der neu gewählte Accesspoint mit dem Client zusammenarbeiten will und kann. Da dies auch ohne weiteres funktionierte, bestand hier zunächst keine Begründung zur Änderung der Handhabe.

Um jedoch eine sicher funktionierende Roamingstruktur aufzubauen, die nicht auf Annahmen basiert („bei drei requests sollte auch bei hoher Datenlast ein Frame durchkommen...“), wurde der Abgleich zwischen Accesspoint und Client etwas näher untersucht. Dabei stellte sich heraus, dass jede Station im Rahmen der IEEE802.11-Richtlinien jedes empfangene Frame mit einem

Acknowledgementframe (Tabelle 6.1) beantwortet. Dass dies geschieht, ist auch sehr wichtig, da diese Acknowledges innerhalb von wenigen μ s gesendet werden müssen. Hier hört die Mächtigkeit des Monitormodus auf, denn Zugriff auf diese Acknowledges ist vom Userland aus unmöglich. Wollte man sich selbst um diese Acknowledges vom Userland aus kümmern, so würde keine sendende Station je diese abgesetzten Acknowledges anerkennen, da sie zu spät kämen. Die geforderten Zeiten von wenigen μ s sind mit einer Userlandsoftware nicht zu realisieren, da der Weg von dort bis herunter in den Kernel bereits um ein Vielfaches zu lang dauern würde. Diese Aufgabe übernimmt daher die bei MadWiFi verwendete und von Atheros selbst gepflegte Hardware Abstraction Layer (HAL). Sie arbeitet noch unterhalb des Treibers, quasi direkt an der Schnittstelle zur Hardware. Jedes Frame, das die WLAN-Karte passiert, kommtt auch an dieser HAL vorbei und so ist dies der optimale Ort, um Frames zu acknowledge.

Bytes	2	2	6	4
Meaning	Frame Control	Duration	Destination Address	Checksum

Tabelle 6.1: Das Format für Acknowledgements [17]

Dies ist notwendig, um Programmen, die Netzwerkverkehr nutzen, eine vergleichbare Datenflusssicherheit einer Kupferleitung zu bieten. Wird ein Frame durch Störungen im drahtlosen Medium nicht empfangen, so wird kein Acknowledgementframe versandt und der Sender muss das betreffende Frame erneut abschicken, bis das Frame quittiert wurde. Bei einem Lauschen auf der „Luftleitung“ mittels **Wireshark** auf der Maschine, die auch die Frames innerhalb des Userlandprogramms behandelt und verschickt, konnte ein solches Frame jedoch für empfangene Daten nicht ausgemacht werden. Offensichtlich wurde es nicht verschickt, was auch der Software entsprechend erschien, da innerhalb selbiger keinerlei Implementierung von Acknowledgementframes vorhanden war. Wie war dann aber ein Datentransfer zwischen dem Client mit der Userlandsoftware und einem anderen Rechner möglich? Eigentlich müsste nun kein Transfer stattfinden und nur dauerhaft das gleiche Frame in einer Endlosschleife gesendet werden, ohne jemals anzukommen, da ohne Acknowledgementframe der sendenden Station nicht bekannt ist, ob ein Frame am Accesspoint angekommen ist.
Erst ein Mitschneiden der Daten auf der drahtlosen Schnittstelle mit einem dritten, unbeteiligten Rechner belegte den Versand dieser Acknowledgementframes ohne Implementierung in der Software selbst. Die Anzeige in Wire-

shark auf dem Clientrechner war also unvollständig und, was ein noch wichtigerer Punkt dieser Beobachtung ist, ein anderes Modul zwischen Userlandsoftware und drahtloser Schnittstelle musste die Acknowledgementframes generieren.

Bei weiterer Analyse von versandten Daten mit Hilfe von Wireshark wurde ebenfalls deutlich, dass die *Sequence Number* und die *Fragment Number* auch nachträglich geändert werden. Das Userlandprogramm kann diese beiden Bytes setzen, sie werden jedoch anschließend von der HAL wieder abgeändert.

Diese Beobachtungen widerlegten nun die Annahme, mit dem Monitormodus volle Kontrolle über die WLAN-Karte zu haben und in einem quasi-Treibermodus zu arbeiten. Auch in diesem Modus gibt es noch Instanzen, die generierte Frames abändern und beeinflussen.

Dennoch fiel die Entscheidung für den Monitormodus recht leicht, da hier passiv sämtliche Accesspoints und deren Signalstärke verfügbar waren und das Management des assoziierten Accesspoints unabhängig vom Treiber geregelt werden konnte.

6.3.3 Aktivierung des Monitormodus

Die Atheroskarte wird mit folgenden Schritten in den Monitormodus gebracht:

```
wlanconfig ath0 destroy
```

Hier wird zunächst die Instanz eines evtl. bereits erstellten Pseudo-Devices gelöscht.

```
wlanconfig ath0 create wlandev wifi0 wlanmode monitor
```

Anschließend wird ein Pseudo-Device für die WLAN-Karte „*wifi0*“ erstellt, welches sich „*ath0*“ nennt und im „Monitormodus“ läuft.

```
ifconfig ath0 up
```

```
iwconfig ath0 channel 6
```

Die soeben erstellte Karte „*ath0*“ muss noch auf einen Kanal assoziiert werden. Hier sollte der Kanal gewählt werden, den der zu assoziierende Accesspoint ebenfalls nutzt, um überhaupt eine Kommunikation zu ermöglichen. Leider ist in diesem Zustand nur ein asymmetrischer Upstream und Downstream möglich. Der Downstream des Rechners mit einer so eingebundenen Karte im Monitormodus beträgt etwa 4 Mbit/s, der Upstream nur etwa 800 Kbit/s.

Um im Monitormodus gleiche 802.11b-Datenraten für Upstream und Downstream zu bekommen, ist folgender Schritt notwendig:

```
iwconfig ath0 rate 11M
```

Nun bleiben die Datenraten für Upstream und Downstream auf 802.11b in beiden Richtungen konstant bei etwa 4 Mbit/s. Dies ist jedoch bei weitem kein 802.11g, was theoretisch 54 Mbit/s und praktisch etwa 25 Mbit/s zur Verfügung stellen sollte. Welche Modi die Karte anbietet, kann mit einem

```
iwlist ath0 rate
```

erfragt werden. Im vorliegenden Fall wurde stets mit einer „rate“ von 54 Mbit/s gearbeitet, was effektiv einer Rate von etwa 25 Mbit/s entspricht.

Kapitel 7

Einrichten eines Tunnels zwischen Netzwerkkarte und WLAN-Karte

7.1 Injizieren von Datenframes

Um Frames von einem Ethernet-Netzwerk in ein WLAN-Netzwerk und umgekehrt verschicken zu können, ist es notwendig, die Frames in ihrem Aufbau abzuändern.

Um nicht den Stationsmodus des MadWiFi-Treibers zu nutzen, der wie in Kapitel 6 bereits erwähnt ein Roaming nicht in zufriedenstellendem Maße bietet, muss die Karte in den in Kapitel 6 genannten Monitormodus gesetzt werden.

Um Daten im Monitormodus zu versenden, muss der Treiber das Injizieren von Frames in diesem Modus erlauben. Im Auslieferungsfall eines Treibers ist dies in der Regel nicht möglich. Injizieren von Frames öffnet Tür und Tor für Angriffe auf Netzwerke.

Das wohl bekannteste Beispiel hierfür ist der „Blaster“- oder „Lovesan“-Virus, der am 15. August 2003 für einen Angriff auf einen Microsoftserver genutzt wurde und auch ein System, das diesen Virus installiert hatte, bei vorhandener Internetverbindung binnen Sekunden zum Herunterfahren zwang. Zwar spielte die WLAN-Technologie hier keine Rolle, jedoch wurde mit der RAW-Sockets Technik gearbeitet, welche auch beim Frameinjizieren im Monitormodus Verwendung finden.

So ergibt sich mit RAW-Sockets die Möglichkeit, selbst Frames aufzubauen, ohne sich an Protokolle halten zu müssen. Mit dieser mächtigen Technik sind viele Angriffe implementiert worden. Beispielsweise kann man sich mit per RAW-Socket injizierten Datenframes als Accesspoint ausgeben und Client-

Stationen so täuschen.

Der Nutzen für diese Diplomarbeit an der Technik des Frameinjizierens ist die Möglichkeit, mit einem Programm im Userland nahezu Vollzugriff auf die WLAN-Karte zu haben. Somit ist die WLAN-Karte beispielsweise nicht mehr nur an einen Accesspoint gebunden, sondern kann parallel diverse Accesspoints beobachten.

Da in dieser Abhandlung abgesehen von ersten Tests mit einer Intel-Karte ausschließlich WLAN-Karten mit Atheros Chipsatz verwendet wurden, welche in Kombination mit dem MadWiFi-Treiber betrieben wurden, konnte der Treiber unverändert Anwendung finden. Der MadWiFi-Treiber erlaubt Frameinjektion im Monitormodus im Auslieferungszustand.

Durch Vorträge wie z.B. auf dem 13. Automation-Day 2007 in Minden wurde ebenfalls deutlich, dass die Implementation solcher Algorithmen im Userland auch den Schutz des geistigen Eigentums vereinfacht. Grundsätzlich unterläge eine Implementation im Kernel der GPL [21] und wäre somit auch konkurrierenden Unternehmen zugänglich. Da im Folgenden auf Abhängigkeiten zu GPL-Code verzichtet wurde, kann die entstandene Lösung auch gut geschützt werden.

7.2 Tunneln von Datenframes

In der vorliegenden Diplomarbeit wurde der Aufbau derart realisiert, dass innerhalb eines Programms eine virtuelle Netzwerkkarte erstellt wurde. Dies gewährleistete Flexibilität hinsichtlich einer vorhandenen oder nicht vorhandenen physikalischen Netzwerkkarte.

Alle Programme können fortan eine eventuell physikalisch nicht vorhandene Netzwerkkarte nutzen, als sei sie im System präsent. Das Userlandprogramm greift später all diese Frames ab und wandelt sie in WLAN-Frames um. Mit dieser Vorgehensweise ist sowohl die Möglichkeit gegeben, das System selbst als Endpunkt für Datenkommunikation zu nutzen (beispielsweise durch Anschließen einer Kamera an USB-Ports oder durch mikrofonisch aufgenommene Audiosignale), als auch weitere Rechner, die an einer physikalisch vorhandenen Netzwerkkarte angeschlossen sind, über das WLAN mit einer Netzwerkstruktur zu verbinden.

Unter Linux kann eine solche virtuelle Netzwerkkarte mit Hilfe des `/dev/net/tun`-Devices erstellt werden. Der entsprechende Code für diese Aktion ist aus Listing 7.1 ersichtlich.

KAPITEL 7. EINRICHTEN EINES TUNNELS ZWISCHEN NETZWERKKARTE UND WLAN-KARTE

Listing 7.1: Erstellen einer virtuellen Netzwerkkarte unter Linux

```
1 int tap_open(char *dev)
2 {
3     struct ifreq ifr;
4     int fd;
5
6     if( (fd = open("/dev/net/tun", O_RDWR)) < 0 )
7         return -1;
8
9     memset(&ifr, 0, sizeof(ifr));
10    ifr.ifr_flags = IFF_TAP | IFF_NO_PI;
11    if( *dev )
12        strncpy(ifr.ifr_name, dev, IFNAMSIZ);
13
14    if (ioctl(fd, TUNSETIFF, (void *) &ifr) < 0) {
15        close(fd);
16        return -1;
17    }
18
19    strcpy(dev, ifr.ifr_name);
20    return fd;
}
```

Der ioctl(2)-Befehl `TUNSETIFF` erstellt das Tunnel-Device und bindet es an den Filedescriptor `fd` (Zeile 14 von Listing 7.1).

Die virtuelle Netzwerkkarte kann ab Implementation dieser Funktion von dem Programm angesprochen werden, da der erstellte Filedescriptor zurückgeliefert wird. Um auf die WLAN-Karte selbst generierte Frames schreiben zu können, müssen nun die erwähnten RAW Sockets erstellt werden. `packet(7)` [1] beschreibt jene RAW Sockets mit den Worten

„Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer.“[1]

Erstellt wird ein solcher Socket schlussendlich mit dem C-Befehl

```
socket( PF_PACKET, SOCK_RAW, htons( ETH_P_ALL ) );
```

RAW Frames werden also benutzt, um Daten auf der OSI Ebene 2 zu verarbeiten. Diese Frames erlauben es dem Nutzer, Protokollmodule im Userland direkt oberhalb der physikalischen Ebene zu erstellen [1]. So wurde für die ausgehenden Daten, wie auch für die eingehenden Daten, je ein Packetfiledeskriptor erstellt. Dieser Schritt ermöglicht es, Frames über die im Monitormodus befindliche WLAN-Karte zu verschicken und zu empfangen.

Werden nun Frames auf dem eingehenden Filedeskriptor registriert, können diese innerhalb der Software analysiert, gegebenenfalls angepasst und anschließend auf die virtuelle Netzwerkkarte herausgeschickt werden. Natürlich ist dieser Schritt auch umgekehrt möglich und so können Frames, die auf dem virtuellen Interface registriert werden, auf die WLAN-Karte geschickt werden und einen Empfänger erreichen. Im Prinzip arbeitet dieses Programm also wie ein Accesspoint, allerdings mit der Möglichkeit, die versandten Daten beliebig zu verändern, zu blocken oder mehrfach zu versenden.

KAPITEL 7. EINRICHTEN EINES TUNNELS ZWISCHEN NETZWERKKARTE UND WLAN-KARTE

Schlussfolgernd ist also festzuhalten, dass jedes Frame, das durch einen Netzwerktunnel mit einer virtuellen Netzwerkkarte läuft, angesehen und auch verändert werden kann. Prinzipiell bietet ein Netzwerktunnel, der sich in der Route eines Frames befindet, volle Kontrolle über grundlegende Dinge wie beispielsweise Sendezzeit oder Dateninhalt dieses Frames.

Kapitel 8

Roaming - der prinzipielle Vorgang

8.1 Technische Erläuterungen

Bei der Bewegung einer Station in einem WLAN-Netz mit mehreren Accesspoints kommt es zwangsläufig zu Zeitpunkten, in denen ein zuvor assoziierter Accesspoint nicht mehr in Reichweite ist oder ein anderer Accesspoint wesentlich stärker in seiner Sendeleistung ist, als ein anderer. An einer Funkstrecke mit mehreren Accesspoints besteht also entweder eine Konkurrenz mehrerer Accesspoints um einen Client oder zuvor aufgebaute Assoziationen zu Accesspoints verlieren Ihre Gültigkeit, weil der Client sich nicht mehr in ihrem Empfangsbereich befindet. In beiden Fällen besteht die Notwendigkeit zum Wechsel der Assoziation zwischen den Accesspoints. Im Idealfall findet der Client im überlappenden Abdeckungsbereich mehrerer Accesspoints den stärksten und assoziiert sich mit diesem, um jeden weiteren Datenverkehr mit eben jenem Accesspoint fortan abzuwickeln.

Um Datenverlust im Moment des Roamings zu vermeiden, muss sichergestellt werden, dass im Zeitraum des Roamings keine Frames mit unklarer Quell-Ziel-Route verloren gehen. Dies geschieht durch Abmeldung beim alten Accesspoint und Anmeldung am neuen Accesspoint. Meldet sich der Client also zuvor beim alten Accesspoint ab, so wird dieser kein Frame mehr in das drahtlose Medium versenden, das als Zieladresse den soeben abgemeldeten Client beinhaltet.

Dieser Prämisse folgend, muss sich ein Client beim aktuell assoziierten Accesspoint abmelden, bevor er roamen kann.

Im angemeldeten Zustand, welcher Datenverkehr zwischen Accesspoint

KAPITEL 8. ROAMING - DER PRINZIPIELLE VORGANG

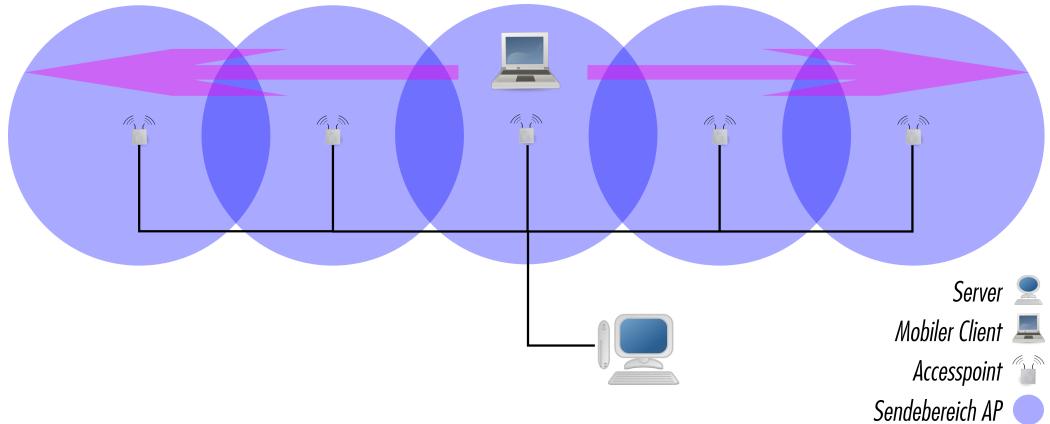


Abbildung 8.1: Bildliche Darstellung eines Roamings

und Client erlaubt, ist ein Client sowohl authentifiziert, als auch assoziiert. Um diese Verbindung zu lösen ist laut [17] lediglich eine Deauthentifizierung notwendig, da bei einer nicht bestehenden Authentifikation auch keine Assoziation bestehen kann: eine Authentifikation ist eine notwendige Bedingung für eine Assoziation. Wird also ein Roaming eingeleitet, so sind folgende Schritte vom Client durchzuführen:

- Deauthentifikation vom alten Accesspoint
- Authentifikation beim neuen Accesspoint
- Assoziation mit dem neuen Accesspoint

Schneidet man einen solchen Vorgang mit Wireshark mit, so nimmt dieser im Schnitt etwa 2,25 ms in Anspruch.

```

537 16.634080  Giga-Byt_25:f1:99      Cisco-Li_ed:e4:fb      IEEE 802 Deauthentication,SN=514,FN=0
538 16.634112  Giga-Byt_25:f1:99      Cisco-Li_b7:53:f6      IEEE 802 Acknowledgement
539 16.634161  Giga-Byt_25:f1:99      Cisco-Li_b7:53:f6      IEEE 802 Authentication,SN=515,FN=0
540 16.634200  Giga-Byt_25:f1:99      Cisco-Li_b7:53:f6      IEEE 802 Acknowledgement
541 16.634254  Giga-Byt_25:f1:99      Cisco-Li_b7:53:f6      IEEE 802 Association Request,SN=516,FN=1, SSID: "turbo"
542 16.634294  Giga-Byt_25:f1:99      Cisco-Li_b7:53:f6      IEEE 802 Acknowledgement
543 16.635040  Cisco-Li_b7:53:f6      Cisco-Li_b7:53:f6      IEEE 802 Authentication,SN=1262,FN=0
544 16.635344  Cisco-Li_b7:53:f6      Cisco-Li_b7:53:f6      IEEE 802 Acknowledgement
545 16.636094  Cisco-Li_b7:53:f6      Cisco-Li_b7:53:f6      IEEE 802 Association Response,SN=1263,FN=0
546 16.636406  Cisco-Li_b7:53:f6      Cisco-Li_b7:53:f6      IEEE 802 Acknowledgement

```

Abbildung 8.2: Ein Roaming als Wireshark-Mitschnitt. Die Differenz zwischen Deauthentication und Absenden des letzten Acknowledges beträgt $16,636406\text{ s} - 16,634080\text{ s} = 2,326\text{ ms}$

8.2 Totzeitfreies Roaming?

Da auf Accesspointseite keine modifizierte Software vorhanden ist, die einen solchen Moment der Nichtassoziation handhabt, entsteht hier faktisch eine Totzeit von eben jenen ~ 2 ms. Die Problematik stellt sich beispielsweise derart dar, dass alle möglichen Accesspoints zu jedem Zeitpunkt einen gewissen Anteil an Frames zwischenspeichern müssten, um diese im Falle einer Neuassoziation dem Client schicken zu können. Eine Synchronisation mit allen anderen Accesspoints wäre permanent notwendig, um Informationen darüber zu haben, welches Frame tatsächlich als letztes zum Client gesendet wurde. Würde dies nicht nur für einen Client, sondern für ungleich mehr Clients geschehen, ergäbe sich in dieser beispielhaften Lösungsmöglichkeit ein hoher Speicherbedarf, der von üblichen Accesspoints nicht gehandhabt werden könnte und zusätzlich entstünde hoher Managementtraffic auf der Ethernetseite unter den Accesspoints, um diese Informationen abzugleichen. Eine andere Möglichkeit wäre, dass der Client den Accesspoints bei einer Neuassoziation mitteilt, welches Frame als letztes empfangen wurde. Dies würde jedoch ein langsameres Roaming bedeuten, da nun in diesem Beispiel nach einer Neuassoziation noch zusätzliche Managementhandshakes durchgeführt werden müssten. Die effektive Roamingzeit würde also definitiv steigen und das Roaming Performanz verlieren.

Grundsätzlich ist eine Roamingzeit von ~ 2 ms als vergleichsweise schnell einzustufen, was die Wahrscheinlichkeit von verlorenen Frames in diesem kurzen Zeitraum sehr klein werden lässt. Des Weiteren wird eine „Totzeit“ noch von der Geschwindigkeit der Frameverarbeitung, den Acknowledges und zuletzt von der Lichtgeschwindigkeit beeinflusst.

Ein weiterer Punkt gegen eine Implementation auf Accesspointseite ist die Tatsache, dass hierdurch Allgemeingültigkeit der Anwendung verloren ginge. Client **und** Accesspoints müssten mit besonderer Software ausgestattet werden, was einen erhöhten Wartungsaufwand darstellt.

Nutzung von verbindungsorientierten Protokollen ist demnach im Falle von notwendigem Fehlerausschluss empfohlen, da diese über Fehlererkennungs- und Korrekturmaßnahmen verfügen, wobei in diesem Fall ohnehin die Nutzung von WLAN als Übertragungsmedium zu überdenken wäre.

Kapitel 9

ROAMEO

„roameo“ steht für „ROaming Algorithm Manufactured by Embigence Organization“ und beschreibt eine Software, die in der Lage ist, innerhalb von wenigen Millisekunden einen Accesspointwechsel, ein „Roaming“, durchzuführen.

Bei der Anwendung dieses Programms werden einige Parameter abgefragt, wobei die drei wichtigsten, zum Programmstart notwendigen die *-m*, *-n* und *-i* Optionen sind. *-m* erwartet als Parameter das *Gerät*, auf dem gearbeitet werden soll (bei Atheros Karten üblicherweise *athX*), als Parameter nach *-n* folgt die zu nutzende *SSID* und *-i* gibt die zu verwendende *IP-Adresse* an.

Ab Zeile 1042 des in Listing B.1 befindlichen Codes von Roameo beginnt die Konfiguration der Geräte, in diesem Fall mit dem Aufruf der Funktion `open_sockets`. Innerhalb dieser Funktion wird zunächst ein UDP-Socket als Interface zum Kernel für sämtliche `ioctl(2)`-Anfragen angelegt. Anschließend folgt das Anlegen eines RAW-Sockets für die WLAN-Karte, welches notwendig ist, um Layer2-Frames abgreifen zu können [1]. Der darauffolgende Befehl ruft die Funktion `openraw` für das WLAN-Gerät auf, was im Kern den Socket für dieses Gerät auf den Empfang von Rohdaten einstellt und an das Gerät bindet Listing 9.1.

Listing 9.1: Das Binden eines Rohdaten-Sockets an ein Gerät

```
/* interface initialization routine */
2 int openraw( char *iface, int fd) {
    struct ifreq request;
    struct sockaddr_ll socket_address_ll;

6     /* find the interface index */
    memset( &request, 0, sizeof( request ) );
8     strncpy( request.ifr_name, iface, sizeof( request.ifr_name ) - 1 );

10    if( ioctl( fd, SIOCGIFINDEX, &request ) == -1 ) {
        perror( "ioctl(SIOCGIFINDEX) failed" );
12        return -1;
    }
14    /* bind the raw socket to the interface */
```

KAPITEL 9. ROAMEO

```

16     memset( &socket_address_ll, 0, sizeof( socket_address_ll ) );
17     socket_address_ll.sll_family = AF_PACKET;
18     /*put the interface number into the sockaddr
19      socket_address_ll.sll_ifindex = request.ifr_ifindex;
20      //get ALL packets ( look in packet(7) )
21      socket_address_ll.sll_protocol = htons( ETH_P_ALL );
22
23      if( bind( fd, (struct sockaddr *) &socket_address_ll,
24                sizeof( socket_address_ll ) ) == -1 ) {
25          perror( "bind(ETH_P_ALL) failed" );
26          return -1;
27      }
28  }

```

Ein ähnlicher Schritt ist noch für ein zu erstellendes Tunnelgerät durchzuführen, was bereits in Abschnitt 7.2 näher erläutert wurde.

In Zeile 1061 werden die Konfigurationen jener Netzwerkkarten (virtuelle und WLAN-Karte) angestoßen, indem die Funktion `configure_devices` aufgerufen wird. Diese Methode stellt beide Karten entsprechend eventuell gegebener Aufrufparameter oder nach default-Vorgaben ein. Am Ende dieser Funktion sind beide Karten im Zustand „UP“ und betriebsbereit.

Sind dann all diese Vorgaben abgearbeitet, kann die eigentlich Funktion der Software gestartet werden. Zeile 1069 beginnt so durch den Aufruf der Funktion `initial_scan` (Listing 9.2) eine Suche nach Beacons von Accesspoints mit der beim Programmaufruf definierten SSID (Parameter `-n`).

Listing 9.2: Die Funktion `initial_scan` im Detail

```

//do an initial scan for available APs and associate the best one
2 int initial_scan( void ) {
3     char in[MAXBUFFER];
4     char received_ssid[SSID_MAX_SIZE];
5     int SubType, received_ssid_len = 0;
6     ieee802_11_hdr *WiFi_header;
7     prism_hdr *Prism_header;
8     ieee802_11_beacon_mgt_frame *beacon_frame;
9     fd_set file_descriptor_set;
10    time_t start, meantime;
11    int elapsed = 0;
12    int best = 0;
13
14    FD_ZERO(&file_descriptor_set);
15    FD_SET(dev.fd_wlan, &file_descriptor_set);
16
17    time(&start);           //get start time of scanning
18    do {                  //start of 1 second loop
19        /* read from wlan card */
20        if( FD_ISSET( dev.fd_wlan, &file_descriptor_set ) ) {
21            memset( in, 0, sizeof(in) );
22            read( dev.fd_wlan, in, sizeof(in) );
23            time( &meantime );           //get current time while scanning
24            elapsed = meantime - start; //calculate elapsed time since start
25
26            //Prism header is the first data of an incoming packet.
27            //It is present, since the driver seems to be appending it...
28            Prism_header = (prism_hdr *)in;
29            //After the prism header, we find the WiFi-header
30            WiFi_header = (ieee802_11_hdr *)(in + sizeof(prism_hdr));
31            //if we have a beacon-frame, then this fits fine:
32            beacon_frame = (ieee802_11_beacon_mgt_frame *)(in+sizeof(prism_hdr) +
33                                              sizeof(ieee802_11_hdr));
34            //filter out only beacons from WLAN-traffic
35            //shift the high nibble to become the low nibble and vice versa

```

KAPITEL 9. ROAMEO

```

36         SubType = (((WiFi_header->subtype & 0xF0) >> 4) +
37             ((WiFi_header->subtype & 0xC) << 2));
38     if ( SubType==mgt_beacon ) {
39
40         //get the length of the transmitted ssid
41         if( beacon_frame->tag_number == ESSID_TAG_NUMBER )
42             received_ssid_len = (int)beacon_frame->tag_length;
43         //copy the ssid to the local variable received_ssid
44         memcpy( received_ssid, beacon_frame->tag_interpretation,
45                 received_ssid_len);
46         received_ssid[received_ssid_len] = '\0';
47
48         if ( DEBUG ) {
49             printf("Beacon seen: %s\n", received_ssid);
50         }
51
52         if ( strncmp( received_ssid, parameter.ssid,
53                         received_ssid_len ) == 0 ) {
54             if( evaluate_beacon( WiFi_header->addr2,
55                     Prism_header->signal.data ) == -1) {
56                 printf("Evaluating beacon FAILED. EXITING!\n");
57                 return -1;
58             }
59             //check if AP has best signal so far
60             if ( accesspoint_pointer->value > best ) {
61                 chosen_accesspoint = accesspoint_pointer;
62                 best = accesspoint_pointer->value;
63             }
64         }
65     }
66     accesspoint_pointer = first_accesspoint;
67     //do scanning at least for 1 second or no AP found yet
68 } while ( elapsed < 1 || first_accesspoint == NULL );
69 //set global values according the chosen AP
70 memcpy( &parameter.dst_mac, &chosen_accesspoint->mac, MAC_ADDRESS_SIZE );
71 ether_ntop( parameter.dst_mac, parameter.dst_mac_dotted );
72 //associate with the selected AP
73 assoc( parameter.dst_mac, parameter.src_mac, parameter.ssid );
74 return 0;
75 }
```

Zeile 22 von Listing 9.2 liest auf der WLAN-Karte alle Frames, die angekommen sind und speichert diese in einem Datenfeld ab. Um den Namen dieser Funktion zu rechtfertigen und eine Abbruchbedingung nach Ablauf einer Zeit von einer Sekunde zu erreichen, wird direkt nach erfolgtem Einlesen noch die Zeit genommen, bis anschließend einige Datentypen als Schablone auf das eingelesene Datenfeld gelegt werden. Nützlich ist dies deswegen, weil so intuitiv auf bestimmte Bestandteile dieses gelesenen Frames zugegriffen werden kann, um schlussendlich den empfangenen Datenstrom zu analysieren.

Im Folgenden wird nun mit verketteten Listen gearbeitet. Hierbei gibt es Zeiger, die auf bestimmte Strukturen (`structs`) zugreifen, innerhalb derer sich wiederum ein Zeiger auf Strukturen des gleichen Typs befindet. So entsteht die Möglichkeit, dynamisch zugewiesenen Speicher mit neuen Inhalten zu füllen oder wieder zu löschen, ohne dass eine Inkonsistenz dieser Liste entsteht. Im vorliegenden Fall wird diese Struktur durch Listing 9.3 dargestellt.

KAPITEL 9. ROAMEO

Listing 9.3: Ein `struct AP` das die Grundlage der globalen Pointer der benutzten verketteten Listen darstellt

```
2 //Structure to handle accesspoints
3 struct AP {
4     u_char mac[MAC_ADDRESS_SIZE];
5     int counter;
6     int value;
7     //be able to calculate average value of signal strength
8     //for one minute (if beacons arrive in 100ms interval!)
9     int signal_database[600];
10    int considerable_results_for_changing_AP;
11    struct timeval timestamp;
12    struct AP *next;
13};
```

Besondere Bedeutung innerhalb von Roameo kommt hinsichtlich der Behandlung dieser Strukturen der Funktion `evaluate_beacon` zu, die die Struktur aus Listing 9.3 sinnvoll mit Daten füllt. Dies wird durch Mittelwertberechnung der Signalstärke der bisher für einen bestimmten Accesspoint empfangenen Beacons, Einsetzen der MAC-Adresse des gefundenen Accesspoints zur Identifikation desselben und Überprüfen der bisher empfangenen Anzahl von Beacons zur Bestimmung der Sinnhaftigkeit eines möglichen Roamings inklusive Zeitstempelgeneration für das empfangene Beacon dargestellt. Außerdem findet in dieser Funktion die Entscheidung über ein durchgeführtes Roaming statt, was allerdings erst von größerem Interesse bei der späteren Hauptfunktion des Programms ist. Der anfängliche initiale Scan ruft lediglich die gleiche Pointerpflegeroutine `evaluate_beacon` auf.

Zeile 35 von Listing 9.2 zeigt eine Klassifizierung des empfangenen Datentyps, um in der Lage sein zu können, die fortlaufende Programmabfolge ausschließlich auf Beacons basierend durchzuführen (Zeile 36). Handelt es sich beim empfangenen Datenframe um ein Beacon, so wird aus diesem die empfangene SSID herausgefiltert und mit der eingestellten, zu nutzenden SSID abgeglichen (Zeilen 39 bis 49). Nur im Falle einer passenden SSID wird das Beacon ausgewertet (Zeile 50) und möglicherweise der Accesspoint als bisher stärkster herausgefiltert. Der globale Pointer `struct AP *chosen_accesspoint` wird folglich auf den Accesspoint des Beacons gesetzt und um diesen Vorgang innerhalb der gegebenen Laufzeit einer Sekunde fortlaufend wiederholen zu können der Pointer `struct AP *accesspoint_pointer` auf den Pointer `struct AP *first_accesspoint` zurückgesetzt.

Nach Ablauf der ersten Sekunde nach Programmstart wird der nun gefundene stärkste Accesspoint assoziiert und die im gesamten Programm gültige globale Variable `parameter.dst_mac` gesetzt.

Da zu diesem Zeitpunkt eine gültige Assoziation zu einem Accesspoint vorausgesetzt werden kann, beginnt von nun an die eigentliche Arbeit des Pro-

gramms. Der Aufruf der Funktion `manage_threads` startet nun vier verschiedene Threads, die fortan nebenläufig auf dem Client abgearbeitet werden.

Diese Threads, deren schematischer Aufbau innerhalb der „roameo“-Software in Abbildung 9.1 dargestellt ist, werden im Folgenden näher erläutert.



Abbildung 9.1: Schematischer Aufbau von „roameo“ mit den vier Hauptthreads

9.1 Lesen auf der virtuellen Netzwerkkarte

Daten, die auf der virtuellen Netzwerkkarte auflaufen, füllen einen Puffer von Daten auf, welcher später von dem Thread `wlan_write_thread` wieder abgearbeitet wird. Dies ist notwendig, um einen gewissen Puffer zu schaffen, in den Frames gelagert werden, falls der Schreibthread für die WLAN-Karte in einer bestimmten Situation ausgelastet ist und langsamer Frames abholen kann, als neue auflaufen. In diesem Fall wird einfach nur das nächste Frame in diesen Speicherbereich abgelegt und kann dann zum nächstmöglichen Zeitpunkt vom Schreibthread abgeholt werden. Auf diese Art und Weise kann sichergestellt werden, dass keine empfangenen Frames verloren gehen und diese in schnellstmöglicher Zeit abgearbeitet werden.

Der kritische Speicherbereich der abgelegten Frames wird hier durch ein `pthread_mutex_lock` in Zeile 14 von Listing 9.4 geschützt. Ist dieser Bereich also gesperrt durch die Lesefunktion auf der virtuellen Netzwerkkarte, so kann kein anderer Thread Daten von hier abholen. Die Speicherkonsistenz ist somit sichergestellt. Diese Lösung ist auch bekannt als Lösung des Erzeuger/Verbraucher Problems.

Listing 9.4: Der Thread zum Lesen von der virtuellen Netzwerkkarte

```

1 void *read_from_vnic(void *arg) {
2     int array_counter, size;
3     char data[MAXBUFFER];
4
5     array_counter = 0;
6     while(1) {
7         //reset buffer
8         memset( &data, 0, MAXBUFFER );
9         //read on vnic
10        size = read( dev.fd_vnic, data, MAXBUFFER );
11        //set empty down by 1
12        sem_wait( &vnic_empty );
13        //lock the mutex
14        pthread_mutex_lock( &mutex[array_counter] );
15        //reset buffer
16        memset( &vnic_data[array_counter], 0, sizeof(struct data) );
17        //copy the data read to vnic_data-array
18        vnic_data[array_counter].size = size;
19        memcpy( vnic_data[array_counter].data, data,
20                vnic_data[array_counter].size );
21        //unlock the mutex
22        pthread_mutex_unlock( &mutex[array_counter] );
23        //set vnic_full up by one
24        sem_post( &vnic_full );
25        //increase array_counter by one
26        array_counter = ( array_counter + 1 ) % DATA_BUFFER;
27    }
28    return NULL;
}

```

9.2 Schreiben auf der WLAN-Karte

Die Funktion `write_on_wlan` (Listing 9.5) arbeitet die durch Listing 9.4 aufge laufenen Daten ab, indem sie sie in eine für das WLAN kompatible Struktur ändert und dann auf die WLAN-Karte schickt, wo sie in das drahtlose Medium versendet werden. In einer Endlosschleife wird mittels einer Abfrage von `sem_wait` [24] auf Existenz von Daten im abzuarbeitenden Datenpuffer ge prüft. Liegen hier Frames zur Abarbeitung vor, so wird wiederum mit einem `pthread_mutex_lock` der entsprechende kritische Datenbereich gesperrt und die Daten ausgelesen. Ab Zeile 31 in Listing 9.5 wird der WLAN-Header entspre chend der Informationen aus dem Ethernet-Frame aufgebaut, anschließend zusammengesetzt und schlussendlich auf die WLAN-Karte geschrieben. Eine vollständige Konvertierung der Ethernetdaten in ein WLAN-verträgliches Format ist somit durchgeführt worden.

Durch Implementation der Funktionen aus Listing 9.4 und Listing 9.5 ist ein Datentransfer vom Client zum Accesspoint möglich.

Listing 9.5: Datenanpassung für WLAN-Frames

```

void *write_on_wlan(void *arg) {
1     int array_counter, size;
2     char out[MAXBUFFER], data[MAXBUFFER];
3     struct ether_header *eth;
4     ieee802_11_hdr *WiFi_header;
5     struct fddi_snap_hdr *snap;
6

```

```

8     array_counter = 0;
9     while(1) {
10
11         //reset buffers
12         memset(out, 0, MAXBUFFER);
13         memset(data, 0, MAXBUFFER);
14         //wait for vnic_full to contain an object
15         sem_wait(&vnic_full);
16         //lock the mutex
17         pthread_mutex_lock(&mutex[array_counter]);
18         //copy data of global variable to local variables
19         memcpy(data, vnic_data[array_counter].data,
20                vnic_data[array_counter].size);
21         size = vnic_data[array_counter].size;
22         //reset global memory
23         memset(&vnic_data[array_counter], 0, sizeof(struct data));
24         //unlock the mutex
25         pthread_mutex_unlock(&mutex[array_counter]);
26
27         //set vnic_empty up by one
28         sem_post(&vnic_empty);
29
30         if ( size > sizeof(struct ether_header) ) {
31             eth = (struct ether_header *)data;
32             /* check if it is an IP or ARP Ethernet frame */
33             if ( ntohs(eth->ether_type) == ETHERTYPE_IP ||
34                 ntohs(eth->ether_type) == ETHERTYPE_ARP ) {
35
36                 /* prepare WLAN header */
37                 WiFi_header = (ieee802_11_hdr *)out;
38                 WiFi_header->subtype = IEEE80211_TYPE_DATA;
39                 WiFi_header->flags = IEEE80211_TO_DS;
40                 memcpy(&WiFi_header->duration, "\xD4\x00", 2);
41
42                 /* copy addresses */
43                 memcpy(WiFi_header->addr1, parameter.dst_mac, ETH_ALEN);
44                 memcpy(WiFi_header->addr2, eth->ether_shost, ETH_ALEN);
45                 memcpy(WiFi_header->addr3, eth->ether_dhost, ETH_ALEN);
46
47                 /* prepare SNAP header, LLC, defaults were
48                  retrieved from header include file */
49                 snap = (struct fddi_snap_hdr *)(((char *)WiFi_header)) +
50                         sizeof(ieee802_11_hdr));
51                 snap->dsap = 0xaa;
52                 snap->ssap = 0xaa;
53                 snap->ctrl = 0x03;
54                 snap->ethertype = eth->ether_type;
55
56                 /* copy payload */
57                 size = size - sizeof(struct ether_header);
58                 memcpy(((char *)WiFi_header)+32, ((char *)eth) +
59                         sizeof(struct ether_header), size);
60
61                 write(dev.fd_wlan, out, size+sizeof(ieee802_11_hdr) +
62                         sizeof(struct fddi_snap_hdr));
63             }
64         }
65         array_counter = (array_counter + 1) % DATA_BUFFER;
66     }
67     return NULL;
68 }
```

9.3 Lesen auf der WLAN-Karte

Um die Daten des WLANs korrekt verarbeiten zu können, wird wieder mit Hilfe von Threads jedes Frame von der WLAN-Karte durchgereicht und vergleichbar der Technik aus Abschnitt 9.1 in einen Framespeicher abgelegt.

Jedes Frame auf der WLAN-Karte wird in einen definierten Speicher gelegt, wo es anschließend von Abschnitt 9.4 zur weiteren Verarbeitung abgerufen werden kann.

Listing 9.6: Lesen von Daten auf der WLAN-Karte

```

void *read_from_wlan(void *arg) {
    2     int array_counter, size;
    3     char data[MAXBUFFER];
    4
    5     array_counter = 0;
    6     while(1) {
    7         //reset data-buffer
    8         memset( &data, 0, MAXBUFFER );
    9         //read on wlan
   10        size = read( dev.fd_wlan, data, MAXBUFFER );
   11        //set wlan_empty down by 1
   12        sem_wait( &wlan_empty );
   13        //lock the mutex
   14        pthread_mutex_lock( &mutex[array_counter] );
   15        //reset buffer
   16        memset( &wlan_data[array_counter], 0, sizeof(struct data) );
   17        //copy the data read to vnic_data-array
   18        wlan_data[array_counter].size = size;
   19        memcpy( wlan_data[array_counter].data, data,
   20                wlan_data[array_counter].size );
   21        //unlock the mutex
   22        pthread_mutex_unlock( &mutex[array_counter] );
   23        //set wlan_full up by 1
   24        sem_post( &wlan_full );
   25        //increase array_counter by 1
   26        array_counter = ( array_counter + 1 ) % DATA_BUFFER;
   27    }
   28    return NULL;
}

```

9.4 Schreiben auf der virtuellen Netzwerkkarte

Sämtliche Frames, die sich nach Abarbeitung durch Listing 9.5 ansammeln, werden im aufwändigsten der vier Threads abgearbeitet. In diesem Falle ist es notwendig, sämtliche Frames nicht nur schlicht für das entsprechend andere Protokoll abzuändern, sondern diese zuvor auszuwerten hinsichtlich ihres Typs. Auf der WLAN-Karte, welche den abzuarbeitenden Datenpuffer für die Funktion `write_on_vnic` speist, laufen neben Datenframes auch Managementframes wie etwa Beacons auf. Ist unter den Daten ein solches Beacon, so muss dieses ausgewertet werden und eventuell eine Umassoziation durchgeführt werden (durch `evaluate_beacons`).

Zu diesem Zweck findet sich in Listing 9.7 in den Zeilen 40 und 43 eine Überprüfung des Datentyps innerhalb des WLAN-Frames. Handelt es sich um ein Beacon, so greift eine Managementroutine ähnlich der aus Listing 9.2. In diesem Fall findet sich auch in den Zeilen 78 bis 83 von Listing 9.7 der eigentliche Befehl zum Roamen. Zunächst wird der alte Accesspoint dissoziiert (Zeile 78), anschließend der neue Accesspoint assoziiert (Zeile 83). Hinter diesen beiden Befehlen verbergen sich zwei Routinen aus der include-Datei

KAPITEL 9. ROAMEO

`wlan_mgmt.c`, welche auf der beigefügten CD im Quellcode ersichtlich sind.

Im Falle von abzuarbeitenden Datenframes wird der Ethernethheader aus dem WLAN-Frame gewonnen, das Frame zusammengebaut und zum Schluss auf die virtuelle Netzwerkkarte geschrieben (Zeile 137 in Listing 9.7).

Der komplette Quellcode der entwickelten Software ist in Anhang B verfügbar.

Listing 9.7: Die Datenbehandlung von WLAN-Frames zur Weiterleitung auf das Ethernet

```
void *(void *arg) {
2     (...)

4     while(1) {
6         (...)

8         //wait for wlan_full to contain an object
10        sem_wait( &wlan_full );
11        //lock the mutex
12        pthread_mutex_lock( &mutex[array_counter] );
13        //copy data of global variable to local variables
14        memcpy( data, wlan_data[array_counter].data,
15                wlan_data[array_counter].size );
16        size = wlan_data[array_counter].size;
17        //reset global memory
18        memset( &wlan_data[array_counter], 0, sizeof(struct data) );
19        //unlock the mutex
20        pthread_mutex_unlock( &mutex[array_counter] );
21        //set vnic_empty up by one
22        sem_post( &wlan_empty );

24        (...)

26        /*check for beacons to be evaluated according to signal strength */
27        if ( SubType == mgt_beacon ) {
28            (...)

30            //check if this accesspoint is the chosen one
31            //and check if we have already enough results to consider switching
32            //and check if accesspoint is already associated
33            if ( (roam == TRUE) &&
34                ( accesspoint_pointer->considerable_results_for_changing_AP == 1 )
35                && ( memcmp( accesspoint_pointer->mac, parameter.dst_mac
36                            , MAC_ADDRESS_SIZE ) != 0 ) ) {
37                (...)

38                chosen_accesspoint->considerable_results_for_changing_AP = 0;
39                chosen_accesspoint = accesspoint_pointer;
40                disassoc( parameter.dst_mac, parameter.src_mac, parameter.ssid );
41                //set global values according the chosen AP
42                memcpy( &parameter.dst_mac, accesspoint_pointer->mac,
43                        MAC_ADDRESS_SIZE );
44                ether_ntop( parameter.dst_mac, parameter.dst_mac_dotted );
45                //associate with the selected AP
46                assoc( parameter.dst_mac, parameter.src_mac, parameter.ssid );
47                roam = FALSE;
48            } else {
49                (...)

50            }
51        }
52    }
53}
54}
55}
56}
57}
58}
```

KAPITEL 9. ROAMEO

```
60      }
62      /* filter out anything but data */
63      if ((SubType & 0xF0) == 0x20 && (SubType & 0xF) < 4) {
64          (Aenderung der Frames in eine ethernetkonforme Struktur)
65          (...)

66      }
67      (...)

68      }
69      (...)

70      (...)

71      }
72      return NULL;
73  }
```

Kapitel 10

Testergebnisse

10.1 Höherer Aufwand pro Frame im WLAN

Allgemein gilt festzuhalten, dass die Übertragung von Daten im WLAN fehleranfälliger ist als im Ethernet. Der größte Einflussfaktor im WLAN ist sicher die Tatsache, dass es in der Regel die Luft als drahtloses Übertragungsmedium nutzt. Es ist somit vielen äußeren Einflüssen wie Störsignalen oder auch Signalüberlagerungen, die durch um das Medium konkurrierende Stationen erzeugt werden, ausgesetzt. Abhängig vom Übertragungsmedium liegt die relative Fehlerhäufigkeit bei

- Lichtwellenleiter $10^{-12} - 10^{-14}$
- Kupfer $10^{-8} - 10^{-9}$
- Luft $10^{-5} - 10^{-3}$

Hierbei ist anzumerken, dass Fehler häufig in kurzen Volllasten (sogenannten Bursts) auftreten.[23]

Um nun Fehler zu erkennen und beheben zu können, gibt es Paritätsprüfungen oder Prüfsummenbildungen, die größeren Overhead und damit erhöhtes Datenvolumen erzeugen.

Weiterhin wird der Overhead vom in Kapitel 5 beschriebenen WLAN-Header beeinflusst, welcher ungleich höher ist als jener im Ethernet. Dieser zusätzliche Overhead zusammen mit dem Koordinationsaufwand ist mitunter ein Hauptgrund für höhere Latenzzeiten im WLAN als beispielsweise im Ethernet. Des Weiteren spielen das voll auf Bestätigungen (im WLAN die Acknowledegeframes (Kapitel 5)) basierende Protokoll und die Unsicherheit des

Mediums eine bedeutende Rolle. Es besteht also die Möglichkeit von wiederholten Frameversendungen und außerdem kann es aufgrund von Managementinstanzen, die Framekollisionen vermeiden sollen, zu Verzögerungen kommen.

Der eigentliche Grund all dieser im WLAN anzutreffenden Sonderheiten ist die physikalische Unsicherheit der Verbindung. Es ist eben keine Kupferleitung gegeben.

Eine schöne Analogie ist die Vorstellung, dass es möglich ist, sich auf einer vom Bund gewarteten Autobahn sehr schnell fortzubewegen, während man im offenen Gelände weitaus langsamer voran kommt. Der Nachteil der Autobahn ist in diesem Falle die Einschränkung der Wege, die befahren werden können, was wiederum mit einem Amphibienfahrzeug keine Einschränkung ist. Dieses Gefährt kann sich mühelos an jeden Punkt im Gelände bewegen, ist jedoch bedeutend langsamer. Die Flexibilität ist hier also ungleich höher. Das Ziel von WLAN kann also nicht sein, das Kabelnetzwerk zu ersetzen, sondern es hinsichtlich Mobilität zu erweitern.

Zum Beweis des höheren Overheads und damit ungleich höherer Paketlaufzeiten sei auf einen Mitschnitt von Rundlaufzeiten verwiesen (siehe Abbildung 10.4). Bei allen Zeiten, die im Rahmen von Tests aufgezeichnet wurden, ist immer zu beachten, dass es sich um Ergebnisse jeweils **eines** exemplarisch gewählten Testergebnisses unter vielen handelt. Schwankungen **sämtlicher** Zahlen waren durchgehend zu beobachten.

10.2 Messung von Rundlaufzeiten ohne Roaming

10.2.1 Round-Trip-Time Messsoftware

Zur Analyse von aufgebauten Verbindungen bezüglich Stabilität, Zuverlässigkeit und Geschwindigkeit wurde ein kleines Round-Trip-Time (RTT) Programm entworfen, welches UDP-Pakete zwischen zwei verbundenen Rechnern verschickt. Ein Rechner mimt hierbei einen Spiegel, der alle ankommenden Pakete auf einem definierten Port zurücksendet, ohne diese zu interpretieren, der andere Rechner analysiert diese Pakete hinsichtlich der Tatsache, ob ein versendetes Paket angekommen ist und welche Zeit es dafür benötigt hat. Hierzu wird beim Versenden eines Paketes die aktuelle Zeit in das Paket geschrieben und in dem Moment, zu dem es wieder beim Ausgangsrechner ankommt, mit der aktuellen Zeit verglichen. Die Differenz ergibt dann die

Rundlaufzeit des Paketes.

Um das RTT nutzen zu können, ist es von Vorteil, die ARP-Einträge auf beiden genutzten Maschinen statisch zu setzen. Wird dieser Schritt nicht gemacht, so werden alle 40 bis 50 Sekunden ARP-Anfragen versandt, die die Testergebnisse beeinflussen, da in dieser Zeit Totzeiten auftreten können. Diese sind damit zu begründen, dass UDP als Protokoll des RTT-Tests verwendet wird. Es ist sinnvoll, UDP für die RTT-Tests zu verwenden, um jedwede Korrektur durch das verbindungsorientierte TCP auszuschließen, mit dem Zweck, verlorene Pakete sicher aufzuspüren. Einen statischen ARP-Eintrag setzt man mit dem Befehl arp(8). Ein statischer Eintrag für eine Netzwerkkarte mit der MAC-Adresse 11 : 22 : 33 : 44 : 55 : 66 und der IP-Adresse 10.0.0.1 wird unter Linux wie folgt in einer Konsole erstellt:

```
arp -s 10.0.0.1 11:22:33:44:55:66
```

Fehlende Pakete werden mit einer speziellen Auswertungssoftware erkannt. Beispielsweise war im Logfile eines Tests die Information zu finden, dass das Paket 1445 fehlt (siehe Listing 10.1). Ein gleichzeitig durchgeföhrter Sniff mit Wireshark auf dem RTT-echo Rechner beweist, dass dieses Paket tatsächlich nicht angekommen ist. Bei der Bildbetrachtung des Wireshark-Mitschnittes ist zu beachten, dass Wireshark Daten im hexadezimalen Format anzeigt.

Listing 10.1: Verlorene Pakete bei einem RTT-Test

```
2      #missing packet 1444
      1445 562
      1446 511
```

10.2.2 RTT-Ergebnisse lastfrei

Abbildung 10.4 zeigt die Laufzeiten von 30000 Paketen in drei verschiedenen Netzwerkkonfigurationen. Mit großem Abstand die besten Werte liefert hier das Ethernet, welches eine durchschnittliche Zeit von $450 \mu\text{s}$ liefert. Eine WLAN-Karte der Firma Atheros im Stationsmodus liefert hier schon einen ungleich höheren Durchschnittswert: $1007 \mu\text{s}$, was etwas mehr als dem Doppelten der Ethernetlaufzeit entspricht.

Die Maximalzeiten unterschieden sich in allen durchgeföhrten Tests jedoch nur marginal. Das Ethernet kommt im abgebildeten Beispiel auf $23866 \mu\text{s}$, die WLAN-Karte auf nur unwesentlich höhere $27646 \mu\text{s}$, ein Zuwachs von etwa 15%.

KAPITEL 10. TESTERGEBNISSE

2295 64.403165	10.0.0.107	10.0.0.179	UDP	Source port: 32817 Destination port: 4711
2296 64.403242	10.0.0.179	10.0.0.107	UDP	Source port: 32875 Destination port: 4711
2297 64.515164	10.0.0.107	10.0.0.179	UDP	Source port: 32817 Destination port: 4711
2298 64.515241	10.0.0.179	10.0.0.107	UDP	Source port: 32875 Destination port: 4711
2299 64.571420	10.0.0.107	10.0.0.179	UDP	Source port: 32817 Destination port: 4711
2300 64.571522	10.0.0.179	10.0.0.107	UDP	Source port: 32875 Destination port: 4711
0000 00 e0 4c 07 b0 55 00 14	85 25 f1 99 08 00 45 00	..L..U.. .%....E.		
0010 00 28 e3 f0 40 00 40 11	41 b7 0a 00 00 6b 0a 00	.(..@. A....k..		
0020 00 b3 80 31 12 67 00 14	33 81 a4 05 00 00 a4 0b	...1.g.. 3.....		
0030 cb 45 0c 38 05 00 00 00	0e 56 eb fd	.E.8.... .V..		

Abbildung 10.1: Das letzte Paket mit erreichtem Ziel (Paket 1444)

2295 64.403165	10.0.0.107	10.0.0.179	UDP	Source port: 32817 Destination port: 4711
2296 64.403242	10.0.0.179	10.0.0.107	UDP	Source port: 32875 Destination port: 4711
2297 64.515164	10.0.0.107	10.0.0.179	UDP	Source port: 32817 Destination port: 4711
2298 64.515241	10.0.0.179	10.0.0.107	UDP	Source port: 32875 Destination port: 4711
2299 64.571420	10.0.0.107	10.0.0.179	UDP	Source port: 32817 Destination port: 4711
2300 64.571522	10.0.0.179	10.0.0.107	UDP	Source port: 32875 Destination port: 4711
0000 00 e0 4c 07 b0 55 00 14	85 25 f1 99 08 00 45 00	..L..U.. .%....E.		
0010 00 28 e3 f2 40 00 40 11	41 b5 0a 00 00 6b 0a 00	.(..@. A....k..		
0020 00 b3 80 31 12 67 00 14	a9 cb a6 05 00 00 a4 0b	...1.g..		
0030 cb 45 92 ed 06 00 00 00	a7 2c 8a 4c	.E..... ,.L		

Abbildung 10.2: Das erste Paket nach verlorenem Paket (Paket 1446)

2295 64.403165	10.0.0.107	10.0.0.179	UDP	Source port: 32817 Destination port: 4711
2296 64.403242	10.0.0.179	10.0.0.107	UDP	Source port: 32875 Destination port: 4711
2297 64.515164	10.0.0.107	10.0.0.179	UDP	Source port: 32817 Destination port: 4711
2298 64.515241	10.0.0.179	10.0.0.107	UDP	Source port: 32875 Destination port: 4711
2299 64.571420	10.0.0.107	10.0.0.179	UDP	Source port: 32817 Destination port: 4711
2300 64.571522	10.0.0.179	10.0.0.107	UDP	Source port: 32875 Destination port: 4711
0000 00 e0 4c 07 b0 55 00 14	85 25 f1 99 08 00 45 00	..L..U.. .%....E.		
0010 00 28 e3 f3 40 00 40 11	41 b4 0a 00 00 6b 0a 00	.(..@. A....k..		
0020 00 b3 80 31 12 67 00 14	e3 f0 a7 05 00 00 a4 0b	...1.g..		
0030 cb 45 56 c8 07 00 00 00	55 e2 e8 ac	.EV..... U...		

Abbildung 10.3: Das nächste Paket im Transfer (Paket 1447)

„roameo“-RTT-Ergebnisse lastfrei

Die für das Roaming genutzte Software „roameo“ kommt lastfrei auf vergleichbare Werte wie eine WLAN-Karte im Stationsmodus. Bei allen durchgeföhrten Tests war jedoch eine durchschnittlich höhere RTT-Zeit mit roaming-Software zu verzeichnen, was sich durch die Tatsache, dass jedes Paket durch das Programm läuft, erklären lässt. Hier entstehen durch die Behandlung und Änderung der Frames zusätzliche Zeiten, die sich anschließend in durchschnittlich höheren Paketlaufzeiten widerspiegeln. Da dieser Wert relativ zum Ethernet gesehen jedoch in einem vergleichbaren Rahmen wie der Stationsmodus zum Ethernet ist, stellt diese Tatsache kein Problem dar. In Zahlen ausgedrückt ist das WLAN im arithmetischen Mittel unbelastet etwa 2,2 mal, das WLAN mit roaming-Software 2,3 mal langsamer als das Ethernet.

Auch die Maximalwerte der Rundlaufzeiten verhalten sich unbelastet unverändert. In Abbildung 10.4 findet sich für die WLAN-Karte mit roaming-Software ein Wert von $26842 \mu\text{s}$, welcher gar besser ist, als jener für die

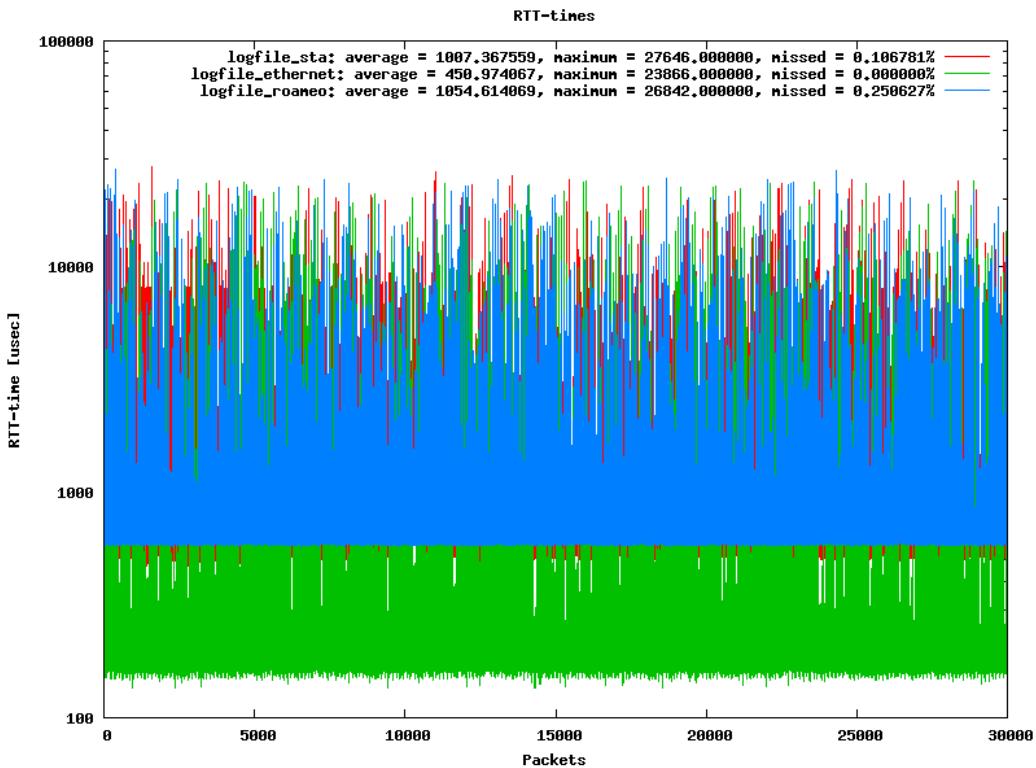


Abbildung 10.4: RTT-Laufzeiten im unbelasteten Zustand

WLAN-Karte im Stationsmodus. Diese Tatsache dieses exemplarischen Versuchsergebnisses zeigt, dass es sich bei diesen Maximalwerten um leicht variable Werte handelt, die je nach Störungen im drahtlosen Medium oder anderen Einflüssen schwanken, was z.B. in der Nutzung eines nicht echtzeitfähigen Kernel begründet ist (Kernel 2.6.15).

Des Weiteren kann aus dieser Beobachtung abgeleitet werden, dass „roameo“ keineswegs höhere Maximalwerte in der Rundlaufzeit fördert. Die WLAN-Karte scheint in beiden Modi, dem Stationsmodus wie dem Monitormodus, äußerer Einflüssen gleichermaßen ausgeliefert zu sein, welche dann Maximalwerte hervorrufen.

10.2.3 RTT-Ergebnisse belastetes Medium

Die in diesem Abschnitt behandelten Ergebnisse wurden erstellt, während iperf [25] das jeweils genutzte Netzwerk belastete. Die Hauptaussage dieser

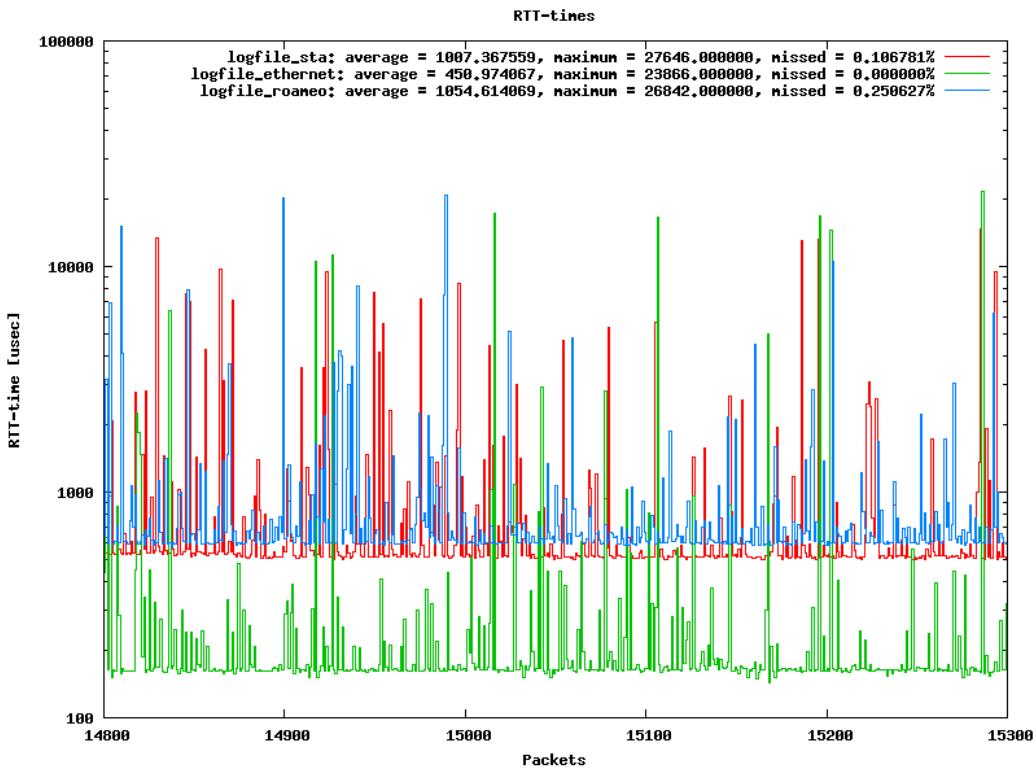


Abbildung 10.5: Vergrößerte Darstellung der Testergebnisse auf Abbildung 10.4

Tests, deren graphische Auswertung Abbildung 10.6 und Abbildung 10.7 darstellen, ist, dass sich die Rundlaufzeiten kaum verändern oder gar besser werden. Vergleicht man also die Werte des unbelasteten Netzwerkes mit denen des belasteten Netzwerkes beispielsweise im Ethernet, so sinkt der Durchschnittswert gar um $53,7 \mu\text{s}$, während der Maximalwert um $122 \mu\text{s}$ steigt. Diese Zahlen sind derart gering, dass sie wohl eher dem üblichen Rauschen zuzuordnen sind, als dass ein Einfluss durch iperf vermutet werden kann.

Auch die Ergebnisse der beiden kabellosen Verbindungsmöglichkeiten lassen kaum Spielraum zum Analysieren. Der Wert für „roameo“ bleibt fast gleich, der Wert für den Stationsmodus steigt zwar merklich an, in Relation zu vielen anderen Messreihen ist dies jedoch nur ein Ausbruch nach oben, der in anderen Messungen in geringerem Rahmen nach unten ging.

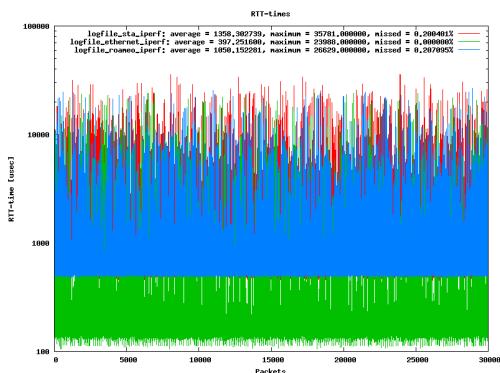


Abbildung 10.6: RTT-Lauffzeiten unter Belastung von `iperf`

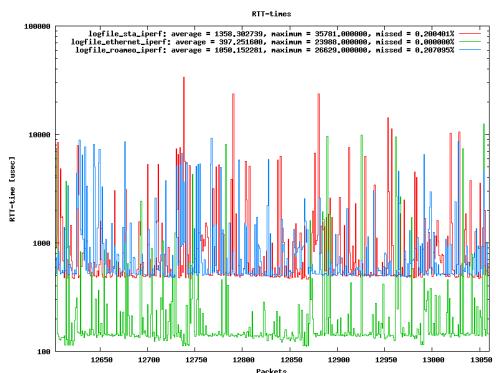


Abbildung 10.7: Vergößerte Ansicht von Abbildung 10.6

10.3 Messung von Rundlaufzeiten mit Roaming

10.3.1 Roaming mit MadWiFi-0.9.2.1

Eine wirklich interessante und aussagekräftige Auswertung ergibt sich hingegen bei Messungen einer belasteten Leitung mit durchgeföhrtem Roaming. Das Problem von Lösungen im Stationsmodus ist jederzeit die Erkennung neuer Accesspoints, da diese nur bei einem durchgeföhrten Scan gefunden werden. Ist die Leitung jedoch lange durch hohen Datendurchsatz ausgelastet, so kann kein Scan abgearbeitet werden. Ein Roaming auf einen möglicherweise seit langer Zeit wesentlich stärkeren Accesspoint ist ausgeschlossen, da dieser bessere Accesspoint gar nicht erkannt wird.

Standardmäßig verhielt sich der MadWiFi-Treiber sogar vollkommen ungeeignet für ein vergleichbar schnelles Roaming, da eine Umassoziation in diesem Modus nur erfolgte, wenn die Verbindung zu einem assoziierten Accesspoint verloren ging. Nach Ablauf von einigen Sekunden wurde die verlorene Verbindung vom Treiber registriert (ein `iwconfig` meldete „Access Point: Not-Associated“) und ein Scan wurde gestartet. Leider versuchte der Treiber daraufhin mehrere Male, sich wieder mit dem zuvor assoziierten Accesspoint zu verbinden, auch wenn dieser nur in sehr schlechter Reichweite war. Erst nach mehreren Fehlversuchen gelang es dann schlussendlich einen neuen, stärkeren Accesspoint zu assoziieren. Eine Beschleunigung dieses Verhaltens, vor allem aber ein „vorausschauendes“ Verhalten des Clients, indem er permanent nach stärkeren Accesspoints sucht, auch wenn noch eine Verbindung besteht, um Verbindungsabrissen vorzubeugen, ist im Standardtreiber nicht implementiert.

Die Lösung aus Abschnitt 6.2 versucht genau diesen Weg zu gehen, da hier permanent Scans ausgesendet und die Ergebnisse ausgewertet werden. Ist bereits vor einem Verbindungsabriss ein anderer Accesspoint signaltechnisch stärker, so wird umgehend umassoziiert.

Das Problem, dass bei ausgelasteter Leitung kein Scan abgearbeitet werden kann, bleibt jedoch auch hier bestehen. So liefert eine wirkliche Lösung nur die in dieser Diplomarbeit erarbeitete „roameo“-Software, da hier kein Paketverlust gemessen wurde und die Abhängigkeit von Scans mit dem Nachteil der Beschränkung auf einen Kanal gelöst wurde.

RTT-Ergebnisse mit MadWiFi-Roaming

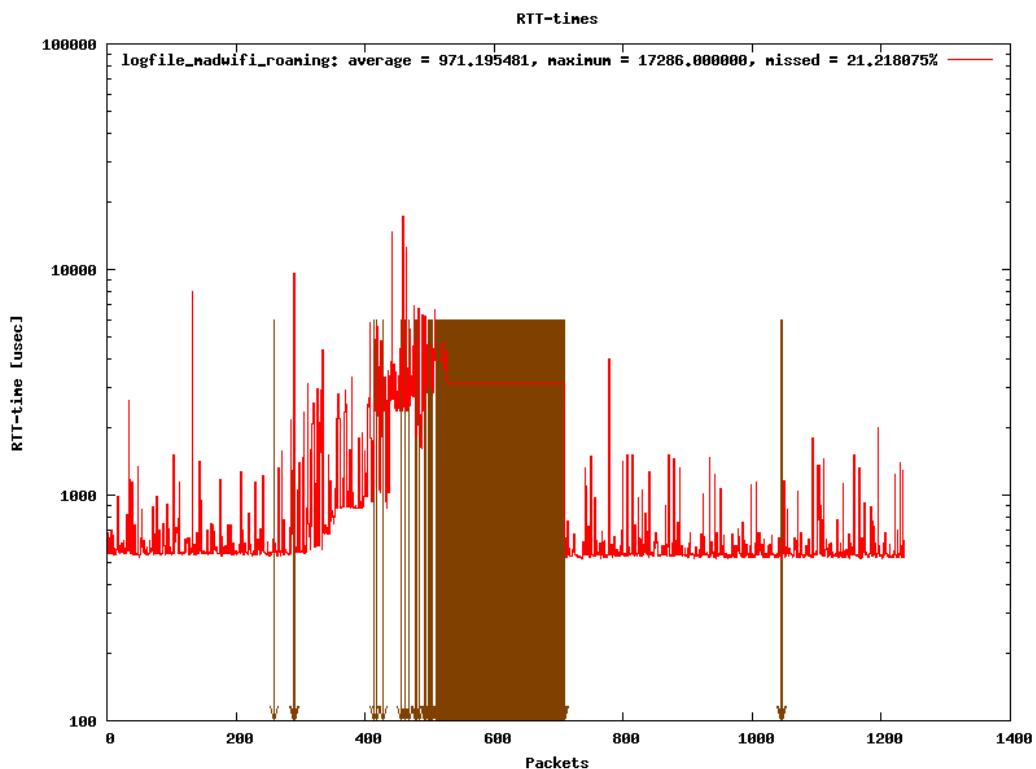


Abbildung 10.8: Ein Roaming mit dem standard MadWiFi-Treiber - braune Pfeile entsprechen verlorenen Paketen

Abbildung 10.8 zeigt den RTT-Mitschnitt eines Roamings mit dem MadWiFi-Treiber in seiner zum Zeitpunkt der Diplomarbeit aktuellen Version 0.9.2.1

mit zwei Accesspoints. Die braunen Pfeile stellen verlorene RTT-Pakete dar. Auffällig ist neben 207 verlorenen Paketen bei diesem Versuchsaufbau, dass die RTT-Zeiten vor dem Verbindungsverlust ansteigen. Dies ist darauf zurückzuführen, dass sich die Qualität der Verbindung kontinuierlich verschlechtert, um letztendlich abzureißen. Eine progressive Früherkennung eines schlechten Links ist hier also nicht implementiert, was dieses Versuchsergebnis zeigt. Erst bei totalem Verbindungsverlust wird aktiv und auf allen Kanälen nach Alternativen gesucht und letztendlich eine neue Verbindung aufgebaut.

Während der Versuche fiel auf, dass sich der Client selbst bei Verbindungsverlust nicht zwangsläufig zum besten gefundenen Accesspoint verbündet. Oft konnte beobachtet werden, dass der Treiber versuchte, sich zum alten, zuvor assoziierten Accesspoint zurückzuverbinden, wohl weil nach einem Scan noch Beacons des schwachen Accesspoints zu empfangen waren, nur eben die Verbindung dann nicht stark genug war, um auch Datentransfer sicherzustellen. In diesem Fall dauerte es wiederum einige Sekunden, bis diese Assoziation fallen gelassen wurde und ein neuer Scan angestoßen wurde bis nach einer nicht näher definierbaren Anzahl von Versuchen letztlich ein stärkerer Accesspoint gefunden wurde.

10.3.2 Roaming mit roameo

RTT-Ergebnisse mit „roameo“-Roaming

Dem Ergebnis aus Abbildung 10.9 liegt exakt derselbe Versuchsaufbau wie in Abschnitt 10.3.1 zugrunde. Der grüne Pfeil kennzeichnet die Übergabe des Clients von einem der beiden Accesspoints zum anderen. Abbildung 10.10 zeigt noch wesentlich deutlicher, dass die RTT-Zeiten offenbar keinen Einfluss durch das Roaming nehmen. Ebenso ist der Legende von Abbildung 10.10 zu entnehmen, dass hier kein Paket verloren wurde.

Die Performance ist dauerstabil und verändert sich auch durch Last auf der Netzwerkkarte nicht. Ersichtlich ist dies aus Abbildung 10.11, bzw. Abbildung 10.12, denen zusätzlich eine mit iperf belastete Netzwerkkarte zugrunde liegt.

Besonders interessant in Abbildung 10.11 ist die Legende, der zu entnehmen ist, dass die Paketverluste bei einem Versuch mit einer derart großen Anzahl an Roamings vergleichbar zu denen sind, die die WLAN-Karte ohne Roaming im Stationsmodus ($\sim 0.2\%$) bzw. im Monitormodus mit „roameo“ ($\sim 0.2\%$) erfährt. Auch der Durchschnittswert der RTT-Zeiten sowie die ma-

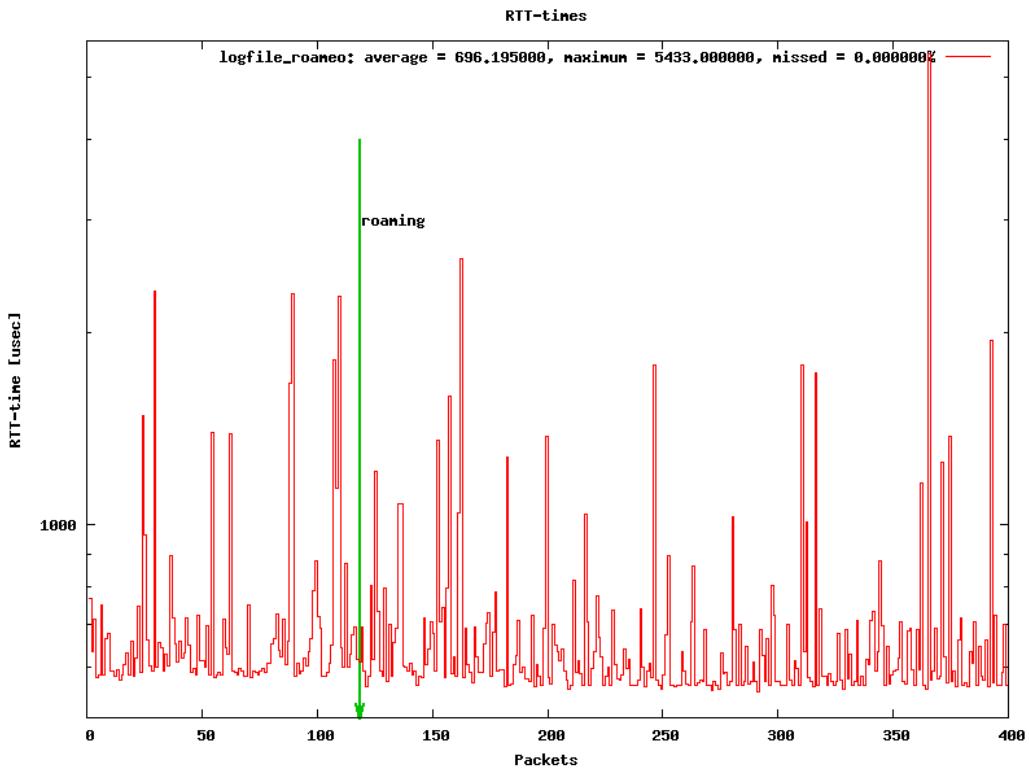


Abbildung 10.9: Ein Roaming mit dem Roamingalgorithmus dieser Diplomarbeit

ximale Laufzeit weichen kaum von denen in Abbildung 10.4 ab.

Einen weiteren Nachweis für die Leistungsfähigkeit der vorgestellten Lösung liefert ein Wireshark-Mitschnitt, der UDP-Pakete aus dem RTT-Programm und einen Roamingvorgang zeigt (illustriert in Abbildung 10.13). Zwischen dem letzten Acknowledge vom Accesspoint und dem ersten RTT-Paket vom Client zum Accesspoint vergehen 2,81ms. Theoretisch wäre noch eine Steigerung dieses Wertes möglich gewesen, da das Roaming im abgebildeten Fall in 2,325ms durchgeführt wurde. Effektiv wurde also innerhalb dieser Zeit ein Roaming durchgeführt.

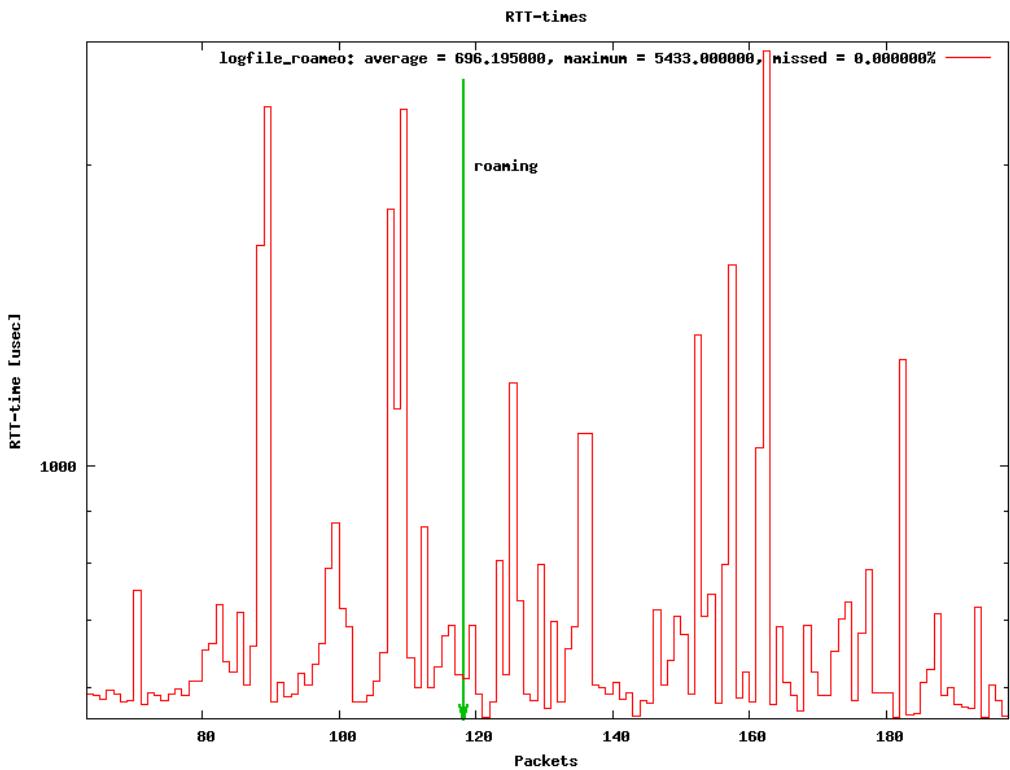


Abbildung 10.10: Das Roaming aus 10.9 in vergrößerter Darstellung des interessanten Bereiches

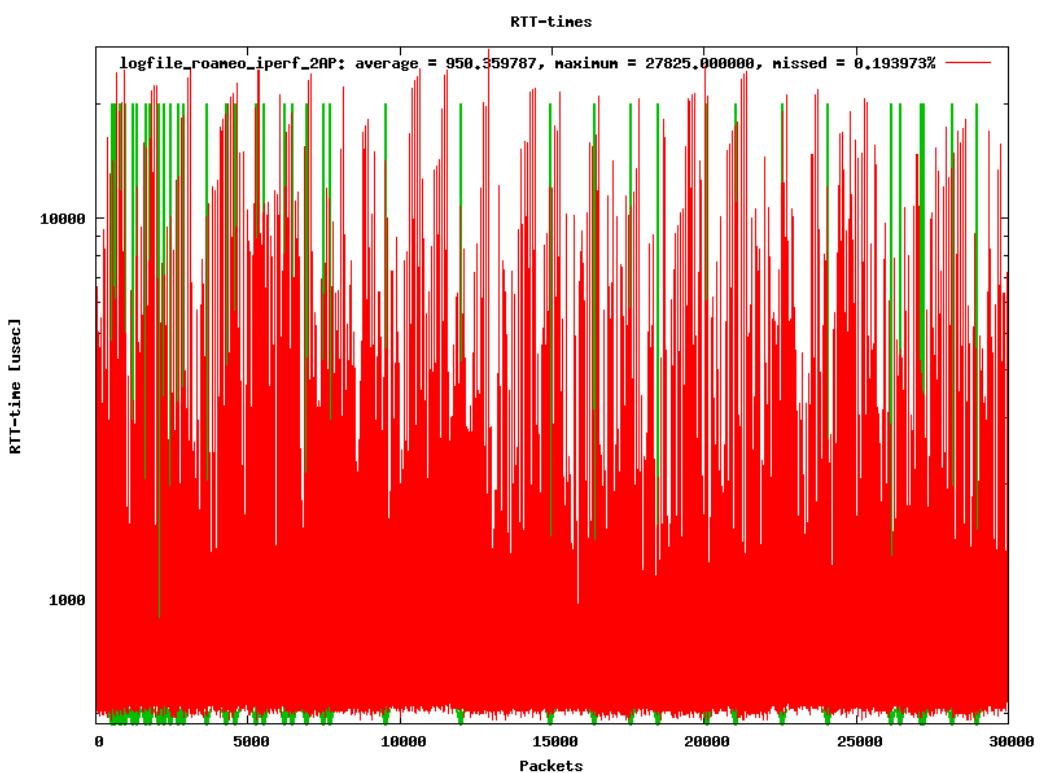


Abbildung 10.11: 30000 RTT-Pakete mit 41 Accesspointwechseln mit „roameo“

KAPITEL 10. TESTERGEBNISSE

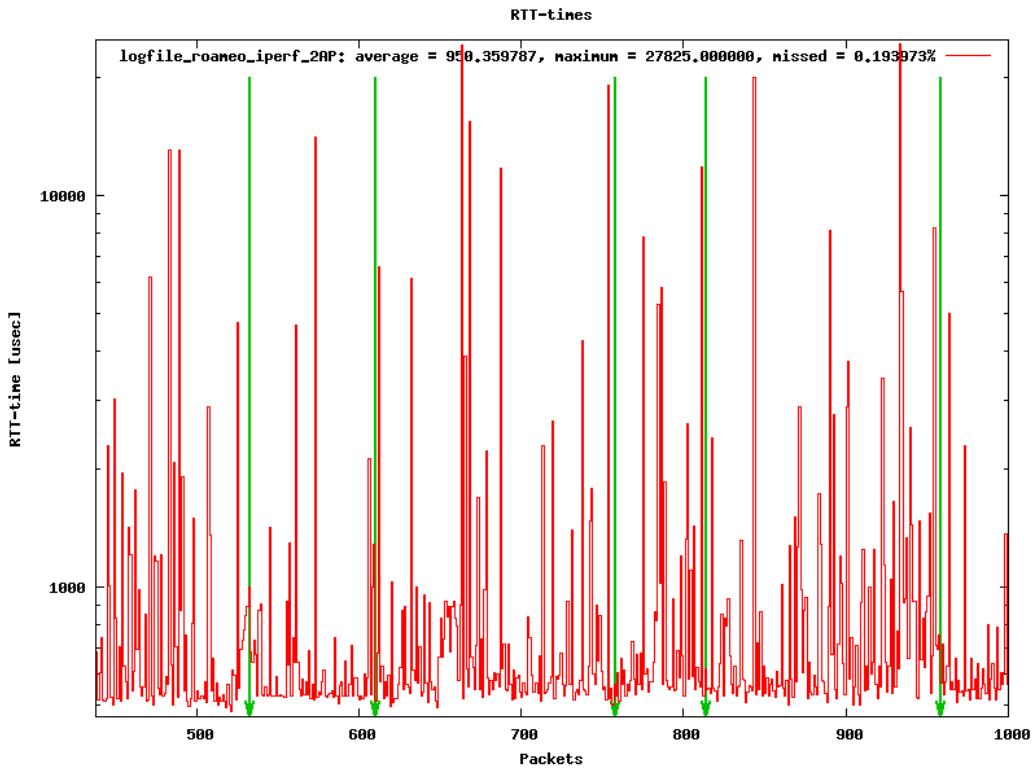


Abbildung 10.12: Vergrößerte Darstellung von 10.11 mit fünf Roamingvorgängen

6981 7.815700	10.0.0.179	10.0.0.107	UDP	Source port: 38548 Destination port: 4711
6982 7.815738		Cisco-Li_ed:e4:fb	(R)	IEEE 802 Acknowledgement
6983 7.816064	Giga-Byt_25:f1:99	Cisco-Li_ed:e4:fb		IEEE 802 Deauthentication, SN=1596, FN=0
6984 7.816093		Giga-Byt_25:f1:99	(R)	IEEE 802 Acknowledgement
6985 7.816145	Giga-Byt_25:f1:99	Cisco-Li_b7:53:f6		IEEE 802 Authentication, SN=1597, FN=0
6986 7.816185		Giga-Byt_25:f1:99	(R)	IEEE 802 Acknowledgement
6987 7.816244	Giga-Byt_25:f1:99	Cisco-Li_b7:53:f6		IEEE 802 Association Request, SN=1598, FN=1, SSID: "turbo"
6988 7.816283		Giga-Byt_25:f1:99	(R)	IEEE 802 Acknowledgement
6989 7.817039	Cisco-Li_b7:53:f6	Giga-Byt_25:f1:99		IEEE 802 Authentication, SN=1424, FN=0
6990 7.817342		Cisco-Li_b7:53:f6	(R)	IEEE 802 Acknowledgement
6991 7.818078	Cisco-Li_b7:53:f6	Giga-Byt_25:f1:99		IEEE 802 Association Response, SN=1425, FN=0
6992 7.818389		Cisco-Li_b7:53:f6	(R)	IEEE 802 Acknowledgement
6993 7.818548	10.0.0.107	10.0.0.179	UDP	Source port: 32792 Destination port: 4711
6994 7.818583		Giga-Byt_25:f1:99	(R)	IEEE 802 Acknowledgement

Abbildung 10.13: Ein Roamingvorgang inklusive letztem und erstem RTT-Paket

Kapitel 11

Implementierung von Treiberfunktionen

11.1 Treibererweiterung in der Theorie

Von Interesse war neben der bestehenden Lösung für das WLAN-Roaming das Erstellen einer vollkommen transparenten Bridge. Der WLAN-Rechner sollte in diesem Funktionsmodus von keinem anderen Rechner im Netzwerk in irgendeiner Weise adressiert werden müssen. Die Idee war eine Implementation als Gerät, das auf einer Seite Ethernetframes und auf der anderen WLAN-Frames bearbeitet, ohne dass dies von anderen Netzwerkkomponenten störend bemerkt werden kann oder eine Konfiguration notwendig wäre.

Der prinzipielle Aufbau einer solchen transparenten Bridge ist in Abbildung 11.1 dargestellt. Wichtig hierbei ist, dass die Daten, die im Bild links in den Accesspoint hineinkommen, exakt identisch auf der Ethernetseite der Bridge zu finden sein sollten und umgekehrt. Es darf also kein Routing einzustellen sein, die Bridge würde wirklich nur die Aufgabe übernehmen, die Ethernetframes in WLAN-Frames umzuwandeln. Nur so wäre eine echte Transparenz zu garantieren.

Um dies vollziehen zu können, ist es notwendig, Pakete zu versenden, die als Absender andere MAC-Adressen beinhalten, als die MAC-Adresse der WLAN-Karte selbst. Im Monitormodus stellt dies kein Problem dar, da schlicht der Inhalt des Paketes durch die in dieser Diplomarbeit vorgestellten Mechanismen ohne Weiteres möglich ist. Ein technisches Problem ergibt sich nur hinsichtlich der Acknowledges. Da die WLAN-Karte Pakete versendet, die laut MAC-Adresse nicht von ihr stammen, werden die empfangenden Sta-

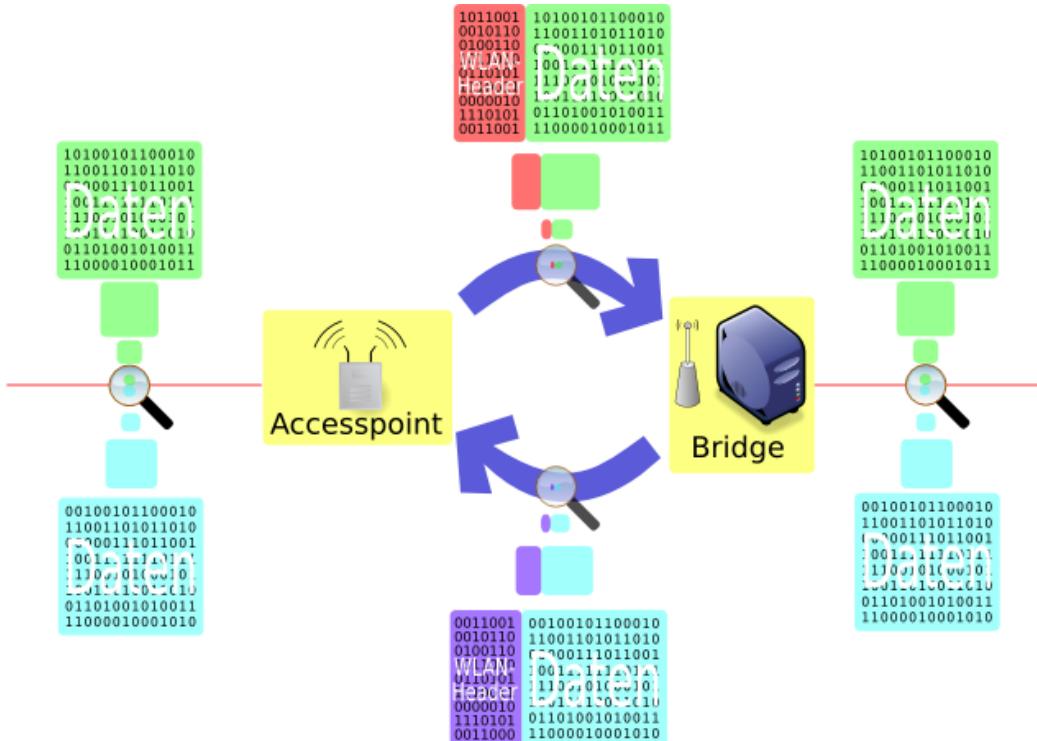


Abbildung 11.1: Schematischer Aufbau einer transparenten Bridge

tionen bzw. Accesspoints Acknowledges schicken, die auch an die sendende MAC-Adresse des empfangenen Paketes adressiert sind.

Diese Acknowledgements erreichen natürlich nie ihr Ziel, da die Pakete faktisch von einer anderen MAC-Adresse versendet wurden. Des Weiteren kann der Client, der dieses Paket verschickt hat, kein Acknowledge für eine fremde MAC-Adresse senden. Dies ist darin begründet, dass die HAL von MadWiFi die Acknowledges für die WLAN-Karte übernimmt, um, wie bereits in Abschnitt 6.3 erwähnt, schnell genug antworten zu können. Im Userspace generierte Acknowledges wären ebenfalls zu langsam. Ein funktionierender Datentransfer wäre also nicht möglich, da ohne passende Acknowledges entweder verpasste Pakete nicht erneut gesendet werden oder nicht gesendete Acknowledges Neuübertragungen bis zu elf Mal hervorrufen.

Die Lösung wäre es also, der HAL mitzuteilen, für welche MAC-Adressen zusätzlich zur eigenen sie Acknowledges schicken soll. Zu diesem Zweck wurde eine kleine Erweiterung des MadWiFi-Treibers entwickelt.

Die Idee dieser Erweiterung ist, die bestehende Fähigkeit des MadWiFi-Treibers für andere MAC-Adressen Acknowledges zu verschicken, zu erweitern. MadWiFi verfügt über die Möglichkeit, mehrere sogenannte Virtual Access Points (VAPs) zu erstellen und eine Karte so gleichzeitig in bis zu vier verschiedenen Modi zu nutzen. So kann zum Beispiel der erste erstellte VAP eine Station sein und zusätzlich noch für Beobachtungszwecke ein zweiter VAP als Monitor hinzugefügt werden. Diese Karten bekommen dann im ersten Byte der MAC-Adresse unterschiedliche Werte, etwa `00:11:22:33:44:55` für den ersten VAP und `06:11:22:33:44:55` für den zweiten. Dies wird im Treiber durch eine sogenannte BSSID-Maske geregelt, welche standardmäßig die Wertigkeit `F1:FF:FF:FF:FF:FF` hat, also bis auf das erste Byte lauter Einsen beinhaltet. Die hier nicht mit einer logischen 1 belegten Bits können in der MAC-Adresse verändert werden und dennoch werden auch für diese MAC-Adresse fortan Acknowledges verschickt. Würde man diese BSSID-Maske also beispielsweise auf ein `00:00:00:00:00:00` aufziehen, so müsste jede beliebige MAC-Adresse von der WLAN-Karte acknowledged werden. Dies funktioniert auch einwandfrei mit dem im folgenden beschriebenen Patch für den Treiber. Sobald diese Maske aber nur für einige bestimmte MAC-Adressen zutreffen soll, die Maske also keiner der beiden hier genannten Masken entspricht, wird gar kein Acknowledge mehr verschickt. Die Funktionalität der HAL ist hier also eingeschränkt und da die HAL bisher Binärkode von Atheros ist, konnte hier keine arbeitende Version entwickelt werden.

Seit kürzerem gibt es ein Projekt innerhalb von MadWiFi namens „OpenHAL“, welches sich damit beschäftigt, diesen Binärkode von Atheros mit einer quelloffenen HAL zu ersetzen. Mit Implementation einer stabilen Version dieser alternativen HAL könnte es eventuell in Zukunft möglich sein, Acknowledges auch für fremde MAC-Adressen zu verschicken, womit eine transparente Bridge realisierbar wäre.

Die folgende Beschreibung zeigt, wie man das Treiberinterface um einige Befehle erweitert.

11.2 Die Schritte zur Treibererweiterung

Die Erweiterung des Treiberinterfaces wurde durch einen `iwpriv(8)`[26]-Befehl genutzt. Um diesen überhaupt verfügbar zu machen, muss die Liste der privaten `ioctl(2)`-Kommandos erweitert werden. Dies muss im MadWiFi-Treiber in der Datei `net80211/ieee80211_ioctl.h` durch das Hinzufügen von

```
IEEE80211_PARAM_BSSIDMASK = 61, /*BSSID mask for ACK packets */
zu den verfügbaren Befehlen erfolgen. Wird nun ein solches Kommando aufgerufen, müssen die entsprechenden Funktionsaufrufe eingebunden werden,
```

KAPITEL 11. IMPLEMENTIERUNG VON TREIBERFUNKTIONEN

was innerhalb von `net80211/ieee80211_wireless.c` durch Einfügen von Listing 11.1 realisiert wurde.

Listing 11.1: Case-Abfragen in `net80211/ieee80211_wireless.c`

```
1 case IEEE80211_PARAM_BSSIDMASK:
2     ieee80211_set_bssidmask(dev, info, w, extra);
3     break;
4     (...)
```

```
6 case IEEE80211_PARAM_BSSIDMASK:
8     ieee80211_get_bssidmask(dev, info, w, extra);
9     break;
```

Diese beiden aufgerufenen Funktionen finden sich in derselben Datei wieder und stellen sich wie in Listing 11.2 zu sehen dar.

Listing 11.2: Die beiden BSSID-Masken verarbeitenden Funktionensaufrufe

```
1 static int
2     ieee80211_set_bssidmask(struct net_device *dev, struct iw_request_info *info,
3     struct sockaddr *addr, char *extra)
4     {
5         int *params = (int *) extra;
6         struct ieee80211vap *vap = dev->priv;
7         struct ieee80211com *ic = vap->iv_ic;
8         ic->ic_set_bssidmask_enabled(ic, params[1]);
9         return 0;
10    }
11
12    static int
13    ieee80211_get_bssidmask(struct net_device *dev, struct iw_request_info *info,
14    void *w, char *extra)
15    {
16        int *params = (int *) extra;
17        struct ieee80211vap *vap = dev->priv;
18        struct ieee80211com *ic = vap->iv_ic;
19        params[0] = ic->ic_get_bssidmask_enabled(ic);
20        return 0;
21    }
```

Der letzte Punkt innerhalb dieser Datei ist ein Hinzufügen von Listing 11.3 in die Funktion

```
static const struct iw_priv_args ieee80211_priv_args[].
```

Listing 11.3: Die neuen per `iwpriv(8)` verfügbaren Argumente

```
2 { IEEE80211_PARAM_BSSIDMASK, IW_PRIV_TYPE_INT | IW_PRIV_SIZE_FIXED |
3   1, 0, "bssidmask" },
4 { IEEE80211_PARAM_BSSIDMASK, 0, IW_PRIV_TYPE_INT | IW_PRIV_SIZE_FIXED |
5   1, "get_bssidmask" }
```

Ferner sind die Funktionsprototypen der Funktionen `ieee80211_set_bssidmask` und `ieee80211_get_bssidmask` in `net80211/ieee80211_var.h` einzufügen.

Finale Implementierung der Funktionalität ergibt sich dann durch Änderungen an der Datei `ath/if_ath.c`. Innerhalb der Funktion

```
int ath_attach(u_int16_t devid, struct net_device *dev)  
sind die Pointer
```

```
    ic->ic_set_bssidmask_enabled = ath_set_bssidmask_enabled und  
    ic->ic_get_bssidmask_enabled = ath_get_bssidmask_enabled.
```

Schließlich folgen die eigentlich ausgeführten Funktionen

```
static void ath_set_bssidmask_enabled(struct ieee80211com *ic, int enabled) und
static int ath_get_bssidmask_enabled(struct ieee80211com *ic) (Listing 11.4).
```

Listing 11.4: Die beiden BSSID-Masken verarbeitenden Funktionensaufrufe

```

2 static void
3 ath_set_bssidmask_enabled(struct ieee80211com *ic, int enabled)
4 {
5     struct ath_softc *sc = ic->ic_dev->priv;
6     struct ath_hal *ah = sc->sc_ah;
7     printk(KERN_DEBUG "ath_set_bssidmask_enabled(%p,%d)[sc->sc_hasbmask=%d]\n",
8           ic, enabled, sc->sc_hasbmask);
9     if(sc->sc_hasbmask) {
10         u_int8_t off[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
11         off[0] = enabled;
12         ath_hal_setbssidmask(ah, off);
13     }
14 }
15
16 static int
17 ath_get_bssidmask_enabled(struct ieee80211com *ic)
18 {
19     int enabled = 1;
20     struct ath_softc *sc = ic->ic_dev->priv;
21     struct ath_hal *ah = sc->sc_ah;
22     printk(KERN_DEBUG "ath_get_bssidmask_enabled(%p)", ic);
23     if(sc->sc_hasbmask) {
24         u_int8_t mask[6];
25         const u_int8_t off[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
26         ath_hal_getbssidmask(ah, mask);
27         printk(KERN_DEBUG ": %02X:%02X:%02X:%02X:%02X\n", mask[0], mask[1],
28               mask[2], mask[3], mask[4], mask[5]);
29         enabled = memcmp(mask, off, sizeof(off));
30     }
31     return enabled;
32 }
```

Kapitel 12

CAD-Zeichnungen von Präsentationsmustern

Für die Produktpräsentationen des EmbiCubes bei LKAB und Atlas Copco in Schweden wurden kleine Gehäuse für die genutzten Linksys-Accesspoints gekauft. Zu diesen wurden als Kopfplatten rechteckige Aluminiumbleche von 1 mm Stärke mit den Abmaßen 197 mm * 67 mm mitgeliefert. Um die Accesspoints in die neuen Gehäuse einbauen zu können, mussten Bohrungen in die Kopfplatten gefertigt werden.

Diese Bleche wurden nach den in Anhang C dargestellten Zeichnungen, welche mit SolidWorks erstellt wurden, im Laserzentrum der FH Münster gefertigt.

Die so entstandenen Accesspoints mit neuen Gehäusen sind in Abbildung 12.1 zu erkennen.



Abbildung 12.1: Entwickeltes Accesspointgehäuse für Präsentationen



Abbildung 12.2: Kontrolldiodenseite des neuen Accesspointgehäuses

Kapitel 13

Ausblick - was in Zukunft noch getan werden kann

Ein wesentlicher Punkt wäre die Erstellung einer „echten“ WLAN-Ethernet-Bridge. Ein solches Programm müsste die Pflege von sämtlichen Clients, die es hinter sich wüsste, übernehmen. Das Problem in der Realisation dieses Punktes in der vorliegenden Diplomarbeit bestand darin, dass es derzeit nicht möglich ist, Acknowledges für fremde Stationen zu versenden. Eine im Treiber implementierte Funktion, die dies mittels einer BSSID-Maske realisieren sollte, scheiterte daran, dass die HAL dies verbietet. Aufgrund der neuerlichen Arbeiten von MadWiFi an einer OpenHAL besteht eventuell in Zukunft die Möglichkeit, dies zu tun. Sobald diese Funktionalität gegeben ist, besteht auch das Potential, Frames an fremde Maschinen durchzurichten und die WLAN-Funktionalität mit versendeten Acknowledges für Accesspoints und andere Clients aufrechtzuerhalten.

Darüber hinaus wird das Programm „roameo“, das als Hauptergebnis dieser Diplomarbeit anzusehen ist, weiter Anwendung in den EmbiCubes, Routern und weiteren Rechnern der Zukunft von Embigence finden. Bereits durchgeführte erfolgreiche Installationen auf EmbiCubes beispielsweise für die Firmen DBT und Eickhoff stimmen hier besonders erwartungsvoll.

Um auch den Sicherheitsansprüchen der Kunden gerecht werden zu können, wird es unabdingbar werden, Verschlüsselung in die Software einzubauen.

Anhang A

Glossar

ANHANG A. GLOSSAR

- Accesspoint (AP)

Einheit, die Stationsfunktionalität liefert und Zugang zu den verteilten Diensten über das drahtlose Medium für assoziierte Stationen bietet.[17]

- AMD

Advanced Micro Devices; Chipsethersteller aus den USA.[3]

- ATA-5

Massenspeicherschnittstelle für Datentransfer mit bis zu 66,6 MByte/s

- Atheros

Halbleiter-Chipsethersteller mit Schwerpunkt auf Funknetzwerken. Zugrundeliegende Hardware für den MadWiFi-Treiber.[4]

- Basic Service Set (BSS)

Eine Menge von Stationen, die von einer einzigen Koordinationseinheit kontrolliert werden.[17]

- CF-Card

siehe Flash-Speicherkarte

- Crosscompiling

Kompilieren von Programmcode auf einer Maschinenarchitektur für eine andere Maschinenarchitektur.

- DDR SDRAM

Double Data Rate Synchronous Dynamic Random Access Memory; Arbeitsspeichertyp in PCs.

- Distribution System (DS)

Ein System, das zur Verbindung einer Reihe von Basic Service Sets (BSSs) mit Local Area Networks (LANs) benutzt wird, um ein Extended Service Set (ESS) herzustellen.[17]

- Distribution System Service (DSS)

Der Satz an Diensten, die das DS anbietet um Kommunikation zwischen Stationen zu ermöglichen, die nicht in direkter Kommunikation miteinander über eine einzelne Instanz des drahtlosen Mediums stehen.[17]

ANHANG A. GLOSSAR

- EIDE

Massenspeicherschnittstelle; notwendig für die Kommunikation zwischen Computer und Speichermedium.

- Embedded

Embedded steht wörtlich für „eingebettet“. In dieser Diplomarbeit wird es zumeist in dem Ausdruck „Embedded PC“ verwandt, was einen Computer beschreibt, der auf seine notwenigsten Bestandteile reduziert ist und ohne weiteres auch ohne einen Monitor oder eine Tastatur betrieben werden kann.

- Extended Service Set (ESS)

Eine Menge von verbundenen BSSs und LANs, die jeder assoziierten Station als ein einziges BSS erscheint.[17]

- Flash-Speicherkarte

Digitales Speichermedium kleiner Bauform, meist ohne mechanische Komponenten und deshalb gut geeignet für den Einsatz in erschütterungsreichen Anwendungsgebieten.

- Frame

Ein Datenframe ist ein Datagramm auf der OSI-Schicht 2. Es beinhaltet daher Senderadresse und Zieladresse und trägt in einem eigens dafür vorgesehenen Datenteil das eigentliche Übertragungsprotokoll mit dessen Daten.

- IEEE

Kurzform für Institute of Electrical and Electronics Engineers; weltweit größter Berufsverband der Elektrotechnik und Informatik. Von dieser Organisation verabschiedete Standards tragen diesen Namen in sich (Bsp.: IEEE802.11).

- Independent Basic Service Set (IBSS)

Ein BSS, das ein eigenständiges Netzwerk bildet und das keinen Zugang zu einem DS bietet.[17] Auch unter dem Begriff „AdHoc-Netzwerk“ bekannt.

- Intel

Kurzform für Integrated electronics; Chipsethersteller aus den USA.[8]

ANHANG A. GLOSSAR

- Medium Access Control (MAC)
Teilschicht des OSI-Modells, die zwischen logischer Kontrollebene und physikalischer Ebene liegt.
- Medium Access Control Service Data Unit(MSDU)
Information, die als eine Einheit zwischen MAC Service Access Points verschickt wird.[17]
- Maschinenarchitektur
Bauart eines Prozessors und damit einhergehend Art des Befehlsatzes, mit dem der Prozessor umgehen kann.
- Mini PCI
Verkleinerte Bauform von PCI. Einsatz vor allem in embedded Geräten und Laptops.
- MIPS
Kurzform für „Microprocessor without interlocked pipeline stages“. 32- oder 64-bit RISC-Maschinenarchitektur, die sich vor allem durch niedrige Leistungsaufnahme auszeichnet.[12]
- MIPSel
MIPS-Architektur mit der Bytereihenfolge Little Endian.
- Motherboard
Hauptplatine eines Computers, auf der sich aufsteckbare oder auflötbare Komponenten wie CPU oder Speicher befinden.
- Paket
Ein Paket ist eine Datenreihe auf der OSI-Schicht 3 und findet sich im Datenteil eines Frames. Die Senderadresse und Zieladresse des Frames sind hier bereits nicht mehr gültig. Ein Beispiel für ein Paket wäre ein IP-Paket.
- PC/104
Technischer Standard für Platinen; Abmaße 90 cm * 95 cm.
- PCI
PCI steht für Peripheral Component Interconnect und stellt einen Standard für die Verbindung von Komponenten und dem Chipsatz auf einem Motherboard dar.

ANHANG A. GLOSSAR

- Prism-Header

Ein WLAN-Header, der Informationen wie z.B. Signalstärke von Stationen beinhaltet.

- Roaming

Beim Wandern in Funknetzwerken vom Empfangsbereich eines Zugangspunktes in den des nächsten durchgeführte Übergabe zur Sicherstellung der Verbindung.

- Station (STA)

Jedes Gerät, das eine IEEE 802.11 konforme Medium Access Control und eine Schnittstelle zum drahtlosen Medium besitzt.[17]

- Station Service (SS)

Der Satz an Diensten, die den Transport von **MSDUs** zwischen Stationen innerhalb eines BSS unterstützen.[17]

- UDMA

Zugriffsprotokoll auf Massenspeichergeräte; es legt fest, wie Daten zwischen Massenspeicher und Mainboard übertragen werden.

- WDS

Abkürzung für Wireless Distribution System. Eine Möglichkeit, im WLAN Daten über mehrere Accesspoints per Funk an ein Ziel zu übermitteln, das sich nicht im gleichen BSS wie die sendende Station befindet.

- WEP

Kurzform für „Wired Equivalent Privacy“; ein Verbindungssicherungsverfahren für WLANs, welches als unsicher zu betrachten ist, da es inzwischen mit einfachsten Mitteln geknackt werden kann.

- Wi-Fi

Kurzform für Wireless Fidelity; Zusammenschluß von Unternehmen, um IEEE802.11-kompatible Geräte zu zertifizieren.

- Wireshark

OpenSource Netzwerkanalysesoftware [18], das den Quasistandard im Bereich der Datenanalyse in Netzwerken darstellt. Ursprünglich wurde dieses Projekt von Gerald Combs unter dem verbreiteten Namen

ANHANG A. GLOSSAR

„Ethereal“ gestartet. Sämtliche Netzwerkanalysen in dieser Diplomarbeit wurden mit dieser Software durchgeführt.

- **WPA**

Kurzform für Wi-Fi Protected Access; verbesserte Verschlüsselungsmethode gegenüber WEP, die auf dynamischen Schlüsseln basiert.

- **WRAP-Board**

Wireless Router Application Platform. Komplettplatine mit integriertem AMD Geode 266MHz-Prozessor und 128 MB SDRAM.

Anhang B

Quellcode

ANHANG B. QUELLCODE

Listing B.1: `roameo.c`

```

1 #include "roameo.h"
2 #include "wlan_mgmt.h"
3
4 /* globals used for data exchange between threads */
5 devices dev;
6 struct PARAMETER parameter;
7 unsigned int DEBUG, LOGGING, LOGGING_VERBOSE;
8 struct AP *first_accesspoint;
9 struct AP *chosen_accesspoint;
10 struct AP *accesspoint_pointer;
11 struct AP *temp_pointer;
12 int roam;
13
14 struct data vnic_data[DATA_BUFFER];
15 struct data wlan_data[DATA_BUFFER];
16 pthread_mutex_t mutex[DATA_BUFFER] = {PTHREAD_MUTEX_INITIALIZER};
17 sem_t vnic_empty, vnic_full, wlan_empty, wlan_full;
18
19 /* function below useful to inspect misc buffers */
20 // void hexdump(char *buffer, int len) {
21 //     int i;
22 //     for(i=0; i<len; i++) {
23 //         if(i%16 == 0) printf("\n%04x: ", i);
24 //         printf("%02x ", (unsigned char)(buffer[i]));
25 //     }
26 //     fflush(stdout);
27 //}
28
29 //this function leaves the global variable "accesspoint_pointer"
30 //pointing on the accesspoint with the given mac at call
31 int evaluate_beacon(u_char *mac, unsigned int signal) {
32     int i;
33     int interim;
34     struct timezone here;
35
36     interim = 0;
37
38     if ( first.accesspoint != NULL ) {
39         accesspoint_pointer = first.accesspoint;
40         //This is not the first accesspoint ever seen
41         while( accesspoint_pointer != NULL ) {
42             //check if current object is already a part of the AP list
43             if( memcmp( mac, accesspoint_pointer->mac, MAC_ADDRESS_SIZE ) == 0 ) break;
44             //remember the position of the last valid pointer
45             temp_pointer = accesspoint_pointer;
46             //shift the pointer to the next memory area
47             accesspoint_pointer = accesspoint_pointer->next;
48         }
49
50         //loop for introducing new APs into the AP-database
51         if ( accesspoint_pointer == NULL ) {
52             if ( (accesspoint_pointer = (struct AP *)malloc(sizeof(struct AP))) == NULL ) {
53                 perror("No memory available for further accesspoints");
54                 exit(-1);
55             }
56             //link the new AP as the next one after the last AP
57             temp_pointer->next = accesspoint_pointer;
58             if ( DEBUG ) printf("Copying Data to new AP\n");
59
60             if ( memset( accesspoint_pointer, 0, sizeof(struct AP) ) == NULL ) {
61                 perror("Error setting memory of new accesspoint to \\"0\\"");
62                 return -1;
63             }
64             if ( memcpy( accesspoint_pointer->mac, mac, MAC_ADDRESS_SIZE ) == NULL ) {
65                 perror("Error copying MAC for new accesspoint");
66                 return -1;
67             }
68         }
69         //usual handling of every AP
70         accesspoint_pointer->signal_database[accesspoint_pointer->counter] = signal;
71         accesspoint_pointer->counter++;
72
73         if ( accesspoint_pointer->counter >= parameter.number_of_beacons ) {
74             accesspoint_pointer->counter = 0;
75             accesspoint_pointer->considerable_results_for_changing_AP = 1;
76         }
77
78         for( i = 0; i < parameter.number_of_beacons; i++ )
79             interim += accesspoint_pointer->signal_database[i];
80         accesspoint_pointer->value = interim / parameter.number_of_beacons;
81         if ( accesspoint_pointer->value > (chosen.accesspoint->value + parameter.threshold) ) {
82             chosen.accesspoint = accesspoint_pointer;
83             roam = TRUE;
84         }
85     }
86     else {
87         //First accesspoint is to be introduced to the database
88         if ( (first.accesspoint = (struct AP *)malloc(sizeof(struct AP))) == NULL ) {
89             perror("No memory available for first accesspoint");
90             return -1;
91         }
92         if ( DEBUG ) printf("Copying Data to first AP\n");
93         if ( memset( first.accesspoint, 0, sizeof(struct AP) ) == NULL ) {
94             perror("Error setting memory of first accesspoint to \\"0\\"");
95             return -1;
96         }
97         if ( memcpy( first.accesspoint->mac, mac, MAC_ADDRESS_SIZE ) == NULL ) {
98             perror("Error copying MAC for first accesspoint");
99             return -1;
100         }
101
102         first.accesspoint->signal_database[first.accesspoint->counter] = signal;
103         first.accesspoint->counter++;
104
105         if ( first.accesspoint->counter >= parameter.number_of_beacons ) {
106             first.accesspoint->counter = 0;
107             first.accesspoint->considerable_results_for_changing_AP = 1;
108         }
109
110         for( i = 0; i < parameter.number_of_beacons; i++ )
111             interim += first.accesspoint->signal_database[i];
112         first.accesspoint->value = interim / parameter.number_of_beacons;
113         first.accesspoint->next = NULL;
114     }
115 }
```

ANHANG B. QUELLCODE

```

116     chosen_accesspoint = first_accesspoint;
117     accesspoint_pointer = first_accesspoint;
118 }
119 gettimeofday( &accesspoint_pointer->timestamp, &here );
120 return( 0 );
121 }
122 int check_timestamps() {
123     struct timeval now;
124     struct timezone here;
125     gettimeofday( &now, &here );
126     accesspoint_pointer = first_accesspoint;
127     while( accesspoint_pointer != NULL ) {
128         /*check if accesspoints timestamp is overlapping seconds-border
129          if ( accesspoint_pointer->timestamp.tv_sec < now.tv_sec ) {
130              //FAR too old -> immediate deletion and quit of this function
131              if ( accesspoint_pointer->timestamp.tv_sec < now.tv_sec-1 ) {
132                  delete_accesspoint( accesspoint_pointer );
133                  if ( DEBUG ) printf("Deleting accesspoint with timestamp FAR too old (>1sec)\n");
134                  return -1;
135              }
136          //only overlapping seconds; normal check if timestamp is too old
137          if ( 1000000 - accesspoint_pointer->timestamp.tv_usec + now.tv_usec
138              > parameter.beacon_time_limit ) {
139              if ( DEBUG ) printf("Deleting accesspoint with old timestamp" \
140                  " (%d)\n",parameter.beacon_time_limit);
141              delete_accesspoint( accesspoint_pointer );
142              return -1;
143          }
144      }
145      //check if timestamp is older than a set limit
146      if ( accesspoint_pointer->timestamp.tv_usec < now.tv_usec - parameter.beacon_time_limit ) {
147          if ( DEBUG ) printf("Deleting accesspoint with old timestamp" \
148              " (%d)\n",parameter.beacon_time_limit);
149          delete_accesspoint( accesspoint_pointer );
150          return -1;
151      }
152      accesspoint_pointer = accesspoint_pointer->next;
153 }
154 accesspoint_pointer = first_accesspoint;
155 return 0;
156 }
157
158 int delete_accesspoint( struct AP *liquidation ) {
159     struct AP *temp_pointer, *temp_chosen;
160     int best = 0;
161
162     temp_chosen = chosen_accesspoint;
163
164     if ( DEBUG ) printf("Deleting accesspoint from list!\n");
165     //check for first element in list
166     if ( memcmp( liquidation->mac, first_accesspoint->mac, MAC_ADDRESS_SIZE ) == 0 ) {
167         //if deleted accesspoint is currently associated accesspoint and
168         //first accesspoint in list, disassociate and chose new accesspoint
169         if ( chosen_accesspoint == first_accesspoint ) {
170             //check signature of all accesspoints except the first in the list,
171             //which is about to be deleted anyway
172             accesspoint_pointer = first_accesspoint->next;
173             //go through the rest of the list
174             while( accesspoint_pointer != NULL ) {
175                 //find the best element in the list
176                 if( accesspoint_pointer->value > best ) {
177                     best = accesspoint_pointer->value;
178                     chosen_accesspoint = accesspoint_pointer;
179                 }
180                 accesspoint_pointer = accesspoint_pointer->next;
181             }
182         }
183         //set the temporary pointer to the second element in the list
184         accesspoint_pointer = first_accesspoint->next;
185         //give memory of old first accesspoint free
186         free( first_accesspoint );
187         //make the second member of the list become the first one
188         first_accesspoint = accesspoint_pointer;
189     } else {
190         //start search at beginning of list
191         accesspoint_pointer = first_accesspoint;
192         //it is not the first element in list, check for the element to be found
193         while( accesspoint_pointer != NULL ) {
194             temp_pointer = accesspoint_pointer->next;
195             //if MACs match, delete accesspoint from list
196             if ( memcmp( temp_pointer->mac, liquidation->mac, MAC_ADDRESS_SIZE ) == 0 ) {
197                 accesspoint_pointer->next = temp_pointer->next;
198                 free( temp_pointer );
199                 break;
200             }
201             //if MACs do not match, go to next accesspoint
202         } else accesspoint_pointer = accesspoint_pointer->next;
203     }
204
205     //if no single accesspoint is in the list anymore, set chosen_accesspoint
206     //to NULL and leave the function
207     if ( first_accesspoint == NULL ) {
208         chosen_accesspoint = NULL;
209     }
210     return 0;
211 }
212
213 //if deleted accesspoint is currently associated accesspoint,
214 //disassociate and chose new accesspoint
215 if ( liquidation == temp_chosen ) {
216     //start from beginning of list
217     accesspoint_pointer = first_accesspoint;
218     //go through the whole accesspoint list
219     while( accesspoint_pointer != NULL ) {
220         if( accesspoint_pointer->value > best ) {
221             best = accesspoint_pointer->value;
222             chosen_accesspoint = accesspoint_pointer;
223         }
224         accesspoint_pointer = accesspoint_pointer->next;
225     }
226     //copy new accesspoint-MAC to global parameter
227     memcpy( &parameter.dst_mac, &chosen_accesspoint->mac, MAC_ADDRESS_SIZE );
228     //associate with new accesspoint
229     assoc( parameter.dst_mac, parameter.src_mac, parameter.ssid );
230 }
231
232 accesspoint_pointer = first_accesspoint;

```

ANHANG B. QUELLCODE

```

234     return 0;
235 }
236 void *read_from_vnic(void *arg) {
237     int array_counter, size;
238     char data[MAXBUFFER];
239
240     array_counter = 0;
241     while(1) {
242         /*reset buffer
243         memset( &data, 0, MAXBUFFER );
244         /*read on vnic
245         size = read( dev.fd_vnic, data, MAXBUFFER );
246         /*set empty down by 1
247         sem_wait( &vnic_empty );
248         /*lock the mutex
249         pthread_mutex_lock( &mutex[array_counter] );
250         /*reset vnic
251         memset( &vnic_data[array_counter], 0, sizeof(struct data) );
252         /*copy the data read to vnic_data-array
253         vnic_data[array_counter].size = size;
254         memcpy( vnic_data[array_counter].data, data, vnic_data[array_counter].size );
255         /*unlock the mutex
256         pthread_mutex_unlock( &mutex[array_counter] );
257         /*set vnic_full up by one
258         sem_post( &vnic_full );
259         /*increase array_counter by one
260         array_counter = ( array_counter + 1 ) % DATA_BUFFER;
261     }
262     return NULL;
263 }
264 void *write_on_wlan(void *arg) {
265     int array_counter, size;
266     char out[MAXBUFFER], data[MAXBUFFER];
267     struct ether_header *eth;
268     ieee802_11_hdr *WiFi_header;
269     struct fddi_snap_hdr *snap;
270
271     array_counter = 0;
272     while(1) {
273         /*reset buffers
274         memset(out, 0, MAXBUFFER);
275         memset(data, 0, MAXBUFFER);
276         /*wait for vnic_full to contain an object
277         sem_wait( &vnic_full );
278         /*lock the mutex
279         pthread_mutex_lock( &mutex[array_counter] );
280         /*copy data of global variable to local variables
281         memcpy( data, vnic_data[array_counter].data, vnic_data[array_counter].size );
282         size = vnic_data[array_counter].size;
283         /*reset global memory
284         memset( &vnic_data[array_counter], 0, sizeof(struct data) );
285         /*unlock the mutex
286         pthread_mutex_unlock( &mutex[array_counter] );
287         /*set vnic_empty up by one
288         sem_post( &vnic_empty );
289
290         if ( size > sizeof(struct ether_header) ) {
291             eth = (struct ether_header *)data;
292             /* check if it is an IP or ARP Ethernet frame */
293             if ( ntohs(eth->ether_type) == ETHERTYPE_IP || ntohs(eth->ether_type) == ETHERTYPE_ARP ) {
294                 /* prepare WLAN header */
295                 WiFi_header = (ieee802_11_hdr *)out;
296                 WiFi_header->subtype = IEEE80211_TYPE_DATA;
297                 WiFi_header->flags = IEEE80211_TO_DS;
298                 memcpy( WiFi_header->duration, "\xD4\x00", 2 );
299
300                 /* copy addresses */
301                 memcpy( WiFi_header->addr1, parameter.dst_mac, ETH_ALEN );
302                 memcpy( WiFi_header->addr2, eth->ether_shost, ETH_ALEN );
303                 memcpy( WiFi_header->addr3, eth->ether_dhost, ETH_ALEN );
304
305                 /* prepare SNAP header, LLC, defaults were retrieved from header include file */
306                 snap = (struct fddi_snap_hdr *)(((char *)WiFi_header) + sizeof(ieee802_11_hdr));
307                 snap->dsap = 0xaa;
308                 snap->ssap = 0xaa;
309                 snap->ctrl = 0x03;
310                 snap->ethertype = eth->ether_type;
311
312                 /* copy payload */
313                 size = size - sizeof(struct ether_header);
314                 memcpy( ((char *)WiFi_header)+32, ((char *)eth) + sizeof(struct ether_header),
315                         size );
316                 write( dev.fd_wlan, out, size+sizeof(ieee802_11_hdr)+sizeof(struct fddi_snap_hdr));
317             }
318         }
319         array_counter = ( array_counter + 1 ) % DATA_BUFFER;
320     }
321     return NULL;
322 }
323 void *read_from_wlan(void *arg) {
324     int array_counter, size;
325     char data[MAXBUFFER];
326
327     array_counter = 0;
328     while(1) {
329         /*reset data-buffer
330         memset( &data, 0, MAXBUFFER );
331         /*read on wlan
332         size = read( dev.fd_wlan, data, MAXBUFFER );
333         /*set wlan_empty down by 1
334         sem_wait( &wlan_empty );
335         /*lock the mutex
336         pthread_mutex_lock( &mutex[array_counter] );
337         /*reset buffer
338         memset( &wlan_data[array_counter], 0, sizeof(struct data) );
339         /*copy the data read to wlan_data-array
340         wlan_data[array_counter].size = size;
341         memcpy( wlan_data[array_counter].data, data, wlan_data[array_counter].size );
342         /*unlock the mutex
343         pthread_mutex_unlock( &mutex[array_counter] );
344         /*set wlan_full up by 1

```

ANHANG B. QUELLCODE

```

350     sem_post( &wlan_full );
351     /*increase array_counter by 1
352     array_counter = ( array_counter + 1 ) % DATA_BUFFER;
353   }
354   return NULL;
355 }
356 void *write_on_vnic(void *arg) {
357   int array_counter, size;
358   char out[MAXBUFFER], data[MAXBUFFER];
359   struct ether_header *eth;
360   ieee802_11_hdr *WiFi_header;
361   struct fddi_snap_hdr *snap;
362   prism_hdr *Prism_header;
363   ieee802_11_beacon_mgt_frame *beacon_frame;
364   int SubType, to_ds, from_ds;
365   int received_ssid_len = 0;
366   char received_ssid[SSID_MAX_SIZE];
367   char dotted_mac[20];
368   struct logging log;
369   array_counter = 0;
370   memset( &log, 0, sizeof( struct logging ) );
371   while(1) {
372     /*reset buffers
373     memset(out, 0, MAXBUFFER);
374     memset(data, 0, MAXBUFFER);
375     /*wait for wlan_full to contain an object
376     sem_wait( &wlan_full );
377     /*lock the mutex
378     pthread_mutex_lock( &mutex[array_counter] );
379     /*copy data of global variable to local variables
380     memcpy( data, wlan_data[array_counter].data, wlan_data[array_counter].size );
381     size = wlan_data[array_counter].size;
382     /*reset global memory
383     memset( &wlan_data[array_counter], 0, sizeof(struct data) );
384     /*unlock the mutex
385     pthread_mutex_unlock( &mutex[array_counter] );
386     /*set vnic_empty up by one
387     sem_post( &wlan_empty );
388
389     Prism_header = (prism_hdr *)data;
390     WiFi_header = (ieee802_11_hdr *)(data + sizeof(prism_hdr));
391     beacon_frame = (ieee802_11_beacon_mgt_frame *) (data+sizeof(prism_hdr)+sizeof(ieee802_11_hdr));
392     /*perform a bitshift of the upper nibble to the lower and vice versa
393     SubType = ((WiFi_header->subtype & 0xF0) >> 4) + ((WiFi_header->subtype & 0xC) << 2);
394
395     /*check for beacons to be evaluated according to signal strength */
396     if ( SubType == mgt_beacon ) {
397       if( beacon_frame->tag_number == ESSID_TAG_NUMBER )
398         received_ssid_len = (int)beacon_frame->tag_length;
399       /*copy the ssid to the local variable received_ssid
400       memcpy( received_ssid, beacon_frame->tag_interpretation, received_ssid_len );
401       received_ssid[received_ssid_len] = '\0';
402
403       if ( strcmp( received_ssid, parameter.ssid, received_ssid_len ) == 0 ) {
404         if( evaluate_beacon( WiFi_header->addr2, Prism_header->signal.data ) == -1) {
405           printf("Evaluating beacon FAILED. EXITING!\n");
406           exit(-1);
407         }
408         if ( DEBUG ) {
409           if ( memcmp( accesspoint_pointer->mac, chosen_accesspoint->mac,
410             MAC_ADDRESS_SIZE ) == 0 ) printf("CHosen ");
411           ether_ntop( accesspoint_pointer->mac, dotted_mac );
412           printf("MAC: %s ", dotted_mac);
413           printf("Value: %d\n", accesspoint_pointer->value);
414         }
415
416       //check if this accesspoint is the chosen one
417       //and check if we have already enough results to consider switching
418       //and check if accesspoint is already associated
419       if ( ( roam == TRUE ) &&
420           ( accesspoint_pointer->considerable_results_for_changing_AP == 1 ) &&
421           ( memcmp( accesspoint_pointer->mac, parameter.dst_mac
422             MAC_ADDRESS_SIZE ) != 0 ) ) {
423
424         //write logging information to logging software
425         if( LOGGING ) {
426           log.flags = ROAMING;
427           ether_ntop( accesspoint_pointer->mac, log.new_ap );
428           ether_ntop( parameter.dst_mac, log.old_ap );
429           write( parameter.logging_socket, &log, sizeof( struct logging ) );
430         }
431         chosen_accesspoint->considerable_results_for_changing_AP = 0;
432         chosen_accesspoint = accesspoint_pointer;
433       }
434     //      assoc( accesspoint_pointer->mac, parameter.src_mac, parameter.ssid );
435     disassoc( parameter.dst_mac, parameter.src_mac, parameter.ssid );
436     //set global values according to the chosen AP
437     memcpy( &parameter.dst_mac, accesspoint_pointer->mac, MAC_ADDRESS_SIZE );
438     ether_ntop( parameter.dst_mac, parameter.dst_mac_dotted );
439     //associate with the selected AP
440     assoc( parameter.dst_mac, parameter.src_mac, parameter.ssid );
441     roam = FALSE;
442   } else {
443     //check if there is no accesspoint yet chosen
444     //if so, ignore considerable results for any accesspoint, but
445     //instead associate at once!
446     if( chosen_accesspoint == NULL ) {
447       //      assoc( accesspoint_pointer->mac, parameter.src_mac,
448       parameter.ssid );
449       //disassociate with old AP; just to be sure :-)
450       //maybe it can still see us...
451       disassoc( parameter.dst_mac, parameter.src_mac, parameter.ssid );
452       //set global values according this one and only AP
453       memcpy( &parameter.dst_mac, accesspoint_pointer->mac,
454         MAC_ADDRESS_SIZE );
455       ether_ntop( parameter.dst_mac, parameter.dst_mac_dotted );
456       //associate with this AP
457       accesspoint_pointer = first_accesspoint;
458       assoc( parameter.dst_mac, parameter.src_mac, parameter.ssid );
459     }
460   }
461   accesspoint_pointer = first_accesspoint;
462   if( check_timestamps() == -1 && DEBUG ) printf("Accesspoint DELETED from list");
463   if( first_accesspoint == NULL && DEBUG ) printf("No accesspoint accessible, waiting...\n");
464 }
465

```

ANHANG B. QUELLCODE

```

468     /* filter out anything but data */
469     if ((SubType & 0xFO) == 0x20 && (SubType & 0xF) < 4) {
470         /* filter out other AP */
471         to_ds = WiFi_header->flags & IEEE80211_TO_DS;
472         from_ds = WiFi_header->flags & IEEE80211_FROM_DS;
473
474         /* must be from AP and sent to client */
475         if ( from_ds && !to_ds && !memcmp(WiFi_header->addr2, parameter.dst_mac, ETH_ALEN) ) {
476             snap = (struct fddi_snap_hdr *)(((char *)WiFi_header)) + sizeof(ieee802_11_hdr);
477
478             /* check if IP/ARP packet */
479             if ( ntohs(snap->etherstype) == ETHERTYPE_IP || ntohs(snap->etherstype)
480 == ETHERTYPE_ARP ) {
481
482                 /* prepare ethernet_header */
483                 eth = (struct ether_header *)out;
484
485                 memcpy(eth->ether_dhost, WiFi_header->addr1, ETH_ALEN);
486                 memcpy(eth->ether_shost, WiFi_header->addr3, ETH_ALEN);
487                 eth->ether_type = snap->etherstype;
488
489                 /* copy IP payload */
490                 size = size - sizeof(prism_hdr) - sizeof(ieee802_11_hdr)
491                 - sizeof(struct fddi_snap_hdr);
492                 memcpy(out+sizeof(struct ether_header),
493                     ((char *)snap)+sizeof(struct fddi_snap_hdr), size);
494
495                 /* send it out of the network card, odd enough: when sending the tun/tap
496                 header is not prepended*/
497                 write(dev.fd_vnic, out, sizeof(struct ether_header) + size);
498             }
499         }
500         array_counter = (array_counter + 1) % DATA_BUFFER;
501     }
502     return NULL;
503 }
504
505 /* management function, that controls all of the called threads */
506 int manage_threads( void ) {
507     pthread_t vnic_read_thread, vnic_write_thread, wlan_read_thread, wlan_write_thread;
508
509     //initialize semaphores
510     sem_init( &vnic_empty, 0, DATA_BUFFER );
511     sem_init( &vnic_full, 0, 0 );
512     sem_init( &wlan_empty, 0, DATA_BUFFER );
513     sem_init( &wlan_full, 0, 0 );
514
515     //start threads
516     pthread_create( &vnic_read_thread, NULL, &read_from_vnic, NULL );
517     pthread_create( &wlan_write_thread, NULL, &write_on_wlan, NULL );
518     pthread_create( &wlan_read_thread, NULL, &read_from_wlan, NULL );
519     pthread_create( &vnic_write_thread, NULL, &write_on_vnic, NULL );
520
521     //wait for threads to stop
522     pthread_join( vnic_read_thread, NULL );
523     pthread_join( wlan_write_thread, NULL );
524     pthread_join( wlan_read_thread, NULL );
525     pthread_join( vnic_write_thread, NULL );
526
527     return 0;
528 }
529
530 int ether_ntop(const u_char address[MAC_ADDRESS_SIZE], char * buf) {
531     return( sprintf(buf, "%02X:%02X:%02X:%02X:%02X",
532                     address[0], address[1], address[2], address[3], address[5]) );
533 }
534
535 int mac_to_binary(const char * orig, unsigned char * mac, int macmax) {
536     const char * p = orig;
537     int maclen = 0;
538
539     /* Loop on all bytes of the string */
540     while(*p != '\0') {
541
542         int temp;
543         int templ;
544         int count;
545
546         /* Extract one byte as two chars */
547         count = sscanf(p, "%1X%1X", &temp, &templ);
548         if(count != 2) {
549             /* Error -> non-hex chars */
550             break;
551         }
552         /* Output two chars as one byte */
553         templ |= temp << 4;
554         mac[maclen++] = (unsigned char) (templ & 0xFF);
555
556         /* Check end of string */
557         p += 2;
558         if(*p == '\0') {
559             return(maclen); /* Normal exit */
560         }
561
562         /* Check overflow */
563         if(maclen >= macmax)
564             if( DEBUG ) {
565                 fprintf(stderr, "mac_to_binary(%s): trailing junk!\n", orig);
566             }
567             errno = E2BIG;
568             return(0); /* Error -> overflow */
569
570         /* Check separator */
571         if(*p != ':')
572             break;
573         p++;
574     }
575
576     /* Error... */
577     if( DEBUG ) {
578         fprintf(stderr, "mac_to_binary(%s): invalid ether address!\n", orig);
579     }
580     errno = EINVAL;
581 }
```

ANHANG B. QUELLCODE

```

584     return(0);
585 }
586
588 int tap_open(char *dev) {
589     struct ifreq request;
590     int fd;
591
592     if( (fd = open("/dev/net/tun", O_RDWR)) == -1 ) {
593         perror("/dev/net/tun not present");
594         return -1;
595     }
596
597     memset( &request, 0, sizeof(request) );
598     request.ifr_flags = IFF_TAP | IFF_NO_PI;
599     if( *dev )
600         strncpy(request.ifr_name, dev, IFNAMSIZ);
601
602     if( ioctl( fd, TUNSETIFF, (void *) &request ) == -1 ) {
603         close( fd );
604         perror("TUNSETIFF failed");
605         return -1;
606     }
607
608     strcpy( dev, request.ifr_name );
609     return fd;
610 }
611
612 /* interface initialization routine */
613 int openraw( char *iface, int fd) {
614     struct ifreq request;
615     struct sockaddr_ll socket_address_ll;
616
617     /* find the interface index */
618     memset( &request, 0, sizeof( request ) );
619     strncpy( request.ifr_name, iface, sizeof( request.ifr_name ) - 1 );
620
621     if( ioctl( fd, SIOCGIFINDEX, &request ) == -1 ) {
622         perror("ioctl(SIOCGIFINDEX) failed");
623         return -1;
624     }
625
626     /* bind the raw socket to the interface */
627     memset( &socket_address_ll, 0, sizeof( socket_address_ll ) );
628     socket_address_ll.sll_family      = AF_PACKET;
629     //put the interface number into the sockaddr
630     socket_address_ll.sll_ifindex   = request.ifr_ifindex;
631     //get ALL packets ( look in packet(7) )
632     socket_address_ll.sll_protocol  = htons( ETH_P_ALL );
633
634     if( bind( fd, (struct sockaddr *) &socket_address_ll, sizeof( socket_address_ll ) ) == -1 ) {
635         perror( "bind(ETH_P_ALL) failed" );
636         return -1;
637     }
638
639     return 0;
640 }
641
642 //do an initial scan for available APs and associate the best one
643 int initial_scan( void ) {
644     char in[MAXBUFFER];
645     char received_ssid[SSID_MAX_SIZE];
646     int SubType, received_ssid_len = 0;
647     ieee802_11_hdr *WiFi_header;
648     prism_hdr *Prism_header;
649     ieee802_11_beacon_mgt_frame *beacon_frame;
650     fd_set file_descriptor_set;
651     time_t start, meantime;
652     int elapsed = 0;
653     int best = 0;
654
655     FD_ZERO(&file_descriptor_set);
656     FD_SET(dev.fd_wlan, &file_descriptor_set);
657
658     time(&start);           //get start time of scanning
659     do {                   //start of 1 second loop
660         /* read from wlan card */
661         if( FD_ISSET( dev.fd_wlan, &file_descriptor_set ) ) {
662             memset( in, 0, sizeof(in) );
663             read( dev.fd_wlan, in, sizeof(in) );
664             time( &meantime ); //get current time while scanning
665             elapsed = meantime - start; //calculate elapsed time since start
666
667             //Prism header is the first data of an incoming packet.
668             //It is present, since the driver seems to be appending it...
669             Prism_header = (prism_hdr *)in;
670             //After the prism header, we find the WiFi-header
671             WiFi_header = (ieee802_11_hdr *)(in + sizeof(prism_hdr));
672             //if we have a beacon-frame, then this fits fine:
673             beacon_frame = (ieee802_11_beacon_mgt_frame *)(in + sizeof(prism_hdr)+sizeof(ieee802_11_hdr));
674             //filter out only beacons from WLAN-trffic
675             //We need to shift the high nibble to become the low nibble and vice versa
676             SubType = ((WiFi_header->subType & 0xF0) >> 4) + (((WiFi_header->subType & 0xC) << 2));
677
678             //get the length of the transmitted ssid
679             if( beacon_frame->tag_number == ESSID_TAG_NUMBER )
680                 received_ssid_len = (int)beacon_frame->tag_length;
681             //copy the ssid to the local variable received_ssid
682             memcpy( received_ssid, beacon_frame->tag_interpretation, received_ssid_len );
683             received_ssid[received_ssid_len] = '\0';
684
685             if ( DEBUG ) {
686                 printf("Beacon seen: %s\n", received_ssid);
687             }
688
689             if ( strncmp( received_ssid, parameter.ssid, received_ssid_len ) == 0 ) {
690                 if( evaluate_beacon( WiFi_header->addr2,Prism_header->signal.data ) == -1 ) {
691                     printf("Evaluating beacon FAILED. EXITING!\n");
692                     return -1;
693                 }
694                 //check if AP has best signal so far
695                 if( chosen_accesspoint->value > best ) {
696                     chosen_accesspoint = accesspoint_pointer;
697                     best = accesspoint_pointer->value;
698                 }
699             }
700         }
701     }
702 }
```

ANHANG B. QUELLCODE

```

    }
    accesspoint_pointer = first_accesspoint;
    //do scanning at least for 1 second or no AP found yet
    } while ( elapsed < 1 || first_accesspoint == NULL );
    //set global values according the chosen AP
    memcpy( &parameter.dst_mac, &chosen_accesspoint->mac, MAC_ADDRESS_SIZE );
    ether_ntop( parameter.dst_mac, chosen_accesspoint->mac, parameter.dst_mac_dotted );
    //associate with the selected AP
    assoc( parameter.dst_mac, parameter.src_mac, parameter.ssid );
    return 0;
}
int open_sockets( void ) {
/* create an UDP-socket for miscellaneous and ioctl(2) requests */
716 if( ( dev.fd_udp = socket( PF_INET, SOCK_DGRAM, 0 ) ) == -1 ) {
    perror("socket(PF_INET) for fd_udp failed");
    return -1;
}
720 /* create the RAW socket */
722 if( ( dev.fd_wlan = socket( PF_PACKET, SOCK_RAW, htons( ETH_P_ALL ) ) ) == -1 ) {
    perror("socket(PF_PACKET) for fd_wlan failed");
    return -1;
}
726 /* open the source/target for WLAN frames */
728 if( openraw( parameter.monitor_device, dev.fd_wlan ) == -1 ) {
    perror("Opening fd_wlan for monitor device failed");
    return -1;
}
732 /* create a virtual network card */
734 if( ( dev.fd_vnic = tap_open( parameter.virtual_device ) ) == -1 ) {
    perror("tap_open() failed");
    return -1;
}
738 return 0;
}
740 int configure_devices( void ) {
742 struct ifreq virtual_ifreq, monitor_ifreq;
744 struct sockaddr_in *address_pointer;
744 char buf[500];
746 //define an interface-request
747 memset( &virtual_ifreq, 0, sizeof(struct ifreq) );
748 memset( &monitor_ifreq, 0, sizeof(struct ifreq) );
749 //set the interface-name inside the if-request; necessary for later operation
750 //for the virtual device...
751 strcpy( virtual_ifreq.ifr_name, parameter.virtual_device, IFNAMSIZ );
752 //...and for the monitor device
753 strcpy( monitor_ifreq.ifr_ifrn.ifrn_name, parameter.monitor_device, IFNAMSIZ );
754
755 //set flag for virtual interface DOWN
756 virtual_ifreq.ifr_flags = IFF_UP;
757 //put virtual interface DOWN
758 if ( ioctl( dev.fd_udp, SIOCSIFFLAGS, (caddr_t)&virtual_ifreq ) == -1 ) {
    perror("Setting virtual device down failed");
    return -1;
}
762 //get hardware address of monitor-device in order to compare it with
763 //a possibly entered source-mac address
764 if( ioctl( dev.fd_udp, SIOCGIFHWADDR, &monitor_ifreq ) == -1 ) {
    perror("Error getting hardware address on monitor-device");
    return -1;
}
768
770 //if the desired MAC in the config_file is different from the MAC of the monitor device,
771 //try to change the MAC of the monitor device
772 if ( memcmp( &monitor_ifreq.ifru.ifru_hwaddr.sa_data, &parameter.src_mac, MAC_ADDRESS_SIZE ) != 0 ) {
773     //copy the desired MAC to the monitor ifreq
774     memcpy( monitor_ifreq.ifru.ifru_hwaddr.sa_data, parameter.src_mac, MAC_ADDRESS_SIZE );
775     //set the address-family of the monitor device
776     monitor_ifreq.ifru.ifru_hwaddr.sa_family = ARPHRD_ETHER;
777     //set MAC of monitor device
778     if ( ( ioctl( dev.fd_udp, SIOCSIFHWADDR, &monitor_ifreq ) ) == -1 ) {
779         //MAC of monitor_device could not be set
780         if ( DEBUG ) {
781             sprintf( buf, "Cannot set desired MAC to" \
782                     "monitor-device(%s) or no MAC set", parameter.monitor_device );
783             //print the error
784             perror( buf );
785             if( parameter.bridge == TRUE || parameter.bridge2 == TRUE ) {
786                 printf("Leaving MACs as is!\n");
787             } else {
788                 printf("Setting MAC of virtual-device(%s) " \
789                     "to MAC of monitor-device(%s)...%n" \
790                     , parameter.virtual_device, parameter.monitor_device );
791                 printf("Press <ENTER> to continue!%n");
792                 getchar();
793             }
794         }
795         //reset errno
796         errno = 0;
797         //set the mac of the network-card only, if it is a virtual card
798         if ( parameter.bridge == FALSE && parameter.bridge2 == FALSE ) {
799             //get MAC of monitor again for setting it to the monitor-device
800             if( ioctl( dev.fd_udp, SIOCGIFHWADDR, &monitor_ifreq ) == -1 ) {
801                 perror("Error getting hardware address on monitor-device");
802                 return -1;
803             }
804             //copy the MAC of the monitor-device to virtual-ifreq
805             memcpy( virtual_ifreq.ifr_ifru.ifru_hwaddr.sa_data, \
806                     monitor_ifreq.ifr_ifru.ifru_hwaddr.sa_data, MAC_ADDRESS_SIZE );
807             //set the address-family of the virtual device
808             virtual_ifreq.ifr_ifru.ifru_hwaddr.sa_family = ARPHRD_ETHER;
809             //set MAC of virtual device
810             if ( ( ioctl( dev.fd_udp, SIOCSIFHWADDR, &virtual_ifreq ) ) == -1 ) {
811                 perror("Setting MAC for virtual-device failed");
812                 return -1;
813             }
814             if ( DEBUG ) printf("Setting MAC of monitor-device(%s) failed -> " \
815                     "setting of virtual-device(%s) SUCCESSFUL!%n" \
816                     , parameter.monitor_device, parameter.virtual_device );
817     }
}

```

ANHANG B. QUELLCODE

```

818     //MAC of monitor_device could be set
819     } else if ( DEBUG ) printf("Setting MAC of monitor-device(%s) SUCCESSFUL!\n" \
820                 ,parameter.monitor_device);
821     //if desired MAC and MAC of monitor-device match, simply apply the MAC of the monitor device
822     //to the virtual device
823     } else {
824         if( parameter.bridge == FALSE && parameter.bridge2 == FALSE ) {
825             //copy the desired MAC to the virtual ifreq
826             memcpy( virtual_ifreq.ifru.ifru_hwaddr.sa_data, parameter.src_mac, MAC_ADDRESS_SIZE );
827             //set the address-family of the virtual device
828             virtual_ifreq.ifr_ifru.ifru_hwaddr.sa_family = ARPHRD_ETHER;
829             //set MAC of virtual device
830             if( ( ioctl( dev_fd_udp, SIOCSIFHWADDR, &virtual_ifreq ) ) == -1 ) {
831                 perror("Setting MAC for virtual-device failed");
832             }
833         }
834     }
835
836     //set IP for virtual device
837     //create a pointer of type "struct sockaddr_in" and cast the ifreq for being
838     //able to handle the ioctl-request, which demands this type of pointer
839     address_pointer = (struct sockaddr_in*) &virtual_ifreq.ifr_addr;
840     //set the address family according to packet(7)
841     address_pointer->sin_family = AF_INET;
842     //convert the ip from the commandline to binary code
843     inet_nton(AF_INET, parameter.ip, &address_pointer->sin_addr.s_addr);
844     //perform the ioctl itself: set the IP-address
845     if( ioctl( dev_fd_udp, SIOCSIFADDR, &virtual_ifreq ) == -1 ) {
846         perror("Could not set IP of virtual-device");
847         return -1;
848     }
849
850     //Activate the two devices
851
852
853     //##### MONITOR DEVICE #####
854     //set flag for interface UP
855     monitor_ifreq.ifr_flags = IFF_UP;
856     //put interface UP
857     if( ioctl( dev_fd_udp, SIOCSIFFLAGS, (caddr_t)&monitor_ifreq ) == -1 ) {
858         perror("Setting monitor-device UP failed");
859         return -1;
860     }
861
862     //##### VIRTUAL DEVICE #####
863     //set flag for interface UP
864     virtual_ifreq.ifr_flags = IFF_UP;
865
866     //put interface UP
867     if( ioctl( dev_fd_udp, SIOCSIFFLAGS, (caddr_t)&virtual_ifreq ) == -1 ) {
868         perror("Setting virtual-device UP failed");
869         return -1;
870     }
871
872     return 0;
873 }
874
875 void print_help_and_exit( char *name ) {
876     printf("USAGE: %s -m <monitor device> -n <ssid> -i <IP> \n" \
877           "optional: [-d] [-b <number of beacons>] [-v <virtual device>] [-s <src_mac>]" \
878           "[-t <beacon time limit>] [-h]\n\n", name);
879     printf("The options are:\n" \
880           "\nMANDATORY options:\n" \
881           "-m <monitor device>; monitor device is the wireless device, which is set into monitor mode. This \n" \
882           "can be achieved by simply issuing '#wlanconfig create athX wlandev wifi0 wlanmode monitor'" \
883           "at the command prompt.\n" \
884           "-n <ssid>; the ssid of the accesspoint, which is to be used. The program will only connect to \n" \
885           "accesspoints using this specified ssid.\n" \
886           "-i <IP>; the IP address the program shall work with. At running time, a virtual network is \n" \
887           "created, which will contain this IP-address. Communication will go over this address and \n" \
888           "the machine, which runs this program will be accessible over WLAN via this IP while this program \n" \
889           "is in use.\n" \
890           "\nADDITIONAL options:\n" \
891           "-<number of beacons>; the number of beacons, which are used for evaluating the signal strength \n" \
892           "of an accesspoint. The value is computed by creating the average value of the last specified \n" \
893           "amount of beacons. Setting this value to a higher value will make roaming more dull, lowering \n" \
894           "its value will result in roaming a lot faster. Lowering this value will NOT speed up roaming \n" \
895           "itself, but the point of time, when roaming is done if another accesspoint becomes stronger \n" \
896           "than the currently associated one. Setting this value to \"1\" will result in hopping between \n" \
897           "the accesspoints in the area, where the handover is to take place. The default value is 20.\n" \
898           "-v <virtual device>; the name of the virtual device created. This has almost no meaning, since \n" \
899           "this program will use any name. It just appears for example in \"#ifconfig -a\". The enduser \n" \
900           "should not be affected by this value in any way. The default name for this device is \"wlan0\". \n" \
901           "-s <src mac>; the MAC-address, which is to be set to the WLAN-card. If the WLAN-card doesn't \n" \
902           "support changing MAC-addresses, the hardware-set MAC-address is kept and used.\n" \
903           "-t <beacon time limit>; If an accesspoint goes down suddenly because of power loss or other \n" \
904           "dangerous errors, this is the time this program waits for beacons of the accesspoints to be \n" \
905           "received. Setting this value below the time that accesspoints send out beacons due to their \n" \
906           "configuration is nonsense, because if only one single beacon is missed, the program will try \n" \
907           "roaming at once, which might be inefficient due to the accesspoint being the strongest anyway. \n" \
908           "This option is to be set in microseconds. Default value for this option is 310000.\n" \
909           "-d: debugging output of the program. The software is more communicative. This should help when \n" \
910           "searching errors in configuration.\n" \
911           "-s <server-ip>; specify the server for logging purposes.\n" \
912           "-t <threshold>; set a threshold for roaming. This is the value in [db], that an AP must be \n" \
913           "stronger than the actually associated one to become the new AP\n" );
914     exit(-1);
915 }
916
917 void sigfunc(int sig) {
918     if(sig != SIGINT) {
919         return;
920     } else {
921         exit( 0 );
922     }
923 }
924
925 int main( int argc, char **argv ) {
926     int getopt_result, i;
927     int virtual_device_set = FALSE, source_mac_set = FALSE, number_of_beacons_set = FALSE;
928     int ssid_set = FALSE, ip_set = FALSE, monitor_device_set = FALSE, beacon_time_limit_set = FALSE;
929     struct ifreq monitor_ifreq;
930
931     signal(SIGINT, sigfunc);
932
933     first_accesspoint = NULL;

```

ANHANG B. QUELLCODE

```

936     chosen_accesspoint = NULL;
937     accesspoint_pointer = NULL;
938     if( getuid() != 0 ) {
939         printf("This program requires root privileges.\n");
940         exit(-1);
941     }
942     roam = FALSE;
943     parameter.threshold = 3;
944
945     while (( getopt_result = getopt(argc,argv,"ds:m:v:n:i:b:p:x:l:z:c:t:") ) != EOF ) {
946         switch( getopt_result ) {
947         case 'd':
948             DEBUG = 1;
949             break;
950         case 's':
951             for( i = 0; i < 17; i++ ) { //convert MAC-address in order to handle it later
952                 if ( ( i==2 && i!=5 && i!=8 && i!=11 && i!=14 ) ) {
953                     strcpy( &parameter.src_mac_dotted[i], &optarg[i] );
954                 } else parameter.src_mac_dotted[i]=':';
955             }
956             //convert imported ASCII-MAC into binary code in order to handle it in this program
957             mac_to_binary( parameter.src_mac_dotted, parameter.src_mac, MAC_ADDRESS_SIZE );
958             source_mac_set = TRUE;
959             break;
960         case 'm':
961             strcpy( parameter.monitor_device, optarg );
962             monitor_device_set = TRUE;
963             break;
964         case 'v':
965             strcpy( parameter.virtual_device, optarg );
966             virtual_device_set = TRUE;
967             break;
968         case 'n':
969             strcpy( parameter.ssid, optarg );
970             ssid_set = TRUE;
971             break;
972         case 'i':
973             strcpy( parameter.ip, optarg );
974             ip_set = TRUE;
975             break;
976         case 'b':
977             parameter.number_of_beacons = atoi( optarg );
978             number_of_beacons_set = TRUE;
979             break;
980         case 'c':
981             parameter.beacon_time_limit = atoi( optarg );
982             beacon_time_limit_set = TRUE;
983             break;
984         case 'h':
985             print_help_and_exit( argv[0] );
986             break;
987         case 'l':
988             //set connection environment for logging-software
989             strcpy( parameter.logging_server, optarg );
990             parameter.logging_address.sin_family = PF_INET;
991             inet_aton( parameter.logging_server, &parameter.logging_address.sin_addr );
992             parameter.logging_address.sin_port = htons(12345);
993             parameter.logging_socket = socket( PF_INET, SOCK_DGRAM, 0 );
994             connect( parameter.logging_socket, (struct sockaddr*)&parameter.logging_address,
995                      sizeof(struct sockaddr) );
996             LOGGING = TRUE;
997             break;
998         case 'z':
999             //set connection environment for logging-software
1000            strcpy( parameter.logging_server, optarg );
1001            parameter.logging_address.sin_family = PF_INET;
1002            inet_aton( parameter.logging_server, &parameter.logging_address.sin_addr );
1003            parameter.logging_address.sin_port = htons(12345);
1004            parameter.logging_socket = socket( PF_INET, SOCK_DGRAM, 0 );
1005            connect( parameter.logging_socket, (struct sockaddr*)&parameter.logging_address,
1006                      sizeof(struct sockaddr) );
1007            LOGGING_VERBOSE = TRUE;
1008            break;
1009        case 't':
1010            //set threshold for roaming
1011            parameter.threshold = atoi( optarg );
1012            break;
1013        default:
1014            print_help_and_exit( argv[0] );
1015            break;
1016        }
1017    }
1018
1019    if ( !virtual_device_set ) strcpy( parameter.virtual_device, "wlan0", IFNAMSIZ );
1020
1021    if ( !ssid_set || !ip_set || !monitor_device_set ) print_help_and_exit( argv[0] );
1022    if ( !number_of_beacons_set ) parameter.number_of_beacons = 20;
1023    if ( !beacon_time_limit_set ) parameter.beacon_time_limit = 310000;
1024
1025    /* create our own session */
1026    setsid();
1027
1028    for(i=0; i<DATA_BUFFER; i++) {
1029        memset( &vnic_data[i], 0, sizeof(struct data));
1030        memset( &wlan_data[i], 0, sizeof(struct data));
1031    }
1032
1033    //check for size of beacon evaluations
1034    if( parameter.number_of_beacons > 600 ) {
1035        printf("Number of beacons higher than 600, please choose a lower value!\n");
1036        exit(-1);
1037    }
1038
1039    //open sockets for communication
1040    if ( DEBUG )
1041        printf("Opening sockets...\n");
1042    if ( open_sockets() == -1 ) {
1043        printf("Error opening sockets\n");
1044        exit(-1);
1045    } else if ( DEBUG ) printf("Opening sockets SUCCESSFUL!\n");
1046
1047    //copy MAC of monitor-device to parameter.src_mac
1048    if ( !source_mac_set ) {
1049        memset( &monitor_ifreq, 0, sizeof(struct ifreq) );
1050        strncpy( monitor_ifreq.ifr_ifrn.ifrn_name, parameter.monitor_device, IFNAMSIZ );
1051        if( ioctl( dev.fd_udp, SIOCGIFHWADDR, &monitor_ifreq ) == -1 ) {

```

ANHANG B. QUELLCODE

```
1052         perror("Getting MAC of monitor-device failed");
1054     }
1056 } memcpy ( &parameter.src_mac, monitor_ifreq.ifr_ifru.ifru_hwaddr.sa_data, MAC_ADDRESS_SIZE );
1058 //configure the virtual device
1059 if ( DEBUG )
1060     printf("Configuring device(%s) to %s...\n",parameter.virtual_device, parameter.ip);
1061 if ( configure_devices() == -1 ) {
1062     printf("Error configuring device(%s)\n",parameter.virtual_device);
1063     exit(-1);
1064 } else if ( !DEBUG ) printf("Configuring device(%s) SUCCESSFUL!\n",parameter.virtual_device);
1065 //do an initial scan
1066 if ( DEBUG )
1067     printf("Doing an initial scan for APs; wait one sec or until an AP comes up...\n");
1068 if ( initial_scan() == -1 ) {
1069     printf("Error performing an initial scan!\n");
1070     exit(-1);
1071 } else if ( !DEBUG ) printf("Initial AP-scan SUCCESSFUL!\n");
1072 //start passing packets and roam
1073 if ( DEBUG )
1074     printf("Looping endless to pass packets and roam...\n");
1075 manage_threads();
1076 exit(0);
}
```

ANHANG B. QUELLCODE

Listing B.2: `roameo.h`

```

1 #include <sys/socket.h>           //for socket()
2 #include <time.h>                //used for timestamps
3 #include <string.h>               //for memcmp()
4 #include <asm/types.h>             //for __u8 etc.
5 #include <linux/if.h>              //for struct ifreq
6 #include <linux/if_packet.h>        //for struct sockaddr_ll
7 #include <linux/if_fddi.h>          //for struct fddi_snap_hdr
8 #include <linux/if_tun.h>           //for IFF_TAP, TUNSETIFF and IFF_NO_PI
9 #include <stdio.h>                 //for printf(), etc.
10 #include <stdlib.h>                //for malloc()
11 #include <unistd.h>                //for getopt() and read()
12 #include <sys/time.h>               //for gettimeofday()
13 #include <errno.h>                  //for errno-numbers
14 #include <sys/types.h>               //for open()
15 #include <sys/stat.h>                //for open()
16 #include <sys/types.h>               //for ioctl()
17 #include <sys/ioctl.h>               //for ioctl()
18 #include <netinet/in.h>              //for ntohs(), htons() in big_endian- and little_endian-version
19 #include <net/ether.h>                //for ethernet-macros, struct ether_header
20 #include <arpa/inet.h>               //for inet_pton(3)
21 #include <signal.h>                 //for signal(2)
22
23 #include <pthread.h>                //for pthread_create(3)
24 #include <semaphore.h>              //for sem_wait
25
26 #define max(a,b) ((a)>(b) ? (a):(b))
27 #define MAC_ADDRESS_SIZE 6           /* Size of a MAC-address */
28 #define SSID_MAX_SIZE 32            /* Maximum size of an SSID */
29 #define TRUE 1
30 #define FALSE 0
31 #define ARPHRD_ETHER 1             /* originally taken from <libnet/libnet-headers.h> */
32 #define ESSID_TAG_NUMBER 0          /* Tag ID for ESSID */
33 #define MAXBUFFER 4096
34
35 #define DATA_BUFFER 100
36 #define LOCK_1
37 #define UNLOCK_1
38 #define PERM 0666
39 #define KEY 123458L
40
41 #define ROAMING 0x1
42 #define FIELD_INTENSITY 0x2
43
44 struct logging {
45     long id;                      //id of rtt-packet
46     struct timeval time;           //time value of rtt-packet
47     char flags;                   //flag for performed roaming, information
48     char old_ap[20];              //MAC of old AP
49     char new_ap[20];              //MAC of new AP
50     int field_intensity;          //field intesity of AP
51 };
52
53 typedef struct devices {
54     int fd_wlan;                  //socket for monitor-device
55     int fd_vnic;                  //socket for virtual card
56     int fd_udp;                   //udp-socket for configuring devices
57 } devices;
58
59 extern devices dev;
60
61 typedef struct ieee802_11_bacon_mgt_frame {
62     //first three ones are fixed parameters
63     u_char timestamp[8];
64     u_short beacon_interval;       //timeinterval at which new beacons arrive
65     u_short capability_information; //contains 16 bits each for different information
66     //appended parameters go here
67     unsigned char tag_number;      //usually "00" = ESSID
68     unsigned char tag_length;      //length of ESSID and then ESSID to follow
69     char tag_interpretation[SSID_MAX_SIZE];
70 } ieee802_11_bacon_mgt_frame;
71
72 typedef struct ieee802_11_hdr {
73     u_char subtype;
74     #define IEEE80211_TYPE_DATA 0x08
75     u_char flags;
76     #define IEEE80211_TO_DS 0x01
77     #define IEEE80211_FRM_DS 0x02
78     #define IEEE80211_MORE_FRAG 0x04
79     #define IEEE80211_RETRY 0x08
80     #define IEEE80211_PWR_MGT 0x10
81     #define IEEE80211_MORE_DATA 0x20
82     #define IEEE80211_WEP_FLAG 0x40
83     #define IEEE80211_ORDER_FLAG 0x80
84     u_short duration;
85     u_char addr1[MAC_ADDRESS_SIZE];
86     u_char addr2[MAC_ADDRESS_SIZE];
87     u_char addr3[MAC_ADDRESS_SIZE];
88     u_short frag_and_seq;
89 } ieee802_11_hdr;
90
91 typedef enum {
92     mgt_assocRequest = 0,
93     mgt_assocResponse = 1,
94     mgt_reassocRequest = 2,
95     mgt_reassocResponse = 3,
96     mgt_probeRequest = 4,
97     mgt_probeResponse = 5,
98     mgt_beacon = 8,
99     mgt_disassoc = 10,
100    mgt_auth = 11,
101    mgt_deauth = 12,
102    data = 20
103 } wifi_frametype;
104
105 /* prism header is prepended as default to atheros monitor packets */
106 typedef struct {
107     unsigned int did;
108     unsigned short status;
109     unsigned short len;
110     unsigned int data;
111 } p80211item_uint32_t;
112
113 //This structure provides different parameters which are read from a config-file
114 struct PARAMETER {
115     char monitor_device[IFNAMSIZ];

```

ANHANG B. QUELLCODE

```
116     char virtual_device[IFNAMSIZ];
117     char ssid[SSID_MAX_SIZE];
118     unsigned char src_mac[MAC_ADDRESS_SIZE];
119     char src_mac_dotted[20];
120     unsigned char dst_mac[MAC_ADDRESS_SIZE];
121     char dst_mac_dotted[20];
122     char ip[16];
123     int number_of_beacons;
124     int beacon_time_limit;
125 //These two variables are need only in bridged version!!!
126     int bridge;
127     int bridge2;
128     char logging_server[16];
129     struct sockaddr_in logging_address;
130     int logging_socket;
131     int threshold;
132 };
133
134 //Structure to handle accesspoints
135 struct AP {
136     u_char mac[MAC_ADDRESS_SIZE];
137     int counter;
138     int value;
139     //be able to calculate average value of signal strength
140     //for one minute (if beacons arrive in 100ms interval!)
141     int signal_database[600];
142     int considerable_results_for_changing_AP;
143     struct timeval timestamp;
144     struct AP *next;
145 };
146
147 typedef struct {
148     u_char tag;
149     u_char length;
150 } ieee_802_11_tag;
151
152 typedef enum {
153     tagSSID = 0,
154     tagRates = 1,
155     tagChannel = 3,
156     tagVendorSpecific = 0xDD
157 } i81tag;
158
159 struct data {
160     int size;
161     int time;
162     u_char data[MAXBUFFER];
163 };
164
165 //Functions used in roameo.c
166 int evaluate_beacon(u_char *mac, unsigned int signal);
167 int check_timestamps( void );
168 int delete_accesspoint( struct AP *liquidation );
169 void *read_from_vnic(void *arg);
170 void *read_from_vlan(void *arg);
171 void *write_on_vnic(void *arg);
172 void *write_on_vlan(void *arg);
173 int manage_threads( void );
174 int ether_ntop(const u_char address[MAC_ADDRESS_SIZE], char * buf);
175 int mac_to_binary(const char * orig, unsigned char * mac, int macmax);
176 int tap_open(char *dev);
177 int openraw( char *iface, int fd);
178 int initial_scan( void );
179 int open_sockets( void );
180 int configure_devices( void );
181 void print_help_and_exit( char *name );
182 void close_sockets( void );
```

ANHANG B. QUELLCODE

Listing B.3: wlan_mgmt.c

```

1 #include "roameo.h"
2 #include "wlan_mgmt.h"
4 /* both below useful to inspect misc buffers */
5 void hexdump(char *buffer, int len) {
6     int i;
8     for(i=0; i<len; i++) {
9         if(i&16 == 0) printf("\n%04x: ", i);
10        printf("%02x ", (unsigned char)(buffer[i]));
11    }
12    printf("\n");
13    fflush(stdout);
14 }
16 /*Association request */
17 int send_association_request(u_char *ap_mac, u_char *src_mac, char *essid) {
18     struct association_frame frame;
19     struct association_response_frame response_frame;
22     memset( &frame, 1, sizeof(struct association_frame) );
23     memset( &response_frame, 0, sizeof(struct association_response_frame) );
24     frame.header.frame_control = ASSOCIATION_REQUEST;
25     frame.header.duration = DEFAULT_DURATION;
26     memcpy( frame.header.address1, ap_mac, ETH_ALEN );
27     memcpy( frame.header.address2, src_mac, ETH_ALEN );
28     memcpy( frame.header.address3, ap_mac, ETH_ALEN );
29     frame.capability_information = 0x0420; //defaults to 0, since this is an association request
30     frame.listen_interval = 100; //defaults to 100, every 100 beacons we can receive
31     //management entity
32     /*SSID element */
33     frame.ssid_element.element_id = SSID;
34     frame.ssid_element.length = strlen(essid);
35     memcpy( frame.ssid_element.ssid, essid, strlen(essid) );
36
38     /*Supported rates element */
37     frame.supported_rates_element.element_id = SUPPORTED_RATES;
38     frame.supported_rates_element.length = 8;
39     frame.supported_rates_element.rates[0] = RATE_1_B;
40     frame.supported_rates_element.rates[1] = RATE_2_B;
41     frame.supported_rates_element.rates[2] = RATE_5_5_B;
42     frame.supported_rates_element.rates[3] = RATE_11_B;
43     frame.supported_rates_element.rates[4] = RATE_18;
44     frame.supported_rates_element.rates[5] = RATE_24;
45     frame.supported_rates_element.rates[6] = RATE_36;
46     frame.supported_rates_element.rates[7] = RATE_54;
47
48     frame.power_capability_element.element_id = POWER_CAPABILITY;
49     frame.power_capability_element.length = 2;
50     frame.power_capability_element.interpretation[0] = 0;
51     frame.power_capability_element.interpretation[1] = 84;
52
53     frame.extended_supported_rates_element.element_id = EXTENDED_SUPPORTED_RATES;
54     frame.extended_supported_rates_element.length = 4;
55     frame.extended_supported_rates_element.rates[0] = RATE_6;
56     frame.extended_supported_rates_element.rates[1] = RATE_9;
57     frame.extended_supported_rates_element.rates[2] = RATE_12;
58     frame.extended_supported_rates_element.rates[3] = RATE_48;
59
60     //Move memory to make SSID stay dynamic in size
61     //this is done with the pointer to "length", since a pointer to ssid points to
62     //wrong memory space
63     memmove( &frame.ssid_element.length + strlen(essid) + 1, &frame.supported_rates_element,
64             ( sizeof(struct supported_rates_element)
65               + sizeof(struct power_capability_element)
66               + sizeof(struct extended_supported_rates_element) ) );
67
68     write( dev.fd_wlan, &frame, sizeof(struct association_frame) - SSID_MAX_SIZE + strlen(essid) );
69
70     return( SUCCESS );
71 }
74 /*Probe request */
75 int send_probe_request(u_char *src_mac) {
76
77     struct probe_request_frame frame;
78     memset( &frame, 0, sizeof(struct probe_request_frame) );
79     frame.header.frame_control = PROBE_REQUEST;
80     frame.header.duration = DEFAULT_DURATION;
81     memcpy( frame.header.address1, BROADCAST, ETH_ALEN );
82     memcpy( frame.header.address2, src_mac, ETH_ALEN );
83     memcpy( frame.header.address3, BROADCAST, ETH_ALEN );
84     frame.header.sequence_control = 0; //set by driver
85     frame.ssid_element.element_id = SSID;
86     frame.ssid_element.length = BROADCAST_SSID;
87
88     return( write( dev.fd_wlan, &frame, sizeof(struct probe_request_frame) ) );
89 }
92 /*Disassociation request */
93 int send_disassociation_request(u_char *ap_mac, u_char *src_mac, char *essid) {
94
95     struct disassociation_frame frame;
96     memset( &frame, 0, sizeof(struct disassociation_frame) );
97     frame.header.frame_control = DISASSOCIATION_REQUEST;
98     frame.header.duration = DEFAULT_DURATION;
99     memcpy( frame.header.address1, ap_mac, ETH_ALEN );
100    memcpy( frame.header.address2, src_mac, ETH_ALEN );
101    memcpy( frame.header.address3, ap_mac, ETH_ALEN );
102    frame.reason_code = STATION_LEAVEING_BSS;
103
104    return( write( dev.fd_wlan, &frame, sizeof(struct disassociation_frame) ) );
105 }
108 /*Authentication request */
109 int send_authentication_request(u_char *ap_mac, u_char *src_mac) {
110
111     struct authentication_frame frame;
112     struct authentication_response_frame response_frame;
113
114     memset( &frame, 0, sizeof(struct authentication_frame) );

```

ANHANG B. QUELLCODE

```
116     memset( &response_frame, 0, sizeof(struct authentication_response_frame) );
118     frame.header.frame_control = (uint16_t)(AUTHENTICATION);
119     frame.header.duration = DEFAULT_DURATION;
120     memcpy( frame.header.address1, ap_mac, ETH_ALEN );
121     memcpy( frame.header.address2, src_mac, ETH_ALEN );
122     memcpy( frame.header.address3, ap_mac, ETH_ALEN );
123     frame.header.sequence_control = 0; //set by driver
124     frame.authentication_algorithm = OPEN;
125     frame.authentication_sequence_number = RESERVED_SEQUENCE_NUMBER;
126
127     write( dev.fd_wlan, &frame, sizeof(struct authentication_frame) );
128
129     return( SUCCESS );
130 }
131
132 /*Deauthentication request */
133 int send_deauthentication_request(u_char *ap_mac, u_char *src_mac) {
134
135     struct deauthentication_frame frame;
136     memset( &frame, 0, sizeof(struct deauthentication_frame) );
137     frame.header.frame_control = DEAUTHENTICATION;
138     frame.header.duration = DEFAULT_DURATION;
139     memcpy( frame.header.address1, ap_mac, ETH_ALEN );
140     memcpy( frame.header.address2, src_mac, ETH_ALEN );
141     memcpy( frame.header.address3, ap_mac, ETH_ALEN );
142     frame.header.sequence_control = 0; //set by driver
143     frame.reason = STATION_LEAVING_BSS;
144
145     return( write( dev.fd_wlan, &frame, sizeof(struct deauthentication_frame) ) );
146 }
147
148 /*blind association without any feedback */
149 int assoc(u_char *ap_mac, u_char *src_mac, char *essid) {
150
151     if ( send_authentication_request(ap_mac, src_mac) != 0 ) {
152         perror("Authentication request could not be sent");
153         exit(-1);
154     }
155     if ( send_association_request(ap_mac, src_mac, essid) != 0 ) {
156         perror("Authentication request could not be sent");
157         exit(-1);
158     }
159     return 0;
160 }
161
162 /*blind disassociation without any feedback */
163 int disassoc( u_char *ap_mac, u_char *src_mac, char *essid ) {
164
165     /*! ACCORDING TO IEEE802.11 NOT NEEDED!!!!
166     /*if ( send_disassociation_request(ap_mac, src_mac, essid) <= 0 ) {
167         perror("Disassociation request could not be sent");
168         exit(-1);
169     }*/
170     if ( send_deauthentication_request(ap_mac, src_mac) <= 0 ) {
171         perror("Deauthentication request could not be sent");
172     }
173     return 0;
174 }
```

ANHANG B. QUELLCODE

Listing B.4: wlan_mgmt.h

```

/*General macros */
2 #define BROADCAST "\xff\xff\xff\xff\xff\xff\xff"
#define DEFAULT_DURATION 0x13a /*default value of 314 */
4 #define BROADCAST_SSID 0

6 /*Management types */
#define ASSOCIATION_REQUEST 0x0 /*Type: Management, Subtype: Probe Request , Type: 0      */
8 #define ASSOCIATION_RESPONSE 0x10 /*Type: Management, Subtype: Probe Request , Type: 1      */
#define PROBE_REQUEST 0x40 /*Type: Management, Subtype: Probe Request , Type: 4      */
10 #define DISASSOCIATION_REQUEST 0xA0 /*Type: Management, Subtype: Disassociation , Type: 10     */
#define AUTHENTICATION 0xB0 /*Type: Management, Subtype: Authentication , Type: 11     */
12 #define DEAUTHENTICATION 0xC0 /*Type: Management, Subtype: Deauthentication , Type: 12     */
#define ACKNOWLEDGEMENT 0xD4 /*Type: Control, Subtype: Acknowledgement , Type: 29      */
14

16 #define MANAGEMENT_MASK 0xFF /* 1111111 for filtering type and subtype */

18 /*Element IDs */
#define SSID 0
#define SUPPORTED_RATES 1
20 #define FH_PARAMETER_SET 2
#define DS_PARAMETER_SET 3
22 #define CF_PARAMETER_SET 4
#define TIN 5
24 #define IBSS_PARAMETER_SET 6
#define RESERVED_ELEMENT_ID 7
26 #define CHALLENGE_TEXT 16
#define POWER_CAPABILITY 33
28 #define EXTENDED_SUPPORTED_RATES 50

30 /*reason codes for deauthenticating and disassociating */
#define RESERVED_0
#define UNSPECIFIED_1
#define AUTHENTICATION_LONGER_VALID_2
34 #define STATION_LEAVE_IBSS_3
#define INACTIVITY_4
36 #define UNABLE_TO_HANDLE_ALL_STATIONS_5
#define CLASS_2_FROM_NONAUTENTICATED_STATION_6
38 #define CLASS_3_FROM_NONAUTENTICATED_STATION_7
#define STATION_LEAVE_BSS_8
40 #define STATION_NOT_AUTENTICATED_9
#define POWER_CAPABILITY_UNACCEPTABLE_10
42 #define SUPPORTED_CHANNELS_UNACCEPTABLE_11
#define FOUR_WAY_HANDSHAKE_TIMEOUT_15
44 #define GROUP_KEY_UPDATE_TIMEOUT_16

46 /*Transmission Rates (each rate has the unit 500 kbit/s */
#define RATE_1 2
#defin RATE_2 4
#define RATE_5_5 11
50 #define RATE_6 12
#defin RATE_9 18
52 #define RATE_11 22
#defin RATE_12 24
54 #defin RATE_18 36
#defin RATE_24 48
56 #defin RATE_36 72
#defin RATE_48 96
58 #defin RATE_54 108
/*802.11b-Rates have a the first bit set to 1. This explains the differences below! */
60 #define RATE_1_B 0x82
#defin RATE_2_B 0x84
62 #define RATE_5_5_B 0xb
#defin RATE_11_B 0x96
64

66 #define WLAN_DEVNAMELEN_MAX 16
68
#defin OPEN 0
#defin SHARED_KEY 1
#defin SUCCESS 0
70 #define RESERVED_SEQUENCE_NUMBER 1

72 typedef struct prism_hdr {
    u_int msg_code;
74    u_int msg_length;
    char cap_device[WLAN_DEVNAMELEN_MAX];
76    p80211item_uint32_t hosttime;
    p80211item_uint32_t mactime;
78    p80211item_uint32_t channel;
    p80211item_uint32_t rss;
80    p80211item_uint32_t sq;
    p80211item_uint32_t signal;
82    p80211item_uint32_t noise;
    p80211item_uint32_t rate;
84    p80211item_uint32_t istx;
    p80211item_uint32_t frrlen;
86 } prism_hdr;

88 struct ssid_element {
    uint8_t element_id;
90    uint8_t length;           //equals 0 for broadcasts
    u_char ssid[SSID_MAX_SIZE];
92 };
94 struct supported_rates_element {
    uint8_t element_id;
96    uint8_t length;
    u_char rates[8];
98 };
100 struct extended_supported_rates_element {
    uint8_t element_id;
102    uint8_t length;
    u_char rates[4];
104 };
106 struct power_capability_element {
    uint8_t element_id;
108    uint8_t length;
    u_char interpretation[2];
110 };
112 struct management_frame_header {
    uint16_t frame_control;
    uint16_t duration;
    u_char address1[MAC_ADDRESS_SIZE]; //destination
114

```

ANHANG B. QUELLCODE

```
116     u_char address2[MAC_ADDRESS_SIZE]; //source
117     u_char address3[MAC_ADDRESS_SIZE]; //bssid
118     uint16_t sequence_control; //is set automatically by driver
119 };
120
122 struct probe_request_frame {
123     struct management_frame_header header;
124     /*frame body from here on*/
125     struct ssid_element ssid_element;
126 };
127
128 struct authentication_frame {
129     struct management_frame_header header;
130     /*frame body from here on*/
131     uint16_t authentication_algorithm;
132     uint16_t authentication_sequence_number;
133     uint16_t authentication_status_code; //not present with no encryption
134 };
135
136 struct authentication_response_frame {
137     struct prism_hdr prism_header;
138     struct management_frame_header header;
139     /*frame body from here on*/
140     uint16_t authentication_algorithm;
141     uint16_t authentication_sequence_number;
142     uint16_t authentication_status_code; //not present with no encryption
143 };
144
145 struct deauthentication_frame {
146     struct management_frame_header header;
147     /*frame body from here on*/
148     uint16_t reason;
149 };
150
151 struct association_frame {
152     struct management_frame_header header;
153     /*frame body from here on*/
154     uint16_t capability_information;
155     uint16_t listen_interval;
156     struct ssid_element ssid_element;
157     struct supported_rates_element supported_rates_element;
158     struct power_capability_element power_capability_element;
159     struct extended_supported_rates_element extended_supported_rates_element;
160 };
161
162 struct association_response_frame {
163     struct prism_hdr prism_header;
164     struct management_frame_header header;
165     /*frame body from here on*/
166     uint16_t capability_information;
167     uint16_t association_status_code;
168     u_char association_id[2];
169     struct supported_rates_element supported_rates_element;
170     struct extended_supported_rates_element extended_supported_rates_element;
171     u_char vendor_specific_information[8]; //this is ignored by us!!!
172 };
173
174 struct disassociation_frame {
175     struct management_frame_header header;
176     /*frame body from here on*/
177     uint16_t reason_code;
178 };
179
180 int send_authentication_request(u_char *ap_mac, u_char *src_mac);
181 int send_deauthentication_request(u_char *ap_mac, u_char *src_mac);
182 int send_probe_request(u_char *src_mac);
183 int send_association_request(u_char *ap_mac, u_char *src_mac, char *essid);
184 int send_disassociation_request(u_char *ap_mac, u_char *src_mac, char *essid);
185 int send_clear_to_send( u_char *dst_mac );
186 int assoc(u_char *ap_mac, u_char *src_mac, char *essid);
187 int disassoc( u_char *ap_mac, u_char *src_mac, char *essid );
```

ANHANG B. QUELLCODE

Listing B.5: `makefile`

```
#####
# Purpose: Makefile for Roameo
# Author.: Tom Stoeveken, modified by Dennis Borgmann
# Version: 1.1
#
#####
SRC1 = roameo.c
OBJ1 = $(SRC1:.c=.o)
HDR1 = $(SRC1:.c=.h)
OUT1 = $(SRC1:.c=)

SRC2 = wlan_mgmt.c
OBJ2 = $(SRC2:.c=.o)
HDR2 = $(SRC2:.c=.h)
OUT2 = $(SRC2:.c=)

## compile with "-pg" for profiling informations
CC = gcc
STRIP = strip

## -g for GDB-infos in the binary
## -I directory to search for header files
## -W warn on errors
## -pg include profiling options
## -O3 optimize code
CFLAGS = -O3 -Wall -I.
LFLAGS = -lpthread

### Build rules ###
all: $(OUT1)
## strip to delete symbols from binary to make binary become smaller
strip $(OUT1)
@echo all done!!

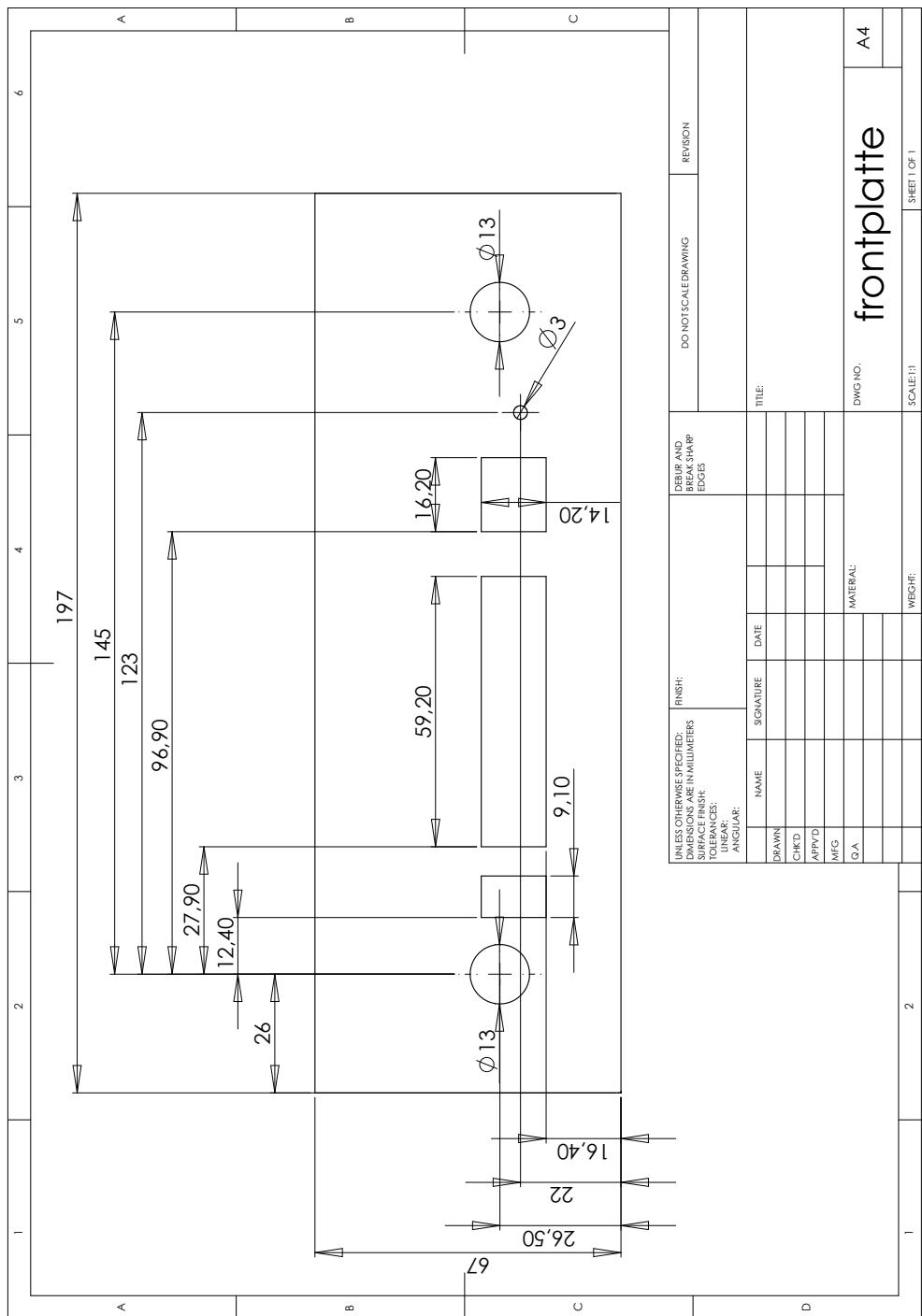
$(OUT1): $(HDR1) $(HDR2) $(SRC2)
    $(CC) $(CFLAGS) -o $(OBJ1) $(SRC1)
    $(CC) -c $(CFLAGS) -o $(OBJ2) $(SRC2)
    $(CC) $(LFLAGS) -o $@ $(OBJ1) $(OBJ2)

tgz: clean
    mkdir -p backups
    tar czvf ./backups/wlan_embi_drv_`date +"%Y_%m_%d_%H.%M.%S"`.tgz --exclude backups *
clean:
    rm -f *.o *
    rm -f $(OUT1)
```

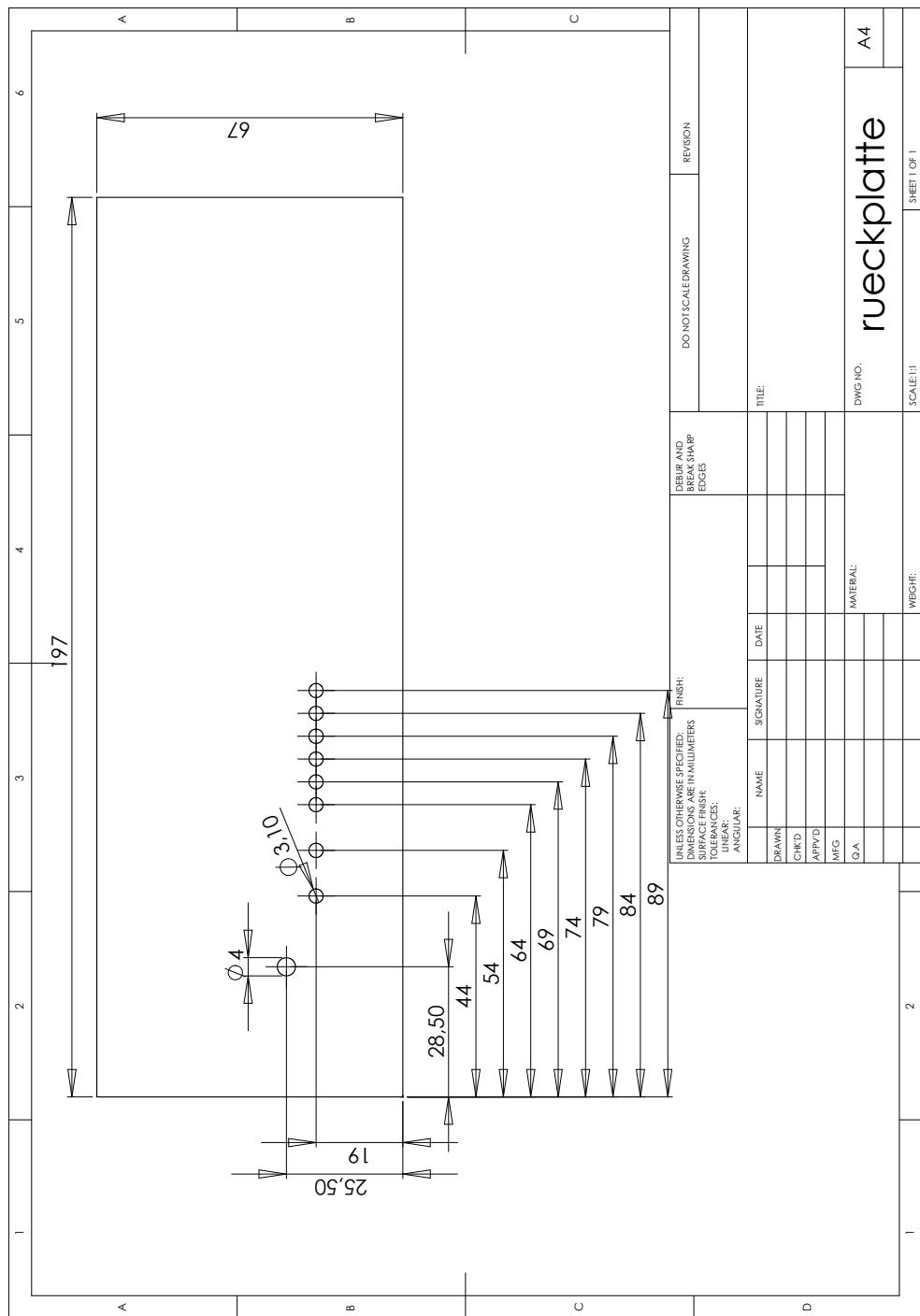
Anhang C

CAD-Zeichnungen

ANHANG C. CAD-ZEICHNUNGEN



ANHANG C. CAD-ZEICHNUNGEN



Literaturverzeichnis

- [1] packet(7). Linux Programmer's Manual, April 1999.
- [2] iwlist(8). Linux Programmer's Manual, Juni 2004.
- [3] Advanced micro devices. <http://www.amd.com>, Januar 2007.
- [4] Atheros communications. <http://www.atheros.com>, Januar 2007.
- [5] Atlas copco. <http://www.atlascopco.com>, März 2007.
- [6] Bochs ia-32 emulator project. <http://bochs.sourceforge.net/>, März 2007.
- [7] Elcard wireless systems oy. <http://www.elcard.fi/>, März 2007.
- [8] Integrated electronics. <http://www.intel.com>, Januar 2007.
- [9] Linksys. <http://www.linksys.com/de>, Januar 2007.
- [10] Lippert. <http://www.lippert-at.com>, Januar 2007.
- [11] Lkab. <http://www.lkab.com>, März 2007.
- [12] Mips technologies. <http://www.mips.com>, Januar 2007.
- [13] Openwrt. <http://www.openwrt.org>, Januar 2007.
- [14] Voyage linux. <http://linux.voyage.hk>, Januar 2007.
- [15] Fabrice Bellard. qemu-manpage. Linux Programmer's Manual, August 2006.
- [16] Fabrice Bellard. qemu. <http://fabrice.bellard.free.fr/qemu/>, Januar 2007.
- [17] IEEE Standards Board. *Wireless LAN Medium Access Control(MAC) and Physical Layer(PHY) Specifications*. IEEE, 445 Hoes Lane, Piscataway, NJ, USA, 802.11 edition, 1999.

- [18] Gerald Combs. wireshark. <http://www.wireshark.org/>, März 2007.
- [19] Christophe Devine. Aircrack-ng. <http://www.aircrack-ng.org>, Januar 2007.
- [20] Pascal Dornier. Pc engines. <http://www.pcengines.ch>, Januar 2007.
- [21] Inc. Free Software Foundation. Gnu public license. <http://www.gnu.org/licenses/gpl.html>, Juni 1991.
- [22] InnoTek Systemberatung GmbH. Virtual box. <http://www.virtualbox.org/>, März 2007.
- [23] Christian Grimm, editor. *Rechnernetze I - Teil 11*. Fachgebiet Distributed Virtual Reality (DVR), Universität Hannover, 2006.
- [24] Xavier Leroy. semaphores(3). Linux Programmer's Manual.
- [25] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. iperf. <http://dast.nlanr.net/Projects/Iperf/>, Januar 2007.
- [26] Jean Tourrilhes. iwpriv(8). Linux Programmer's Manual, Oktober 1996.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt 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.

Ladbergen, März 2007

Dennis Borgmann