Hochschule für Technik,
Wirtschaft und Kultur
Leipzig

Masterarbeit zum Thema

Entwurf eines objektorientierten WLAN-Stacks für das Opensource Betriebssystem Haiku

Zur Erlangung des akademischen Grades

Master of Science

Eingereicht von: Colin Günther, 06IN-M
Betreut von: Prof. Dr. rer. nat. Klaus Bastian
Zweitgutachter: Prof. Dr.-Ing. Dietmar Reimann

Eidesstattliche Erklärung

Ich versichere, die Masterarbeit selbständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Brandis, den 30. April 2010

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Konventionen

Bei Bezugnahme auf einzelne IEEE Standards wird entsprechend des „IEEE Standards Style Manual" ([Style, Kapitel 13.8, Seite 23]) verfahren:

1. Wenn das Dokument an sich gemeint ist, wird IEEE Std 1234 verwendet.
2. Ist die standardisierte Technologie gemeint, wird mit IEEE 1234 darauf Bezug genommen.

Alternativ wird anstelle von IEEE 802.11 auch die Abkürzung WLAN (Wireless-Local-Area-Network) verwendet.

Der IEEE Std 802.11 definiert viele domänenspezifische Begriffe und Abkürzungen, welche naturgemäß in Englisch gehalten sind. Die vorliegende Arbeit versucht jeden Begriff zunächst in seiner deutschen Fassung einzuführen, wird dann aber im weiteren Verlauf bei der englischen Variante bleiben. Dies soll das Zurechtfinden im Standard erleichtern.

Bei Bezugnahme auf den objektorientierten Entwurf von IEEE 802.11 wird bevorzugt die Abkürzung WLAN-Stack verwendet. Diese ist zwar technisch gesehen mehrfach belegt[1], im Kontext dieser Masterarbeit aber stets eindeutig.

[1] Es gibt weitere drahtlose Netzwerktechnologien, beispielsweise WiMAX (IEEE Std 802.16).

1. Einleitung und Motivation

IEEE 802.11 ist eine weit verbreitete Technologie für die Errichtung von Drahtlosnetzwerken. Im Gegensatz zum drahtgebundenen IEEE 802.3 – auch als Ethernet bekannt – sind die Anforderungen an die MAC-Schicht (Abschnitt 2.1) wesentlich komplexer.

Die existierenden, überwiegend strukturorientierten Implementierungen (Kapitel 4) dieses Standards, erlauben zwar schnellen, effizienten Code, erschweren jedoch auf der anderen Seite das Verständnis und die Erweiterung dieses Codes. In Anbetracht der ständigen Weiterentwicklung von IEEE 802.11 seit seiner ersten Veröffentlichung im Jahre 1997 (Abschnitt 2.7) ist besonders der Aspekt der Erweiterbarkeit einer Implementierung von zentraler Bedeutung.

Die vorliegende Masterarbeit demonstriert dabei unter der Verwendung objektorientierter Konzepte, wie sich die Verständlichkeit und die Erweiterbarkeit einer Implementierung von IEEE 802.11 steigern lassen. Dabei wird der Fokus auf den Entwurf eines objektorientierten WLAN-Stacks und weniger auf seine Implementierung gelegt, damit sich der Umfang der Masterarbeit in einem überschaubaren Rahmen bewegt. Die Problematik der Implementierung wird jedoch mit berücksichtigt. Dafür wird im Abschnitt 4.5 ein detaillierter Implementierungsplan vorgestellt und es werden Ideen zum Testen der Implementierung diskutiert (Abschnitt 4.6).

Als Einsatzgebiet für die Implementierung wurde das Opensourcebetriebsystem Haiku ausgewählt. Zum einen erlaubt Haiku die Ausführung von objektorientiertem Code im Kernelland und zum Anderen besaß es zu Beginn der Masterarbeit noch keinen eigenen WLAN-Stack. Gerade das Fehlen eines WLAN-Stacks führte zur Initialmotivation, einen WLAN-Stack für Haiku bereitzustellen. Durch eine Analyse der WLAN-Stacks anderer Betriebssysteme[1] hat sich gezeigt, dass diese zum Einen strukturorientiert und zum Anderen – wo Opensource – schwierig zu verstehen sind. Daraus ergab sich schließlich die Hauptmotivation für diese Masterarbeit, einen objektorientierten Neuentwurf auszuarbeiten.

Die Masterarbeit ist dabei so gegliedert, dass zunächst die Grundlagen für das Verständnis des objektorientierten Entwurfs gelegt werden. Dabei wird zunächst die Begriffswelt von IEEE 802.11 (Kapitel 2) vorgestellt, so dass der Leser am Ende dieses Kapitels die Fachausdrücke zuordnen kann und einen Eindruck von der Komplexität dieser Technologie erhält.

Daran schließt sich eine Einführung in die Geschichte und die Netzwerarchitektur von Haiku (Kapitel 3) an. Am Ende dieses Kapitels wird der Leser das Betriebssystem und

[1] Diverse Versionen von Mac OS X, Windows, Linux und BSD

seine Besonderheiten verstehen und wird ein fundiertes Wissen über die Komponenten von Haikus Netzwerkstack erhalten haben.

Anschließend wird der objektorientierte Entwurf behandelt (Kapitel 4). Dabei wird demonstriert, wie objektorientierte Prinzipien eingesetzt werden, um Erweiterbarkeit und Verständlichkeit des Entwurfs und einer späteren Implementierung zu erhöhen. Nach Lesen dieses Kapitels wird auch deutlich, wie die Spezifikationen von IEEE Std 802.11 in den Entwurf eingeflossen sind und wie in Haikus Netzwerkarchitektur die Implementierung des Entwurfs einzufügen ist.

Die gewonnenen Ergebnisse werden schließlich im letzten Kapitel (Kapitel 5) zusammengefasst und es werden Verbesserungsmöglichkeiten beziehungsweise Anknüpfungspunkte an diese Masterarbeit aufgezeigt.

2. WLAN gemäß IEEE 802.11

Die Drahtlostechnologie IEEE 802.11 ist als Teil einer ganzen Technologiefamilie, namentlich IEEE 802, anzusehen. Somit beinhaltet IEEE 802.11 Anforderungen, welche ihn als Mitglied dieser Familie auszeichnen (beispielsweise das Verhalten gegenüber höheren Schichten), aber auch Eigenheiten (beispielsweise die verwendeten Protokolle), welche ihn als einzigartiges Familienmitglied charakterisieren. Im nachfolgenden werden zunächst in Abschnitt 2.1 für das Verständnis wichtige Familieneigenschaften diskutiert, bevor die nachfolgenden Abschnitte sich ausschließlich mit den Eigenheiten von IEEE 802.11 beschäftigen.

2.1. IEEE 802

Die Standardfamilie 802 des IEEE-Verbandes beschreibt Regeln und Normen für Netzwerktechnologien. In Abbildung 2.1[1] wird ein Überblick über einige Standards dieser Familie gegeben. Für die Masterarbeit relevante Standards sind dabei fett hervorgehoben.

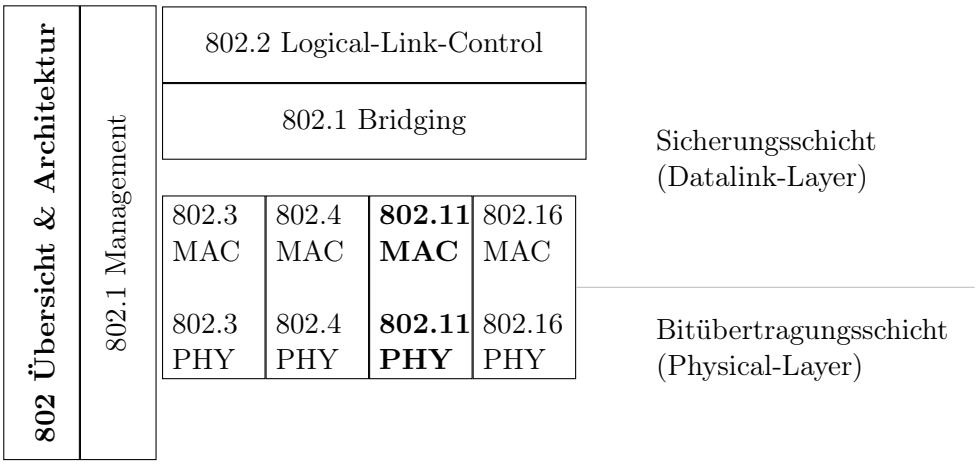


Abbildung 2.1.: Übersicht einiger IEEE 802 Standards

Der IEEE Std 802 gibt dabei einen allgemeinen Überblick über die Architektur. Dabei

[1] In Anlehnung an die Abbildung in [Std802, Seite IV]

wird verdeutlicht, dass sich der Standard ausschließlich auf Schicht Eins und Zwei des OSI-Referenzmodells (Abbildung2.2) bewegt.

Abbildung 2.2.: OSI-Referenzmodell

Desweiteren unterteilt dieser Standard die Sicherungsschicht in die Logical-Link-Control-Schicht (LLC) und die Medium-Access-Control-Schicht (MAC).

Erstere ist dabei im IEEE Std 802.2 beschrieben und spezifiziert eine einheitliche Schnittstelle für die darüberliegende Vermittlungsschicht. Auch die Schnittstelle zur MAC-Schicht wird hierin beschrieben.

Die Spezifikation der MAC-Schicht ist von der verwendeten Netzwerktechnologie abhängig und daher auf mehrere Standards verteilt. So beschreibt der IEEE Std 802.3 die auf Carrier-Sense-Multiple-Access-with-Collision-Detection (CSMA/CD) beruhende Netzwerktechnologie – besser bekannt unter dem Namen Ethernet.

Neben der Beschreibung der MAC-Schicht wird auch die zugehörige Bitübertragungsschicht (PHY in Abb. 2.1) im jeweiligen Standard beschrieben. Beispielsweise können, für den dieser Arbeit zugrundeliegenden IEEE Std 802.11, die Spezifikationen für die verschiedenen Übertragungsgeschwindigkeiten dem Physical Layer zugeordnet werden (Tabelle 2.1).

Kapitel	Datenraten in Mbs^{-1}	Technologie	Frequenzbereich/ Spektrum
14	1/2	Frequenzsprungverfahren	2,4 GHz
15	1/2	Frequenzspreizverfahren	2,4 GHz
16	1/2	Infrarot	850 - 950 nm
17	1/5 - 54	Mehrträgerverfahren	5 GHz
18	1 - 11	Frequenzspreizverfahren	2,4 GHz
19	1 - 54	Frequenzspreiz- & Mehrträgerverfahren	2,4 GHz

Tabelle 2.1.: Übertragungsgeschwindigkeiten im IEEE Std 802.11-2007

Historisch gesehen waren die Spezifikationen der Kapitel 14 - 16 bereits Teil der ersten

Veröffentlichung von IEEE Std 802.11 im Jahre 1997. 1999 kamen dann die Kapitel 17 (IEEE Std 802.11a) und 18 (IEEE Std 802.11b), 2003 das Kapitel 19 (IEEE Std 802.11g) hinzu. Die Kleinbuchstaben zeigen an, dass es sich dabei um eine Ergänzung (Amendment) des aktuellen IEEE Std 802.11 handelt, welche erst in nachfolgenden Ausgaben eingearbeitet werden.

2.2. Drahtlosnetzwerkarchitektur

IEEE 802.11 liegt die Idee zu Grunde, dass zwei Rechner, mittels kabelloser Übertragungstechnik, Daten austauschen können. Im IEEE Std 802.11 werden diese Rechner als Stationen (STA) bezeichnet. In Abbildung 2.3 ist diese Grundidee dargestellt. Diese beiden Stationen bilden das kleinste drahtlose Netzwerk, welches im WLAN-Standard vorgesehen ist. Ein solcher Verbund wird unabhängig von seiner Teilnehmeranzahl auch Basic Service Set (BSS) genannt. Das räumliche Gebiet, innerhalb dessen Stationen dem BSS beitreten können, heißt Ausleuchtungszone und wird mittels eines durchgängigen Ovals gekennzeichnet.

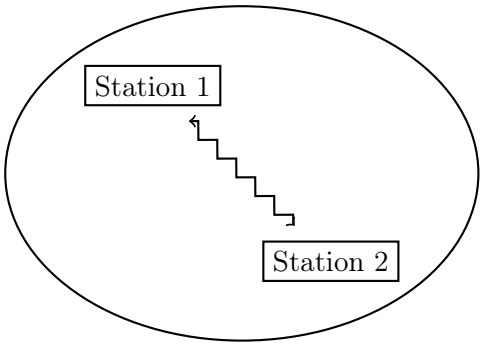


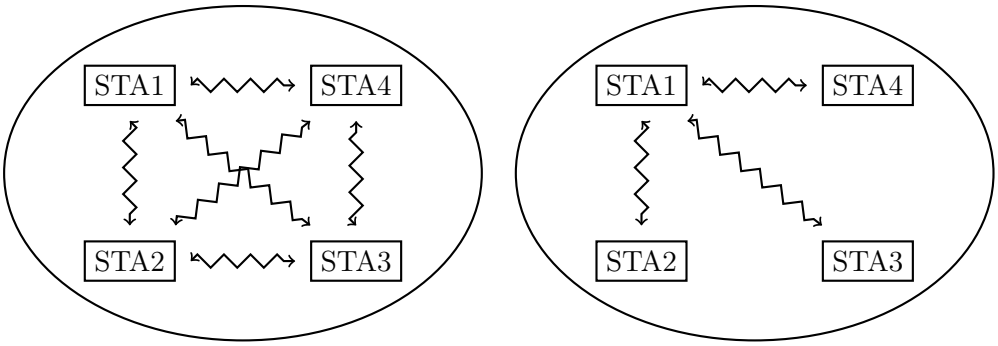
Abbildung 2.3.: Grundidee der Drahtloskommunikation

Abhängig vom Modus in dem eine Station betrieben wird, ergeben sich zwei grundlegend verschiedene, logische Netzwerktopologien für ein BSS. In Abbildung 2.4 werden im linken Netzwerk alle Stationen im Ad-hoc-Modus betrieben, wodurch jede Station mit jeder Anderen direkt kommunizieren kann. Es ergibt sich somit ein vermaschtes Netz. Der Vorteil dieser Netzwerkform liegt besonders darin, ein Drahtlosnetzwerk nur dann aufzubauen, wenn es benötigt wird. Man bezeichnet es deswegen auch als Ad-hoc-Netzwerk. Ein Nachteil ist allerdings die aufwändige Kontrolle von Zugangsberechtigungen, welche auf jeder Station separat verwaltet werden müsste.

Im Gegensatz zum Ad-hoc-Netzwerk können die Stationen im rechten Netzwerk nur mit Hilfe der Kontrollstation in Verbindung treten. Dabei wird die Kontrollstation STA1 im Zugangskontrollmodus (Access-Control-Mode[2]) und die restlichen Stationen im Clientmodus betrieben. Dies entspricht somit einer Sterntopologie und wird von IEEE Std

[2] [Std80211, Anhang N.2 „Terminology", Seite 1169]

802.11 als Infrastrukturnetzwerk bezeichnet. Der herausragende Vorteil hierbei ist die zentrale Konfigurierbarkeit mittels der Kontrollstation. Diese bestimmt, welche Station dem Netzwerk beitreten kann und welche Voraussetzungen diese dafür erfüllen muss. Ein wesentlicher Nachteil ist die erhöhte Latenz bei der Kommunikation zwischen Clientstationen.



Vermaschtes Netz Sterntopologie

Abbildung 2.4.: Logische Netzwerktopologien

Da das vermaschte Netzwerk keine spezielle Kontrollstation benötigt, wird es auch als unabhängiges Netz bezeichnet. Der IEEE Std 802.11 bezeichnet es als Independent-BSS (IBSS). Allerdings ist das vermaschte Netz kein vollständig vermaschtes Netz, wie es in der Abbildung 2.4 erscheint. Dies wird deutlich, wenn sich die Anordnungen der einzelnen Stationen zu einander, entsprechend der Abbildung 2.5, verändern. Dies ist ohne weiteres möglich, denn schließlich handelt es sich hier um räumlich bewegliche Rechner (zum Beispiel: Notebooks). Aufgrund von Reichweitenbegrenzungen[3] der drahtlosen Kommunikation kann in diesem Fall STA1 nicht mit STA4 Daten austauschen, obwohl sie Teil des Netzwerkes sind. Dabei wird auch ersichtlich, dass die anderen Stationen keine Daten von STA1 an STA4 weiterleiten.

[3]Hindernisse, wie Wände oder individuelle Übertragungsleistungen der Stationen, beeinflussen die Reichweite

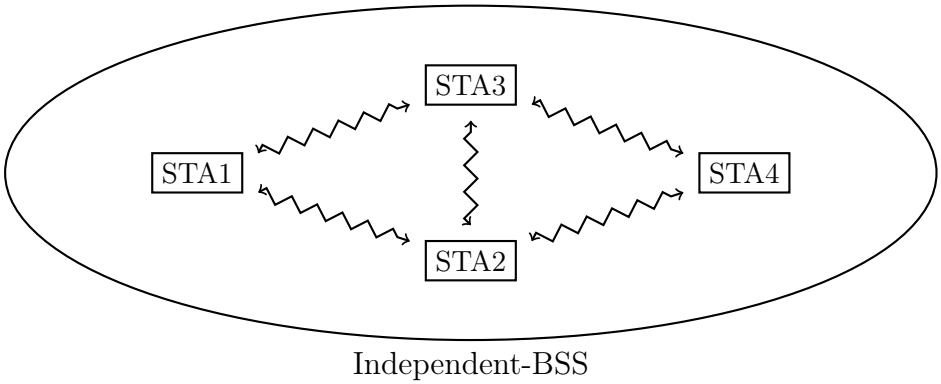


Abbildung 2.5.: Reichweitenbegrenzung im unabhängigen Netzwerk (IBSS)

Dieses Reichweitenproblem kennt dass zentral-verwaltete BSS (Infrastrukturnetzwerk) nicht. Da hier eine entfernt-liegende Station keinen Kontakt zur Kontrollstation aufnehmen kann, ist ihre Mitgliedschaft in diesen Netzwerk schlichtweg ausgeschlossen. Für diesen Fall sieht IEEE 802.11 einen Zusammenschluss mehrerer zentral verwalteter-BSS vor. Das so entstehende Netzwerk weist eine logische Baumtopologie entsprechend Abbildung 2.6 auf und wird erweitertes Infrastrukturnetzwerk (Extended-Service-Set – ESS) genannt.

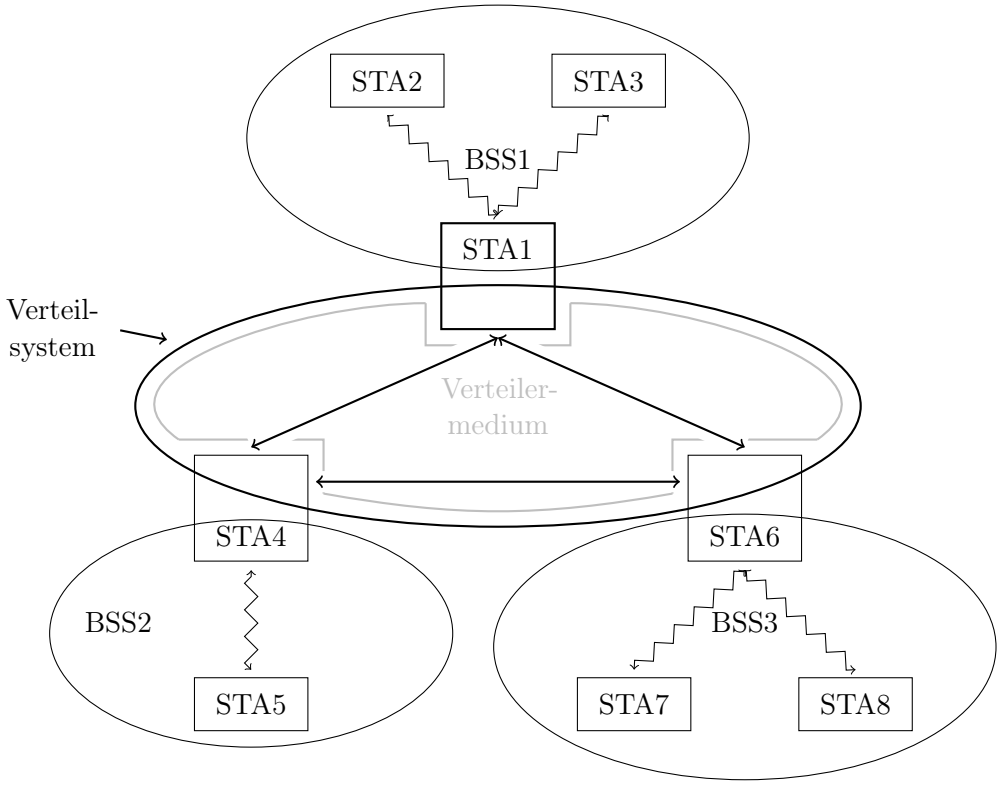


Abbildung 2.6.: Reichweitenerhöhung und Baumtopologie

In einem ESS werden die einzelnen Kontrollstationen um die Fähigkeit erweitert, mit anderen Kontrollstationen in Verbindung zu treten. Über welches Medium diese Verbindung technisch realisiert wird, lässt der Standard offen. Er spricht hier allgemein von dem Verteilermedium (Distribution-System-Medium – DSM). Die Gesamtheit von Kontrollstationen und Verteilermedium wird schließlich als Verteilsystem (Distribution-System – DS) bezeichnet. Die erweiterte Bedeutung der Kontrollstationen wird schlussendlich durch die Bezeichnung als Access-Point (AP) hervorgehoben.

Mittels des DS ist es nun für STA7 aus BSS3 möglich mit STA5 aus BSS2 zu kommunizieren. In diesem Zusammenhang sei am Rande noch erwähnt, dass IEEE 802.11 (innerhalb eines ESS') den Übergang von Mitglieds-BSS zu Mitglieds-BSS unterstützt (Roaming), falls deren Ausleuchtungszonen sich überlappen (in Abbildung 2.6 wäre dies also nicht möglich).

Von den vorgestellten drei Netzwerkarchitekturen (IBSS, Infrastrukturnetzwerk und ESS) wird sich die Masterarbeit auf das Infrastrukturnetzwerk fokussieren.

2.3. Architektur einer Station

Die Architektur einer Drahtlosstation ist, unabhängig von ihrem Modus, stets in die beiden Schichten MAC und PHY aufgeteilt. In Abbildung 2.7 ist eine weitere Unterteilung der PHY-Schicht in die Physical-Layer-Convergence-Procedure- (PLCP) und die Physical-Medium-Dependent-Schicht (PMD), zu erkennen[4]. Zu jeder Schicht gehört eine Managementkomponente[5], welche im wesentlichen die Aufgabe hat, Zugriff auf die Stellschrauben der jeweiligen Schicht zu bieten und MAC-interne Zustände vorzuhalten. Beide Komponenten werden im Abschnitt 2.6 zusammen mit der Station-Management-Entity näher behandelt.

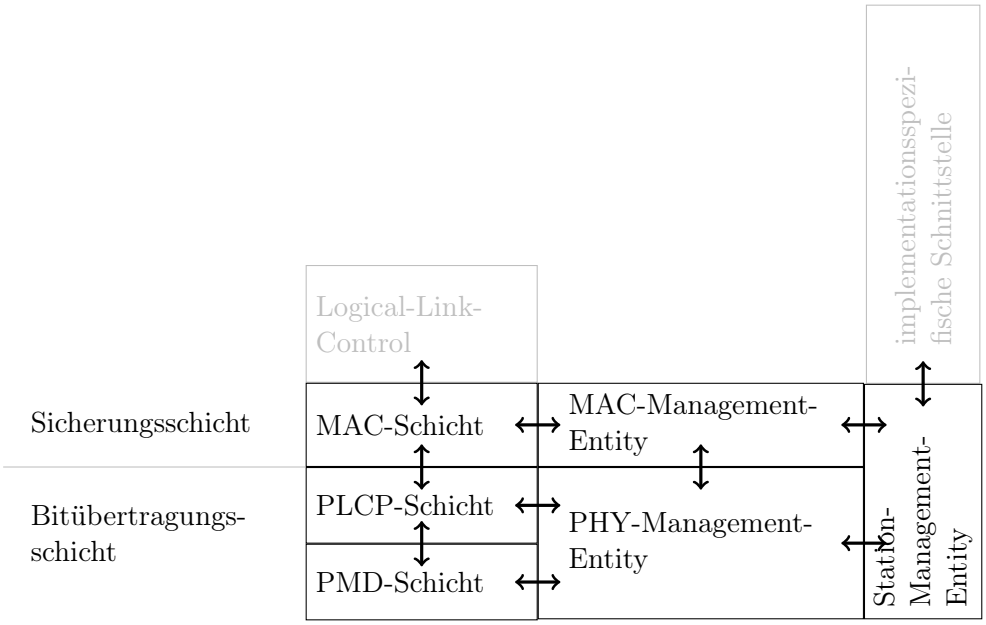


Abbildung 2.7.: Architektur einer Station entsprechend [Std80211, Abbildung 10-1 „GET and SET operations", Seite 313]

Die vertikal durchgängig dargestellte Station-Management-Entity (SME) ist der zentrale Zugangspunkt zu den Managementkomponenten. Die SME selbst ist weder im IEEE 802.11 noch in anderen Standards spezifiziert.

2.4. MAC-Architektur

Die Architektur der MAC-Schicht ist abhängig vom Modus (Abschnitt 2.2) in dem sich die Station befindet. Nachfolgend, wie auch in Abbildung 2.8, wird ausschließlich auf den

[4] Abschnitt 2.5
[5] MAC-Layer-Management-Entity – MLME und PHY-Layer-Management-Entity – PLME

Clientmodus Rücksicht genommen[6]. Die im Abschnitt 2.7 vorgestellten Standarderweiterungen sind ebenfalls außen vorgelassen.

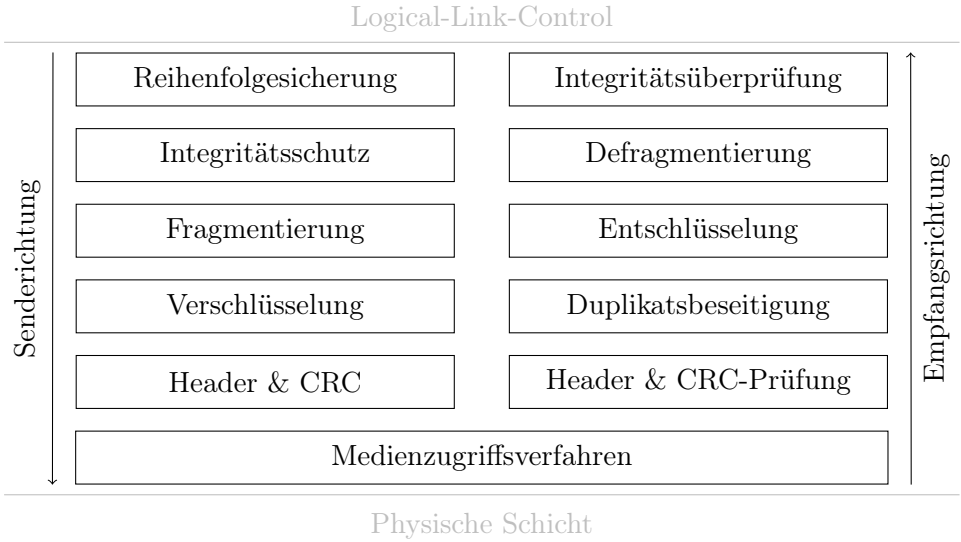


Abbildung 2.8.: Clientmodus MAC-Architektur

2.4.1. Frames

Das Verpacken der Nutzdaten aus der LLC-Schicht in einen MAC-Frame ist eine der zentralen Aufgaben der MAC-Schicht. Diese Aufgabe untergliedert sich in mehrere Teilschritte, welche bereits in Abbildung 2.8 ersichtlich sind und in den nachfolgenden Abschnitten näher betrachtet werden.

2.4.1.1. Basisformat

Allen Frames liegt eine gemeinsame Struktur zu Grunde. Abbildung 2.9 weicht dabei von dem in IEEE Std 802.11 dargestellten Aufbau[7] in drei Punkten ab, da diese für die Masterarbeit unbedeutend sind:

1. kein doppeldeutiges Dauer-/ID-Feld[8]
2. kein Adresse-4-Feld[9]

[6] Eine komplette Übersicht findet sich in [Std80211, 6.5.1 „MAC data Service architecture", Seite 54 ff. und 9.1 „MAC architecture", Seite 241 ff.]

[7] [Std80211, 7.1.2 „General frame format", Seite 60]

[8] Als ID-Feld wird es nur von Stationen im Energiesparmodus verwendet, welche einen für sie gespeicherten Frame abholen wollen. [Std80211, 7.1.3.2 „Duration/ID field", Seite 64]

[9] Wird nur für Inter-Access-Point-Kommunikation benötigt. Auch bekannt als Wireless-Distribution-System (WDS). Das Adresse-4-Feld beinhaltet in diesen Fällen die MAC-Adresse der Initiatorstation [Std80211, Tabelle 7-7 „Address field contents", Seite 77 und 7.1.3.3.7 „SA field", Seite 66].

3. kein QoS-Steuerungsfeld[10]

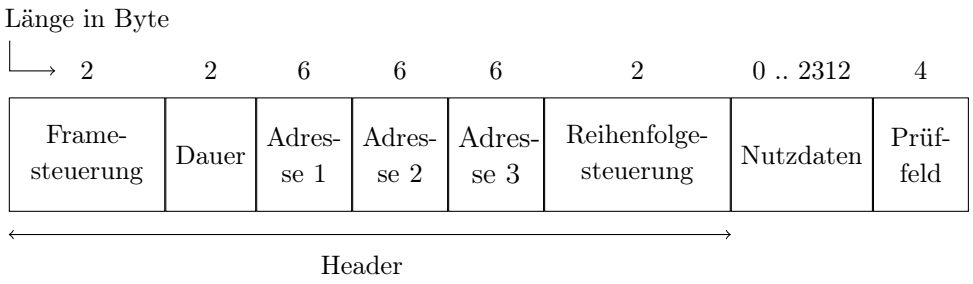


Abbildung 2.9.: Basisframeformat

Die einzelnen Felder sind teilweise nochmals unterteilt. Für eine detailliertere Beschreibung sei hier auf die Literatur[11] verwiesen. Im Folgenden werden nur ihre, für das weitere Verständnis wichtigen, Aufgaben beschrieben:

Framesteuerungsfeld (Frame-control). Enthält alle Informationen, um das Frameformat korrekt interpretieren zu können. So legt es beispielsweise fest, um welchen Frametyp (Abschnitte 2.4.1.2, 2.4.1.3 und 2.4.1.4) es sich handelt und wie die einzelnen Adressfelder zu interpretieren sind.

Dauer (Duration). Zeitdauer (µs) für die das Medium während des Sendevorgangs belegt sein wird.

Adressen IEEE-802-MAC-Adressen zur Identifikation des Senders[12], sowie zur Adressierung des Empfängers[13] und des Drahtlosnetzwerkes[14].

Reihenfolgesteuerung (Sequence-control). Fortlaufend vergebene Nummer, um mehrfach empfangene Frames zu erkennen und Teilframes (Fragmente) wieder zusammenzuführen.

Nutzdaten (Data). Bytefolge, die mit der nächst höheren Schicht (Abbildung 2.7 auf Seite 19) ausgetauscht wird.

Prüffeld (Frame-Check-Sequence-field). Prüfsumme[15] zum Erkennen fehlerhafter Übertragungen. Header und Nutzdaten fließen in ihre Berechnung ein.

[10] Wird nur von Stationen erzeugt/benötigt, welche die QoS-Erweiterung unterstützen [Std80211, 7.1.3.5 „QoS Control field", Seite 67].
[11] [Std80211, 7.1.3 „Frame fields", Seite 60 ff.] und [WiFi, 4.5 „802.11-Frameformat", Seite 179 ff.]
[12] Source-Address (SA)
[13] Destination-Address (DA)
[14] BSSID
[15] Cyclic-Redundancy-Check (CRC); [Std80211, 7.1.3.7 „FCS field", Seite 70]

2.4.1.2. Managementframes

Managementframes haben den Zweck, die Verwaltungsaufgaben der jeweiligen Station zu unterstützen. Das Auffinden von Drahtlosnetzwerken, der Beitritt und auch die Abmeldung sind jeweils über den Austausch und das Auswerten von Managementframes realisiert.

Der IEEE Std 802.11 kennt 12 verschiedene Managementframes[16]. In Tabelle 2.2 sind die, für eine Clientstation bedeutenden, Frames ihrer Aufgabe entsprechend eingeteilt.

Auffinden	Beitritt	Abmeldung	Statusinformationen
Beacon	Authentication	Deauthentication	Action
Probe-request	Association-request	Disassociation	
Probe-response	Association-response		
	Reassociation-request		
	Reassociation-response		

Tabelle 2.2.: Managementframes nach Aufgabe gruppiert

Die Verwendung der einzelnen Managementframes innerhalb eines Drahtlosnetzwerkes ist für das Verständnis der Masterarbeit irrelevant.

2.4.1.3. Datenframes

Der Zweck von Datenframes ist der Transport der Daten der höheren Netzwerkschichten. Dabei sind 15, in Tabelle 2.3 gelisteten, Frameformate[17] definiert. Von denen ist nur das, von der Distribution-Coordination-Function (Abschnitt 2.4.7.1) verwendete, Data-Frameformat für den WLAN-Stack-Entwurf notwendig.

Die Einteilung der Datenframes erfolgt nach Medienzugriffsverfahren, welche übersichtsweise in Abschnitt 2.4.7 auf Seite 26 vorgestellt werden.

Die anderen Formate sind zum Einen der Quality-of-Service-Erweiterung (Abschnitt 2.7.2) und zum Anderen dem (optionalen) wettbewerbsfreien (Contention-Free – CF) Medienzugriffsverfahren geschuldet (Abschnitt 2.4.7.2).

[16] [Std80211, 7.2.3 „Management frames", Seite 79 ff.]
[17] [Std80211, Tabelle 7-1 „Valid type and subtype combinations (continued)", Seite 62]

Dezentral	Contention-Free	Quality-of-Service
Data	Data+CF-Ack	QoS-Data
	Data+CF-Poll	QoS-Data+CF-Ack
	Data+CF-Ack+CF-Poll	QoS-Data+CF-Poll
	Null	QoS-Data+CF-Ack+CF-Poll
	CF-Ack	QoS-Null
	CF-Poll	QoS-CF-Poll
	CF-Ack+CF-Poll	QoS-CF-Ack+CF-Poll

Tabelle 2.3.: Einteilung der Datenframeformate

2.4.1.4. Kontrollframeformat

Kontrollframes werden vom jeweiligen Medienzugriffsverfahren (Abschnitt 2.4.7) für dreierlei Aufgaben eingesetzt.

1. Zur aktiven **Mediumreservierung**, damit nur Stationen auf das Medium zugreifen, die am bevorstehenden Austausch eines Management- beziehungsweise Datenframes beteiligt sind.
2. Zur aktiven **Empfangsbestätigung**, damit der Sender vom Empfänger über den erfolgreichen Eingang des Management-/Datenframes informiert werden kann.
3. Zur **Energiemanagementunterstützung**, damit im Energiesparmodus-befindliche Stationen für sie gepufferte Management-/Datenframes anfordern können.

Da die Generierung und Verarbeitung von Kontrollframes komplett von der Hardware übernomen wird, wird das Thema Kontrollframes in der Masterarbeit nicht weiter vertieft.

2.4.2. Duplikatsabwehr

Eine Technik, die das Erkennen und Aussortieren von mehrfach gesendeten Frames ermöglicht. Auf der Senderseite werden dafür ganzzahlige Nummern in lückenfreier, aufsteigender Reihenfolge aus dem Wertebereich [0, 4095] vergeben. Ist der höchste Wert erreicht, wird wieder bei Null begonnen. Die ermittelte Nummer wird letztendlich in einem 12 Bit breiten Bereich des Reihenfolgesteuerungsfeldes (Abschnitt 2.4.1.1) eingetragen.

Da nun jeder neue Frame eine eindeutige Nummer hat, kann die Empfängerseite erkennen, wann sie ein und denselben Frame mehrfach empfangen hat. Mehrfachaussendungen entstehen, sobald der Sender keine Empfangsbestätigung erhält und den vorhergehenden Frame erneut sendet.

2.4.3. Integritätsschutz

Der Integritätsschutz soll im Wesentlichen das Erkennen von Man-in-the-middle-Attacken ermöglichen. Er wurde, als Reaktion auf die unsicher gewordene[18] WEP[19]-Verschlüsselung, erst nachträglich zu IEEE 802.11[20] hinzugefügt.

Der Integritätsschutz wird durch eine spezielle Prüfsumme – der Message-Integrity-Code (MIC) – realisiert. Dieser Code wird vom Sender an den Datenteil des Frames angefügt. In die Prüfsummenberechnung fließen unter anderem die Empfänger- und Senderadressen, sowie der Datenteil ein.

Der Empfänger überprüft die Integrität des Frames, nachdem er ihn vollständig empfangen und entschlüsselt hat. Dafür berechnet die Empfängerseite den MIC zunächst selbst. Anschließend vergleicht sie diesen mit dem empfangenen MIC. Bei Verschiedenheit wird die Stationenmanagementkomponente darüber informiert. Diese leitet dann geeignete[21] (zum Beispiel: Wechsel des Verschlüsselungscodes) Maßnahmen ein.

2.4.4. Fragmentierung

Die Fragmentierung ist eine Technik, um wiederholtes Aussenden von fehlerhaft übertragenen Frames zu vermeiden. Dabei wird der Umstand ausgenutzt, dass kleinere Frames zuverlässiger übermittelt werden können.

Der Sender splittet den Datenteil eines Frames in kleinere Segmente (Abbildung 2.10), wobei jedes dieser Fragmente die, zum Ursprungsframe gehörenden, Headerinformationen erbt.

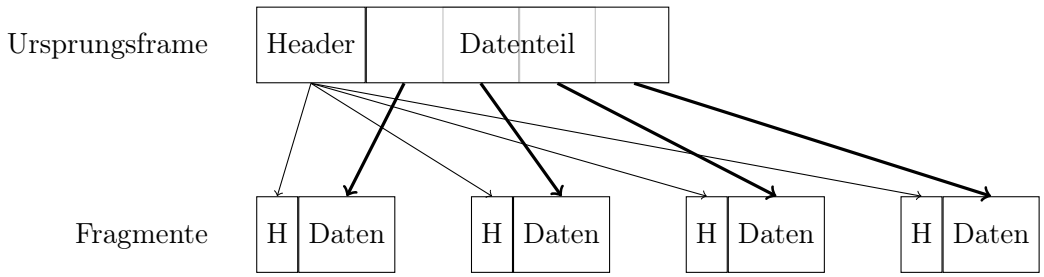


Abbildung 2.10.: Fragmentierung

Ob ein Frame fragmentiert wird oder nicht, legt ein MAC-interner Parameter fest. Der IEEE Std 802.11 definiert dabei eine Untergrenze von 256 Byte[22], unterhalb derer keine Fragmentierung stattfinden darf. Damit der Empfänger die Fragmente in der rich-

[18] [Std80211, 8.3.2.3 „TKIP MIC", Seite 169]
[19] Wired-Equivalent-Privacy, sinngemäß: Eine der Verkabelung vergleichbare Privatsphäre.
[20] Ammandment IEEE 802.11i. Kommerziell verwirklicht als Bestandteil von WPA/WPA2.
[21] [Std80211, 8.3.2.4 „TKIP countermeasures procedures", Seite 171 ff.]
[22] Header + Datenteil

tigen Reihenfolge wieder zusammensetzten kann, werden diese aufsteigend durchnummeriert. Diese Fragmentnummer wird ebenfalls im Reihenfolgesteuerungsfeld[23] abgelegt (Abschnitt 2.4.2).

Das Zusammensetzen der einzelnen Fragmente auf der Empfängerseite wird Defragmentierung genannt.

2.4.5. Frameverschlüsselung

Die Frameverschlüsselung trägt dem erhöhten Sicherheitsbedarf eines Drahtlosnetzwerkes Rechnung. Dabei definiert der IEEE Std 802.11 zur Absicherung des Datenverkehrs drei verschiedene Verschlüsselungsverfahren:

1. **Wired-Equivalent-Privacy** (WEP) ist das Verschlüsselungsverfahren der ersten Stunde und gilt mittlerweile als unsicher. Es wird nur noch aus Abwärtskompatibilitätsgründen im Standard aufgeführt.
2. **Temporal-Key-Integrity-Protocol** (TKIP) – kommerziell bekannter als WPA[24] – verwendet prinzipiell das gleiche Verschlüsselungsverfahren wie WEP, reichert die Frames jedoch um zusätzliche Sicherheitsmerkmale (siehe auch Integritätsschutz, Abschnitt 2.4.3, Stichwort MIC) an und wechselt die Schlüssel in regelmäßigen Zeitabständen.
3. **CTR-with-CBC-MAC-Protocol** (CCMP)[25] – kommerziell als WPA2 bekannt – ist eine Neuentwicklung, welche die Schwächen von WEP (und auch von TKIP) behebt. Dies wird vor allem durch den Einsatz des AES-Verschlüsselungstandards erreicht.

Für den Entwurf des WLAN-Stacks ist wichtig zu wissen, dass die Ver- und Entschlüsselung komplett von der Hardware übernommen wird. Somit können die Details der Verschlüsselung unberücksichtigt bleiben.

Jedoch könne sich Verschlüsselungsalgorithmen im Nachhinein als unsicher herausstellen (siehe WEP). Deswegen ist es weitsichtig, wenigstens eine Schnittstelle für das Anbinden von Softwareverschlüsselung vorzusehen.

Die Verschlüsselungsverfahren und ihre Schwächen werden sehr ausführlich[26] in [Attacks] diskutiert.

2.4.6. Frameprüfung

Das Ziel der Frameprüfung ist das Erkennen von Übertragungsfehlern. Dabei wird das, aus dem IEEE Std 802.3 bekannte, Prüfsummenverfahren (Abbildung 2.9, Prüffeld) ein-

[23] In einem 4 Bit breiten Teilfeld.
[24] Wi-Fi-Protected-Access
[25] Counter-mode-with-Cipher-block-chaining-Message-authentication-code-Protocol
[26] Beispielsweise werden Algorithmen zum Angriff auf WEP und TKIP vorgestellt.

gesetzt. Das Erzeugen und Auswerten der Prüfsumme wird von der Hardware übernommen. Daher kann dieser Vorgang vom WLAN-Stack ignoriert werden.

2.4.7. Medienzugriffsverfahren

Wie bei IEEE 802.3 auch, teilen sich in einem IEEE-802.11-Netzwerk mehrere Teilnehmer ein Medium. Senden mehrere Teilnehmer gleichzeitig, spricht man von einer Kollision. Diese können (im Unterschied zu IEEE 802.3) bei IEEE 802.11, aufgrund der physikalischen Eigenschaften des Mediums, nicht erkannt werden. Daher wird ein Verfahren eingesetzt, welches versucht Kollisionen von vornherein zu vermeiden. Dieses CSMA/-CA[27] genannte Verfahren, ist in die Hardware eingebaut und dadurch eine Blackbox für den WLAN-Stack.

Generell haben alle Stationen die gleiche Berechtigung auf das Medium zuzugreifen. Welche Station letztendlich senden darf, wird durch die nachfolgend vorgestellten Koordinierungsfunktionen geregelt. Diese unterscheiden sich dabei ausschließlich im verwendeten Übertragungsprotokoll (sprich: Welche Frametypen[28] wann und wie versendet werden.). Die Quality-of-Service-Erweiterung (IEEE 802.11e) definiert eine eigene Koordinierungsfunktion, welche allerdings im QoS-Abschnitt 2.7.2 behandelt wird.

2.4.7.1. Distribution-Coordination-Function

Dieses Übertragungsprotokoll erlaubt eine dezentral geregelte Kommunikation, wie sie zum Beispiel in unabhängigen Basic-Service-Sets (IBSS) vorkommt. Prinzipiell hat hier jede Station die gleiche Berechtigung bei der Bewerbung um den Medienzugriff. Damit möglichst nur eine Station auf das Medium zugreift, wartet jede Station eine zufällig gewählte Zeitdauer. Der Algorithmus hinter der Wartedauerauswahl wird als Backoff-Algorithmus bezeichnet und ist ebenfalls in Hardware realisiert.

2.4.7.2. Point-Coordination-Function

Die optionale Point-Coordination-Function kann nur in Infrastrukturnetzwerken (Stichwort: Access-Point, Abschnitt 2.2) verwendet werden. Sie sieht zwei Zeitphasen vor. Die Wettbewerbsphase (Contention-Phase – CP), entsprechend den Regeln der Distribution-Coordination-Function, und die wettbewerbsfreie Phase (Contention-Free – CF).

Während der wettbewerbsfreien Phase sendet nur jene Station, welche vorher vom Access-Point die Erlaubnis dazu erhalten hat.

[27] Carrier-Sense-Multiple-Access-with-Collision-Avoidance
[28] Abschnitt 2.3

2.5. PHY-Architektur

Der WLAN-Stack hat mit der Bitübertragungsschicht (Physical-Layer) keine Berührungspunkte. Der Vollständigkeit halber werden aber die Aufgaben von Physical-Layer-Convergence-Procedure und Physical-Medium-Dependent übersichtsweise wiedergegeben. Eine grafische Aufbereitung und Einordnung der PHY-Architektur findet sich in Abbildung 2.7.

Physical-Layer-Convergence-Procedure

Stellt für die MAC-Schicht eine einheitliche, von der verwendeten Übertragungstechnologie abstrahierende, Schnittstelle bereit. Hier werden die von der Koordinierungsfunktion übergebenen Frames mit einem technologiespezifischen Header versehen.

Physical-Medium-Dependent

In [WiFi, Seite 69] fasst Rech die Aufgabe dieser Schicht prägnant zusammen. Sinngemäß schreibt er, dass diese Schicht die Schnittstelle zur eigentlichen Sende-/Empfangseinheit darstellt.

2.6. Management-Architektur

Da einem Drahtlosnetzwerk der physische Netzwerkanschluss fehlt, muss IEEE 802.11 andere Methoden zur Netzwerkzuordnung verwenden. Erreicht wird dies durch die Verwendung der, in Abschnitt 2.4.1.2 vorgestellten, Managementframes. Diese stellen den Dreh- und Angelpunkt für die in Abbildung 2.11 gezeigten Managementkomponenten dar.

Der physische Netzwerkanschluss eines Ethernetnetzwerkes wird somit durch einen logischen Anschluss ersetzt. Dafür spielen die in Abbildung 2.11 dargestellten Komponenten Hand in Hand zusammen. Die Hände bilden dabei die Schnittstelle zwischen der Stationsmanagementeinheit [29] und der MAC-Management-Einheit[30], sowie zwischen MAC-Management und PHY-Management.

[29] Im IEEE Std 802.11 als Station-Management-Entitiy (SME) bezeichnet.
[30] IEEE Std 802.11: MAC-Layer-Management-Entity (MLME)

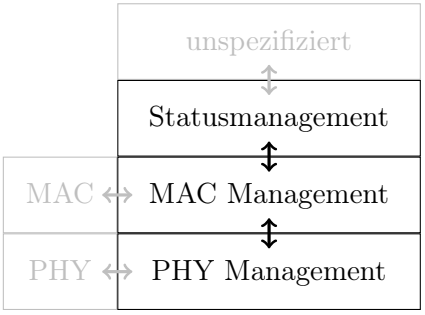


Abbildung 2.11.: Management-Architektur

Abweichend von der im IEEE Std 802.11 verwendeten Interaktionsdarstellung (Abbildung 2.7), besitzt die Stationsmanagementeinheit hier keinen direkten Zugriff auf die PHY Managementkomponente. Dies spiegelt eine, von IEEE 802.11 erlaubte, Entwurfsentscheidung wider, mit dem Ziel die Abhängigkeiten der Komponenten zu reduzieren (4.1).

Die grau dargestellten Interaktionspfeile erinnern daran, dass die Managementkomponenten nicht im luftleeren Raum stehen, sondern die MAC-/PHY-Komponenten beeinflussen, beziehungsweise (im Falle der MAC-Managementeinheit) auch Frames darüber versenden/empfangen.

2.6.1. MAC-Management

Der IEEE Std 802.11 spezifiziert zwar die Schnittstellen und das Verhalten, aber nicht wie die MAC-Managementkomponente zu strukturieren ist. Jedoch gibt er eine Beispielimplementation[31] in Form von Zustandsmaschinen an, aus der die innere Architektur ersichtlich wird.

Die in Abbildung 2.12 dargestellte Architektur ordnet die 5 Managementkomponenten[32] einer Komponente für aktive Tätigkeiten[33] (Management-Dienste) und einer für passive Tätigkeiten[34] (Management-Datenhaltung) zu. Die Informationselemente der Datenhaltung werden im Anhang D[35] von IEEE Std 802.11 spezifiziert.

[31] [Std80211, C.3 „State machines for MAC STAs", Seite 871 ff. und 878 ff.]
[32] MIB, Mlme_Requests, Mlme_Indications, Mlme_Sta_Services, Power_Save_Monitor
[33] Mlme_Requests, Mlme_Indications, Mlme_Sta_Services, Power_Save_Monitor
[34] MIB (Management-Information-Base)
[35] [Std80211, „MAC Attribute Templates", Seite 1008 bis Seite 1026 „End of dot11QosCounters TABLE"]

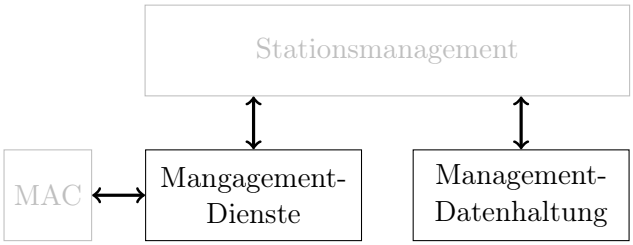


Abbildung 2.12.: MAC-Management-Architektur entsprechend der Beispielimplementierung des IEEE Std 802.11

Der WLAN-Stack orientiert[36] sich an dieser Zweikomponentenarchitektur, um damit seine Verständlichkeit[37] zu erhöhen.

2.6.2. PHY-Management

Das PHY-Management besteht ausschließlich aus einer Komponente. Der IEEE Std 802.11 beschreibt ihre Aufgabe (Kapitel 13 „PHY Management") mit der simplen Aussage, dass diese Komponente all die Daten bereitstellt, welche zum Verwalten einer Station notwendig sind. Die Informationselemente werden im Anhang D ([Std80211, „PHY Attribute Templates", Seite 1028 bis Seite 1050 „End of dot11PhyERP TABLE"]) von IEEE Std 802.11 spezifiziert.

Wie in Abbildung 2.13 zu sehen, wird die PHY-Management-Datenhaltung an die MAC-Management-Datenhaltung angeschlossen, wodurch der in Abschnitt 2.6 präsentierte Ansatz der Abhängigkeitenreduktion eingehalten wird.

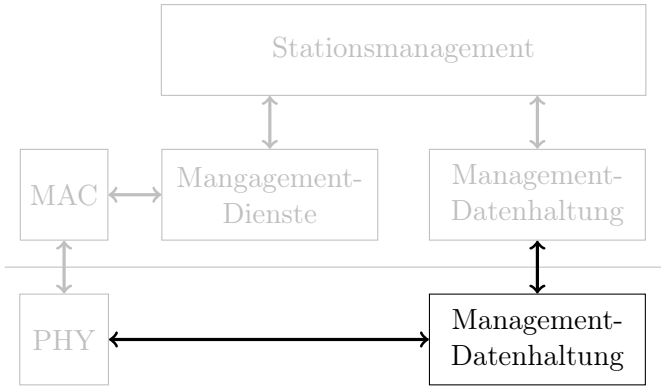


Abbildung 2.13.: Anbindung der PHY-Management-Datenhaltung an MAC-Managementschicht

[36] Anmerkung: orientieren ≠ kopieren
[37] Im Sinne von: Andere Anwender können sich leichter im WLAN-Stack zurechtfinden.

2.6.3. Stations-Management

Die Architektur der Stations-Management-Einheit (SME) ist kein Bestandteil von IEEE Std 802.11, weswegen hier keine allgemeine graphische Darstellung gegeben wird. Stattdessen kann eine mögliche Strukturierung dem Klassendiagramm (Abschnitt 4.4.2) des WLAN-Stacks entnommen werden.

Die Aufgaben der SME lassen sich allerdings schon aus dem IEEE Std 802.11 ableiten. Im Grunde bestimmt die SME das Verhalten einer Station gegenüber dem Betriebssystem sowie innerhalb eines Drahtlosnetzwerkes. Dazu gehören das Verwalten von Netzwerken[38], des Energiesparmodus' sowie der Mobilität innerhalb eines erweiterten Infrastrukturnetzwerkes (Roaming).

Da die SME entscheidet, welchem Netzwerktyp[39] eine Station beitreten kann, entscheidet sie letztendlich auch, in welchem Betriebsmodus[40] eine Station betrieben wird. Abhängig vom Betriebsmodus fallen die obengenannten SME-Aufgaben unterschiedlich aus. Tabelle 2.4 gibt darüber einen Überblick.

Diese Tabelle führt auch das ordentliche Beenden eines Drahtlosnetzwerkes auf, obwohl in IEEE Std 802.11 nirgends davon die Rede ist. Stattdessen sieht der Standard vor, dass jede Station mit dem plötzlichen Verschwinden des Netzwerkes zu rechnen und anstandslos damit zurechtzukommen hat (beispielsweise durch das Verwenden von Timeouts). Ein ordnungsgemäßes Beenden lässt sich jedoch indirekt aus den Beitritts- und Austrittsregeln herleiten.

[38] Verbindungsaufbau/-abbau, Bekanntgabe des erzeugten Netzwerks (Beacon)
[39] unabhängiges Netzwerk (IBSS) oder Infrastrukturnetzwerk
[40] Ad-hoc-, Client-, Zugriffskontrollmodus

Modus → Aufgabe ↓	Client	Zugriffskontrolle	Ad-hoc
Netzwerk starten	Nicht möglich.	Nur einfaches Infrastrukturnetzwerk, kein erweitertes Infrastrukturnetzwerk.	Nur unabhängiges Drahtlosnetzwerk.
Bekanntgabe	Nicht möglich.	Regelmäßig und nach Aufforderung.	siehe Zugriffskontrolle
Beitritt	Nur zu Infrastrukturnetzwerken. Schrittfolge: 1. Synchronisierung 2. Authentisierung 3. Assoziation	Verwaltet den Beitritt von Clientstationen.	Nur zu unabhängigen Netzwerken. Synchronisierung ausreichend.
Energiesparmodus	1. Ankündigung des Moduswechsels 2. Sendeeinstellung 3. Periodisches Überprüfen auf gepufferte Daten	Verwaltung der Energiesparzustände aller Teilnehmer und Pufferung von ausgehenden Daten. Benachrichtigung über gepufferte Daten.	Vereinigung der Aufgaben von Client- und Zugriffskontrollmodus.
Roaming	Nur bei erweiterten Infrastrukturnetzwerken (ESS): 1. Authentisierung 2. Reassoziation	Keine Mobilitätsverwaltung für Clientstationen.	Nicht möglich.
Austritt	1. Deassoziation 2. Deauthentisierung 3. Interaktion einstellen	Verwaltet den Austritt von Clientstationen	Interaktion einstellen. Keine Austrittsbekundung notwendig.
Ordentliches Netzwerk-beenden	Nicht möglich.	1. Verweigerung neuer Beitrittsersuche 2. Austrittsbenachrichtigung an alle Teilnehmer. 3. Einstellung der Bekanntgabe	Nicht möglich. Netzwerk existiert solange wenigstens eine Station noch die Aufgaben der Bekanntgabe wahrnimmt.

Tabelle 2.4.: Aufgaben der Stations-Management-Einheit abhängig vom Betriebsmodus

2.7. Standarderweiterungen

Der IEEE Std 802.11 ist in ständigem Fluss. In Form von Erweiterungen werden neue Funktionen hinzugefügt, alte Funktionen als hinfällig deklariert und Fehler ausgebessert. Die nachfolgenden Erweiterungen finden keine weitere Beachtung im WLAN-Stack-Entwurf, da Drahtlosnetzwerke nach wie vor auch ohne diese Erweiterungen aufgebaut werden können.

2.7.1. Sicherheit (802.11i)

Hauptschwerpunkt ist die Beseitigung der Schwachstellen des WEP-Sicherheitsprotokolls. Dabei liegen die maßgeblichen Neuerungen in der Verwendung des AES-Verschlüsselungsalgorithmus', der Erkennung von Man-in-the-Middle-Attacken und des Authentisierungsframeworks IEEE 802.1X.

2.7.2. Quality-of-Service (QoS) (802.11e)

Manche Daten sind wichtiger als andere. So lässt sich das Konzept hinter Quality-of-Service zusammen fassen. Mit dieser Erweiterung werden vier Prioritätsstufen eingeführt. Dadurch wird es möglich Audio-Daten aussetzerfrei zu übertragen, während gleichzeitig im Internet gesurft wird.

Übertragungsprotokoll

Quality-of-Service erforderte umfangreiche Anpassungen des Standards. Dies zeigt sich besonders im neu eingeführten, prioritätsfähigen Übertragungsprotokoll **Hybrid-Coordination-Function** (HCF). Dabei werden die Prioritäten im Wesentlichen durch das Einführen von dringlichkeitsabhängigen Wartezeiten realisiert: Wenn dass Medium frei ist, wird abhängig von der Framepriorität, eine zeitlang mit der Aussendung gewartet. Je höher die Priorität desto kürzer ist diese Wartezeit. Dringliche Daten werden also immer vor weniger dringlichen an das Medium übergeben.

2.7.3. Beachtung länderspezifischer Funkgesetze (802.11d)

Nicht alle der im ursprünglichen IEEE Std 802.11 spezifizierten Frequenzen können in jedem Land verwendet werden. Im Sinne der Autonomie regelt jedes Land die Frequenzvergabe selbst, wodurch beispielsweise in Japan 14 Kanäle in Europa aber "nur" 13 zur Verfügung stehen. Damit Notebooks aber auch länderübergreifend einem WLAN beitreten können, wurde die Erweiterung 802.11d erarbeitet. Dabei wird bei der regelmäßigen WLAN-Bekanntgabe eine Länderkennung mitgesendet. Jede 802.11d fähige Station kann sich damit automatisch auf die Regularien einstellen. Diese sind dabei in jeder einzelnen Station hinterlegt.

2.7.4. Frequenz- und Sendeleistungs-Management (802.11h)

In Europa muss das 5 GHz Band mit militärischen Radaranlagen geteilt werden. Damit ein gleichzeitiger Betrieb von WLAN und Radar möglich ist, wurde die 802.11h Erweiterung erarbeitet. Die Idee dabei ist „Erkenne und Vermeide" Radaranlagen. Mit anderen Worten: Die WLAN-Frequenz wird gewechselt, sobald die Aktivität einer Radaranlage festgestellt wird (Auch als dynamische Frequenzwahl bekannt).

Das Einhalten länderspezifischer Sendeleistungsvorgaben wird als Sendeleistungssteuerung (Transmit-Power-Control) bezeichnet und ist eine Erweiterung zu den mit 802.11d (Abschnitt 2.7.3) eingeführten Regulatory-Domains.

2.8. Zusammenfassung

In den vorhergehenden Abschnitten wurde ein Einführung in die Begrifflichkeit und die Funktionalitäten von IEEE 802.11 gegeben. Dabei wurden die Themenbereiche nur soweit vertieft, wie es für das Verständnis des WLAN-Stack-Entwurfs notwendig ist. Somit wurde der Schwerpunkt besonders auf die verschiedenen Architekturebenen und deren Komponenten gelegt. Im Sinne einer vollständigen Übersicht wurden auch Erweiterungen des IEEE Std 802.11 vorgestellt, welche im weiteren Verlauf keine Rolle mehr spielen.

3. Haiku

Das Betriebssystem Haiku hat sich als idealer Kandidat für den Entwurf und das Testen eines objektorientierten WLAN-Stacks angeboten. Sein Quellcode ist frei zugänglich, es verwendet von Haus aus objektorientierte Konzepte, es hat kurze Turn-Around-Zeiten, es läuft auf der Testhardware (Anhang A), es bietet einen integrierten Debugger und es unterstützt objektorientierte Programmierung im Kernelland. Der größte Nachteil von Haiku ist dabei jedoch die Abwesenheit eines WLAN-Stacks, wobei dies gleichzeitig auch motiviert, einen wertvollen Beitrag für das Haiku-Projekt zu leisten.

Der Nachteil konnte im Rahmen dieser Arbeit abgestellt werden, indem der – strukturorientierte – FreeBSD WLAN-Stack auf Haiku portiert wurde. Damit kann dieser – bereits bewährte – WLAN-Stack für das Testen des objektorientierten WLAN-Stack-Entwurfs unte Haiku herangezogen werden.

Da Haiku ein weitestgehend unbekanntes Betriebssystem[1] ist, wird in Abschnitt 3.1 zuerst eine geschichtliche Einordnung vorgenommen sowie ein Ausblick auf die weitere Entwicklung gegeben. Die oben erwähnten Vorteile von Haiku werden in Abschnitt 3.2 näher beleuchtet. Im Abschnitt 3.3 wird die Netzwerkarchitektur von Haiku vorgestellt, in die der objektorientierte WLAN-Stack letztendlich integriert werden soll. Die Beseitigung des ebenfalls oben angesprochenen Nachteils wird abschließend in Abschnitt 3.5 vorgestellt.

3.1. Geschichte und Zukunft

Begonnen hat für Haiku alles mit der Insolvenz von Be, dem Hersteller des Betriebssystems BeOS, im Jahre 2001. Das Haiku-Projekt, welches sich zunächst OpenBeOS nannte, wurde aus der Taufe gehoben, mit dem Ziel ein zu BeOS R5 binärkompatibles, quelloffenes Betriebssystem zu entwickeln.

Es gründeten sich zunächst mehrere Projekte zur Weiternutzung der BeOS-Hinterlassenschaften (Worunter hier auch die kommerzielle Weiterentwicklung Zeta gezählt wird), jedoch überlebte nur das Haiku-Projekt bis zum heutigen Tag. Diese Hinterlassenschaften beschreibt der Abschnitt 3.1.1 näher. Wie das Haiku-Projekt sich schrittweise an BeOS angenähert hat, erklärt im Anschluss der Abschnitt 3.1.2.

[1] Obwohl es bereits seit 2001 in der Entwicklung ist.

3.1.1. BeOS

Am 3. Oktober 1995 wurde BeOS das erste mal der Öffentlichkeit vorgestellt. Seit dem erschienen mehrere Versionen – Releases genannt – bis hin zur letzten Version – BeOS Release 5 (kurz: BeOS R5) –, welche im Jahr 2000 veröffentlicht wurde. Der kommerzielle Misserfolg von BeOS trug schließlich auch zur Insolvenz von Be Inc. im Jahre 2001 bei.

BeOS war auch bekannt als „das" Multimediabetriebssystem. Es bestach unter anderem durch seine hohe Reaktionsfähigkeit, sein datenbankähnliches Dateisystem und seine durchgehend objektorientierte Programmierschnittstelle. In dieser Zeit hießen die Wettbewerber (auf dem Markt der Heimanwender) Windows 98 und Mac OS 9 und die Rechner auf denen sie liefen besaßen Einkern-Prozessoren mit Taktfrequenzen im Bereich von 500 bis 1000 MHz.

Mehrere, gleichzeitig ablaufende Videos und Musikstücke (ohne Ruckler und Aussetzer) sowie Livequeries[2], waren beliebte Szenarien zur Unterstreichung von BeOS' Multimedialität. Videofenster ließen sich hin und herschieben, ohne – das von den Wettbewerben bekannte – Einfrieren der Videoanzeige. Trotzdem starteten Programme nahezu unmittelbar und das ganze System erweckte dabei den Eindruck von Reaktionsfreudigkeit.

Die technische Grundlage für diese Responsivität waren Multithreading, feingliedriger Einsatz von Synchronisierungsverfahren im Kernel und die Vergabe von höheren Prioritäten[3] an Vordergrundanwendungen als an Hintergrundanwendungen.

Livequeries rückten vor allem die Fähigkeiten des datenbankähnlichen Dateisystem – BFS genannt – in den Mittelpunkt. Gerne wurde hier nach E-Mails gesucht, welche von einem bestimmten Absender geschickt wurden oder welche den Status „Ungelesen" hatten. Eine Livequery nach allen neuen E-Mails wurde sofort bei Eintreffen von neuen E-Mails aktualisiert, wodurch vorallem das „Live" Betonung fand.

Durch umfassenden Einsatz von erweiterten Dateiattributen wurde die datenbankähnliche Verwendung des Dateisystems realisiert. Diese Funktionalität bietet die Möglichkeit einer Datei weitere Attribute zuzuordnen. Eine Musikdatei kann dann zum Beispiel zusätzlich die Attribute Interpret, Titel und Bewertung zugewiesen bekommen, unabhängig davon, ob das Dateiformat selbst solche Zusatzinformationen unterstützt. Dass Dateien auch anhand ihrer Attributinhalte gefunden werden können, ermöglicht schließlich umfangreiche Suchanfragen wie: „Finde alle Musikdateien deren Interpreten mit A beginnen und eine Bewertung von 3 Sternen oder besser haben".

Für die Entwicklung von Anwendungssoftware wurde ein übersichtliches Angebot an Schnittstellen bereitgestellt. Auf der einen Seite gab es das objektorientierte API[4], welche in sogenannte Kits[5] eingeteilt und primär zum Entwickeln nativer Anwendungen gedacht war. Auf der anderen Seite gab es die Schnittstelle, welche ein Untermenge der

[2] Marketingbegriff für Echtzeitsuche nach Daten, welche auf dem Dateisystem abgelegt sind.
[3] Bezeichnet als Desktoppriorität.
[4] Application Programming Interface
[5] Beispielsweise das Media-Kit, für Audio- und Videoverarbeitung, oder das Network-Kit für Netzwerkkommunikation.

POSIX-Funktionalität bereitstellte und somit das Portieren von BeOS-fremder Software erleichterte.

3.1.2. Haikus Weg zum Ziel

Wie lässt sich ein zu BeOS R5 binär- und quellcodekompatibles Betriebssystem entwickeln, welches das Weiterverwenden der bestehenden proprietären Applikationslandschaft sowie das einfache Neukompilieren von quelloffener Software ermöglicht? Diese Frage wurde vom OpenBeOS-Projekt nach seiner Gründung im Jahr 2001 mit einem inkrementellen Entwicklungsmodell beantwortet.

Dabei wurden proprietäre Systemkomponenten von BeOS Schritt für Schritt durch quelloffene, neuentwickelte Komponenten ersetzt. Dabei erforderten die beiden Ziele der Binär- und Quellcodekompatiblität die strikte Einhaltung der von BeOS R5 vorgegebenen öffentlichen Schnittstellen (API) sowie den Einsatz der GNU Kompilersammlung (GNU-Compiler-Collection – GCC) in der (altertümlichen) Version 2.95.3[6].

Bis zum Jahr 2004 ließen sich die Wurzeln von OpenBeOS deutlich im Namen ablesen, bevor dann die Umbenennung in Haiku erfolgte. Damit sollten potentielle Rechtsstreitigkeiten um die Namensgebung vermieden werden. Der neue Name Haiku wurde durch die Nutzer- und Entwicklergemeinschaft von OpenBeOS gewählt und bezieht sich auf die Fehlermeldungen des Webbrowsers NetPositive[7], welche in der japanischen Gedichtform Haiku verfasst waren.

Im Jahre 2008 konnte Haiku das erste mal innerhalb von Haiku kompiliert werden, es war von da an also selbsttragend (selfhosting). Nachdem die Entwicklung vorher maßgeblich unter BeOS beziehungsweise Linux und FreeBSD stattgefunden hatte, konnte seit dem auch in Haiku selbst entwickelt werden. Dies war ein wichtiger Schritt für die Stabilität von Haiku, da von nun an mehr und mehr Entwickler auch den Schritt zur Entwicklung innerhalb von Haiku gingen und so dem System (und seinen Fehlern) beständig ausgesetzt waren[8].

Der vorläufige Höhepunkte war die Veröffentlichung der ersten Alpha-Version von Haiku[9] am 14. September 2009. Diese wurde hauptsächlich als Möglichkeit gesehen, das Betriebssystem von einem möglichst breiten Publikum testen zu lassen, um die Stabilität weiter zu steigern. Diese Version wurde weit über 90.000 mal heruntergeladen[10].

Aktuell[11] wird die Veröffentlichung der zweiten Alpha vorbereitet beziehungsweise ist mit Fertigstellung dieser Masterarbeit bereits erschienen. Dabei ist geplant, den im Rahmen dieser Masterarbeit portierten FreeBSD WLAN Stack mit auszuliefern. Dadurch wird

[6] Aktuell ist Version 4.5, Stand 25. April 2010.
[7] Bestandteil von BeOS
[8] Sie fraßen also ihr eigenes Hundefutter[Showstop], um einen bekannten Spruch von David Cutler (Windows NT Chefentwickler) aufzugreifen.
[9] Haiku R1/Alpha1
[10] 95781 Stand: 03. März 2010
[11] März 2010

sich seine Stabilität weiter festigen, was seiner Rolle als Vergleichs- und Testbasis für den objektorientierten WLAN-Stack zuträglich ist.

3.2. Haiku als Entwicklungsumgebung

Haiku unterstützt objektorientierte Entwicklung, das wurde bereits in der Einleitung zu Kapitel 3 erwähnt. Nun gibt es jedoch mehrere Programmiersprachen, in denen objektorientierte Konzepte umgesetzt werden können. In Haiku ist dabei C++ die Sprache der Wahl. Bei der Implementierung des WLAN-Stacks gibt es dabei eine Besonderheit zu beachten, welche im folgenden vorgestellt wird.

Im Kernelland kann die Exceptionsfunkionaltiät von C++ nicht verwendet werden. Exceptions bieten, kurz gesagt, die Möglichkeit Fehlerbehandlungen explizit als solche zu kennzeichnen. Prinzipiell ließe sich diese Fähigkeit auch im Kernelland nutzen. Diese Restriktion ist jedoch bewusst von den Kernelentwicklern getroffen wurden, um den Kernel möglichst schlank zu halten.

Die Dokumentation für BeOS kann in weiten Teilen für Haiku weiterverwendet werden, was nützlich bei der Einarbeitung in das Betriebssystem ist. Besonders sei hier die „Be-Book“ ([BeBook]) genannte Dokumentation erwähnt, welche die gesamte öffentliche API von BeOS R5 (und damit auch Haiku) abdeckt und somit auch die Kernelschnittstellen beschreibt.

Haiku besitzt einen Kerneldebugger, der genau wie bei BeOS, im laufenden Betrieb aktiviert werden kann und damit ein Werkzeug bereitstellt, das alle Zustände des Systems anzeigen kann. Der Kerneldebugger wird entweder manuell per Tastaturkombination oder automatisch durch einen Kernelfehler aktiviert. In Abbildung 3.1 sieht man den Kerneldebugger in Aktion. Der weiß hinterlegte Abbildungsbereich ist dabei der interessante Bereich (die Kerneldebuggershell) in dem die Verwendung des Kerneldebuggerkommandos `bt` demonstriert wird. Dieses wird verwendet, um einen Backtrace des gerade ausgeführten Codestücks auszugeben.

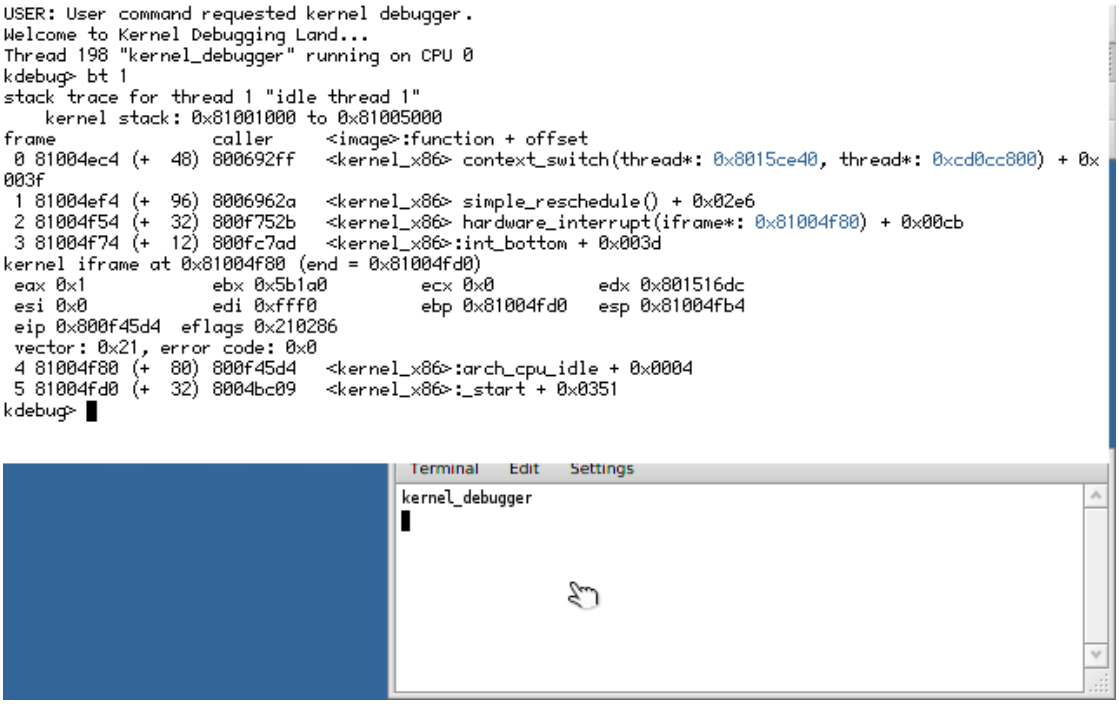


Abbildung 3.1.: Haikus Kerneldebugger in Aktion

Für das Debuggen von Userlandcode steht der GNU Debugger gdb zur Verfügung. Im Gegensatz zum Kerneldebugger bietet dieser zusätzliche Funktionalitäten, wie das quelltextgenau Zuordnen von aufgetretenen Fehlern und das schrittweise Abarbeiten des Quellcodes, hat aber auch eine geringere Systemdurchdringung (kein Zugriff auf das Kernelland möglich). Dennoch bietet gerade das quelltextgenaue Debuggen einen erheblichen Zeitgewinn gegenüber dem assemblergenauen Debuggen des Kerneldebuggers. Unter anderem deswegen bietet Haiku auch die Möglichkeit, den Netzwerkstack sowie Treiber im Userland ausführen zu können. Diese Option besteht dabei ausschließlich zum leichteren Testen und Debuggen[12] von Kernellandcode und wird nicht im Produktivbetrieb verwendet.

Ein Neustarten des Betriebssystems ist, gerade bei Arbeiten im Kernelland beheimateten WLAN-Stack, häufig notwendig. Dabei sind die kurzen Bootzeiten (von rund 20 Sekunden[13]) der Entwicklungsgeschwindigkeit zuträglich und halten die Turn-Around-Zeiten[14] gering.

[12] Beziehungsweise in den Anfangstagen von OpenBeOS/Haiku zur Unterstützung der inkrementellen Entwicklungsmethode (Abschnitt 3.1.2).
[13] Auf echter/emulierter (QEMU, VirtualBox, VMware) Hardware.
[14] Zeitspanne von neuerstelltem, ausführbaren Code bis zur Ausführung des neuen Codes.

3.3. Netzwerkarchitektur

Der objektorientierte WLAN-Stack fügt sich in die von Haiku bereitgestellte Netzwerkinfrastruktur ein. Diese Infrastruktur wird im Abschnitt 3.3.1 vorgestellt und damit die architektonische Einordnung des WLAN-Stacks in Kapitel 4 ermöglicht. In Abschnitt 3.3.2 werden die Techniken vorgestellte, welche verwendet wurden, um die Anwendungsfälle des Abschnitts 3.3.3 aus dem Netzwerkstack-Quellcode zu extrahieren. Abschließend (Abschnitt 3.3.3) werden drei wesentliche Anwendungsfälle von Haikus Netzwerkstack beschrieben. Diese gehen dabei sehr detailliert auf die Netzwerkarchitektur ein.

3.3.1. Aufbau

Haikus Netzwerkstack ist komplett im Kernelland realisiert, wobei Anwendungen im Userland über zwei Schnittstellen auf die Netzwerkfunktionalität zugreifen können. Abbildung 3.2 visualisiert diese grobe Einteilung. Diese grobe Schichtung wird in der nachfolgenden Diskussion weiter verfeinert, so dass das Verständnis für Haikus Netzwerkstack eine zunehmende Vertiefung erfährt.

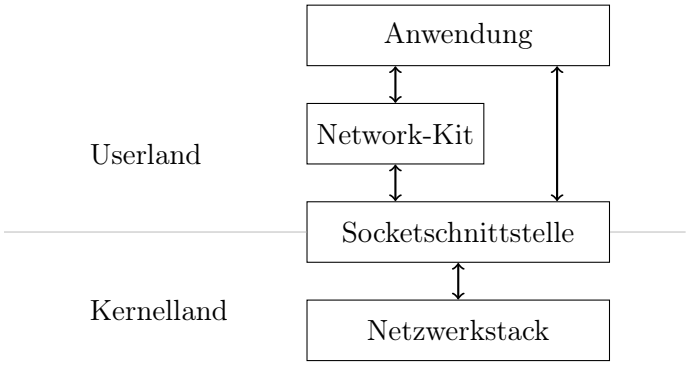


Abbildung 3.2.: Anwendungsschnittstellen des Netzwerkstacks und Grobeinteilung in User-/Kernelland

Die in der Abbildung 3.2 dargestellten Schnittstellen sind zum Einen die grundlegende, POSIX-konforme Socket-Schnittstelle und zum Anderen die darauf aufbauende, objektorientierte, Haiku-eigene Network-Kit-Schnittstelle. Das Network-Kit konzentriert sich dabei auf das Senden und Empfangen von Daten. Zusätzlich dazu werden Managementaufgaben (beispielsweise Zählen der erkannten Netzwerkgeräte) durch die Socket-Schnittstelle bereitgestellt.

Der im Kernelland angesiedelte Teil der Netzwerkinfrastruktur lässt sich grob in zwei weitere Schichten unterteilen, welche in Abbildung 3.3 dargestellt werden. Die gesamte Geschäftslogik[15] des Netzwerk-Stacks befindet sich dabei in der Steuerungsschicht. Die

[15] Netzwerkprotokolle, Benachrichtigungen, usw.

Anbindung der Netzwerkhardware und die dafür verwendeten Treiber sind Bestandteil der Treiberschicht. Die Treiberschnittstelle wird dabei durch, an POSIX-orientierte[16], Funktionsaufrufe definiert.

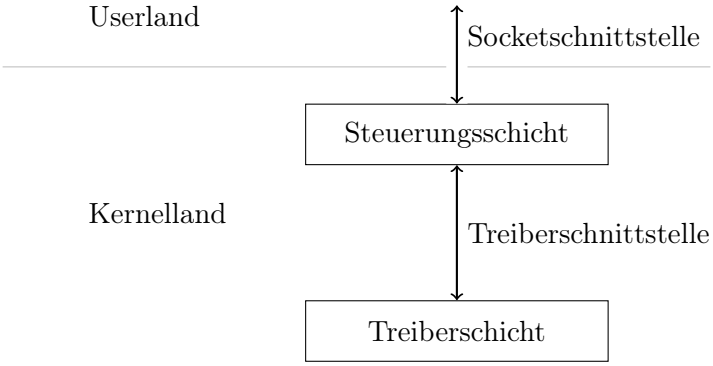


Abbildung 3.3.: Steuerungs- und Treiberschicht des Netzwerkstacks

Die detailliertere Darstellung der Protokollschicht in Abbildung 3.4 lehnt sich an die, unter [HaiNet] zu findende, Schichtung des Netzwerkstacks an. In der Protokollschicht sind dabei alle Netzwerkprotokolle[17] angesiedelt, welche unabhängig von der eingesetzten Netzwerktechnologie[18] sind. Dementsprechend beinhaltet die Datalink-Schicht netzwerktechnologieabhängige Protokolle[19]. Die Herstellung der Verbindung von Protokollen der Protokollschicht und der zu verwendeten Netzwerktechnologie ist ebenfalls Aufgabe der Datalink-Schicht. Die Geräteschicht stellt schließlich den Zugriff auf die eigentliche Netzwerktechnologie bereit. Dabei werden die einer Netzwerktechnologie gemeinsamen Funktionen (Geräteschnittstelle) in der Geräteschicht vereint, damit die eventuell[20] darunterliegenden Netzwerktreiber diese nicht jedesmal selbst implementieren müssen.

[16] Beispielsweise: read(), write(), ioctl(), ...
[17] Beispielsweise: TCP, UDP, IP
[18] Ethernet, Loopback, Bluetooth, ...
[19] Beispielsweise: ARP, Ethernetframing
[20] Das Loopback-Gerät benötigt keinen Treiber.

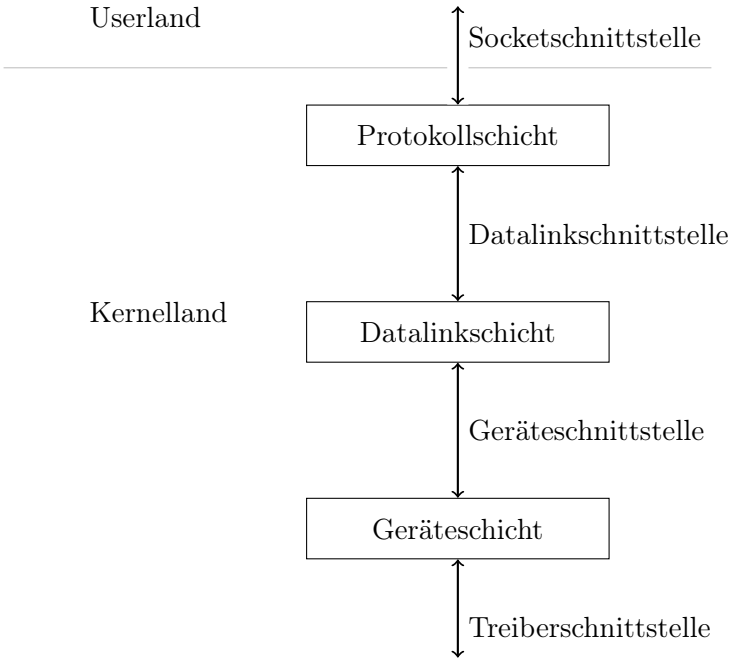


Abbildung 3.4.: Aufbau der Steuerungsschicht

Die Gesamtarchitektur von Haikus Netzwerkstack ist abschließend in Abbildung 3.5 dargestellt. Dabei werden der jeweiligen Schicht beispielhaft einige Vertreter zugeordnet. So sind zum Beispiel WebPositive[21] und ifconfig[22] Vertreter der Anwendungsschicht. TCP und IP – als Vertreter der Protokollschicht – seien hier als Beispiel für objektorientiert implementierte Netzwerkprotokolle nochmals extra erwähnt.

[21] Haikus Webbrowser
[22] Kommandozeilenprogramm zur Netzwerkkonfiguration

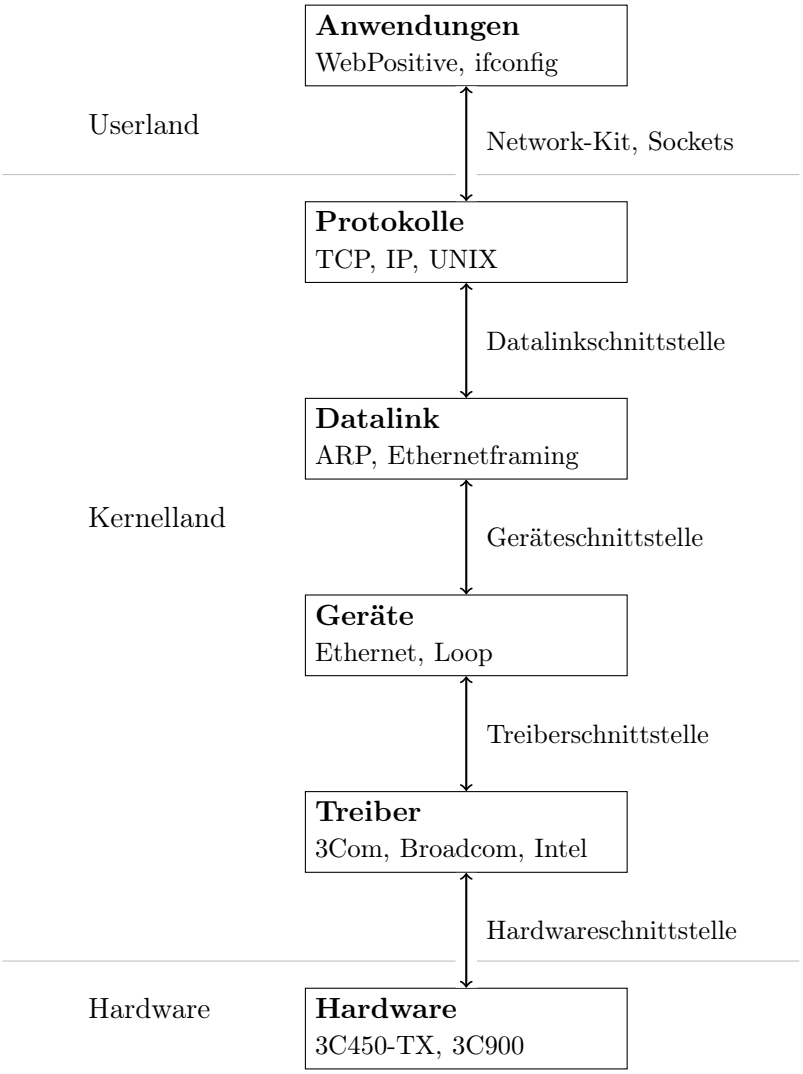


Abbildung 3.5.: Gesamtarchitektur von Haikus Netzwerkstack

3.3.2. Quellcodestudium

Literatur über die Funktionsweise des Netzwerkstacks ist spärlich gesät. Es steht an Informationsquellen hauptsächlich[23] die vom Haiku-Projekt bereitgestellte Dokumentation und der Quellcode selbst bereit. Allerdings ist die Aktualität der Dokumentation stets mit Vorsicht zu genießen, da – wie in jedem anderen Opensourceprojekt auch – der Code sich schneller ändert als die Dokumentation angepasst wird. Dennoch stellt die Dokumentation zum Networkstack einen guten Überblick über die Architektur (Abschnitt 3.3.1) und die wesentlichen Datenstrukturen dar.

[23] Mailinglisten sind außenvorgelassen, da diese im wesentlichen dem Klären spezieller Fragen dienen.

3.3.2.1. Analysetechniken

Für das Verstehen der Funktionsweise und des Zusammenspiels von Komponenten ist es notwendig, diese in Aktion zu sehen. Es ist also unumgänglich, den relevanten Quellcode auszuführen. Dabei kann diese Ausführung entweder direkt auf dem Rechner oder virtuell im Kopf erfolgen.

Rechnerausführung

Für die Beobachtung von direkt ausgeführtem Quellcode stehen prinzipiell zwei Techniken zur Verfügung:

1. Echtzeitbeobachtung
2. Auswertung einer aufgezeichneten Ausführung

Die Echtzeitbeobachtung erfolgt dabei mittels eines Debuggers, wobei in Haiku der GNU Debugger gdb – wie in der Opensourcewelt üblich – eingesetzt werden kann. Der Debugger wird hierbei verwendet, um Schritt für Schritt – stepping genannt – den Quellcode zu durchwandern, während dieser auf dem Rechner ausgeführt wird. Diese Technik lässt sich allerdings nur dann anwenden, wenn Zeit eine untergeordnete Rolle spielt, denn das Programm wird nach jedem Schritt vom Debugger angehalten. Somit ist diese Technik für das Verständnis des Netzwerkstacks ungeeignet, da hier beispielsweise abgelaufene Timeouts beim Senden von Daten die Ausführungsreihenfolge schwer nachvollziehbar gestalten würden.

Beim Auswerten einer aufgezeichneten Ausführung – Tracing genannt – stellt das vorher erwähnte Zeitproblem eine kleinere Schwierigkeit dar. In Haiku gibt es dafür zwei Verfahren[24]. Jedoch stellt sich bei beiden Verfahren die Frage, wie sich die anfallenden Datenmengen auf ein handhabbares Maß begrenzen lassen. Dafür ist im Grunde bereits ein Verständnis des zu untersuchenden Quellcodes notwendig, um die Randbedingungen[25] des Tracers spezifisch genug setzen zu können. Das Problem der anfallenden Datenmengen ist jedoch genereller Natur und nicht auf die von Haiku bereitgestellten Werkzeuge zurückzuführen.

Kopfsimulation

Es zeigt sich somit, dass für die Analyse des Netzwerkstacks das virtuelle Ausführen des Quellcodes im Kopf unabdingbar ist. Für diese organische Simulation reicht in der Regel ein Texteditor, ein Suchprogramm und Kenntnisse in den eingesetzten Programmiersprachen[26] aus. Etwas komfortabler als ein reiner Texteditor und ein Suchprogramm sind Entwicklungsumgebungen, wie Eclipse[27] oder QT Creator, mit eingebauter Quell-

[24] Das Programm strace für die Aufzeichnung von Systemcalls und ltrace für die Aufzeichnung von Bibliotheksaufrufen.
[25] Aufzeichnungszeitraum, Aufrufpfadtiefe, interessierende Komponenten, ...
[26] Überwiegend C und C++ sowie etwas Assembler.
[27] Mit dem C/C++ Development Tooling (CDT) Plugin.

codenavigation. Diese vereinen die Texteditor- und Suchprogrammfunktionalitäten und erleichtern dadurch das hin und her Springen zwischen einzelnen Funktionen. Dabei werden Funktionsaufrufe im Quellcode als Link dargestellt (wie bei einem Webbrowser), wobei ein Anklicken die Implementierung der entsprechenden Funktion aufruft. In der Abbildung 3.6 wird eine solche Quellcodenavigation unter Verwendung von QT Creator demonstriert.

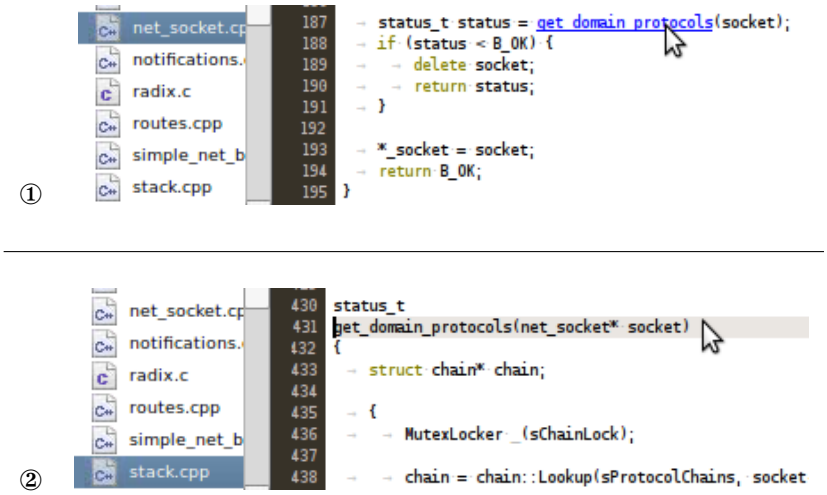


Abbildung 3.6.: Quellcodenavigation in QT Creator

Neben der Navigation innerhalb einer Quellcodedatei und zwischen mehreren Quellcodedateien bieten diese Entwicklungsumgebungen auch ein Variablenhervorhebungsfunktionalität. Dabei wird das Auftreten einer Variable innerhalb ihres Gültigkeitsbereiches (scope) optisch hervorgehoben. Dies erleichtert das Ableiten des Variablenzweckes aus ihrer Verwendung in den entsprechenden Quellcodeabschnitten. Abbildung 3.7 demonstriert diese Hervorhebungsfunktionalität ebenfalls an der Entwicklungsumgebung QT Creator.

```
static status_t
create_socket(int family, int type, int protocol, net_socket_private** _socket)
{
	struct net_socket_private* socket = new(std::nothrow) net_socket_private;
	if (socket == NULL)
		return B_NO_MEMORY;

	socket->family = family;
	socket->type = type;
	socket->protocol = protocol;

	status_t status = get_domain_protocols(socket);
	if (status < B_OK) {
		delete socket;
		return status;
	}

	*_socket = socket;
	return B_OK;
}
```

Abbildung 3.7.: Hervorhebung der Variablenverwendung in QT Creator

3.3.2.2. Verzeichnisstruktur

Für das Quellcodestudium ist es unerlässlich zu wissen, wo sich der zu studierende Quellcode überhaupt befindet. Das Haikuprojekt stellt für die Toplevel-Verzeichnisstruktur ein Einführungsdokument ([RepoLay]) bereit, in dem der Zweck der einzelnen Toplevel-Verzeichnisse beschrieben wird. Abbildung 3.8 beinhaltet eine Baumdarstellung dieser Toplevelstruktur, wobei die, für im Folgenden relevanten, Verzeichnisse fett hervorgehoben sind. Das `docs`-Verzeichnis, beinhaltet Dokumentation aller Art, `headers` enthält Haiku-weit verwendbare Headerdateien und in `src` befindet sich schließlich der compilierbare Sourcecode sowie Komponenten-private[28] Headerdateien.

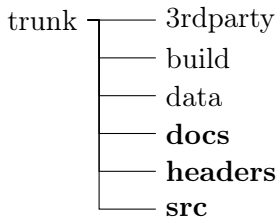


Abbildung 3.8.: Toplevel-Verzeichnisstruktur von Haiku

Das Herausfinden der Verteilung der Netzwerkstackfunktionalitäten auf die einzelnen Unterverzeichnisse ist jedoch schon eine Eigenleistung, welche teilweise auf Quellcodestudium beruht. Ein Ergebnis dieses Studiums ist somit die folgende netzwerkspezifische Verzeichniseingrenzung:

[28] Haiku ist in viele Komponenten aufgeteilt, wobei manche Komponenten Headerdateien verwenden, die nur im Rahmen der Komponente sinnvoll genutzt werden können, und somit nicht Haiku-weit bereitgestellt werden.

- Netzwerkstackdokumentation befindet sich im Verzeichnispfad `docs/develop/net/`.
- Haiku-weite Netzwerkheader sind in den Verzeichnissen `headers/os/`[29], `headers/posix/`[30] und `headers/private/net/`[31] zu finden.
- Netzwerkstack-relevanter Quellcode befindet sich schließlich in den Verzeichnissen `src/add-ons/kernel/network/`[32], `src/add-ons/kernel/drivers/net/`[33], `src/kits/network`[34], `src/servers/net/`[35] und `src/system/kernel/fs/`[36].

3.3.2.3. Buildsystem

Das Buildsystem ist ebenfalls eine hilfreiche Informationsquelle, wenn es um das Verstehen einer unbekannten Komponente geht. Es enthält Anweisungen, welche Quellcodedateien beziehungsweise Bibliotheken zu einer ausführbaren Komponente zusammengefasst werden. Diese Anweisungen sind in einer buildsystemeigenen Sprache verfasst.

Bei Haiku kommt das Buildsystem jam zum Einsatz, welches Builddateien mit dem Namen `Jamfile` zur Steuerung des Buildprozesses verwendet. Beispielsweise wird im folgenden Listing am Ausschnitt eines Jamfiles[37] verdeutlicht, welche Quelldateien und Bibliotheken notwendig sind, damit der Netzwerkdienst `net_server` gebaut werden kann.

```
1  Server net_server :
2      NetServer.cpp
3      Settings.cpp
4      AutoconfigClient.cpp
5      AutoconfigLooper.cpp
6      DHCPClient.cpp
7      Services.cpp
8
9      : be libnetwork.so $(TARGET_LIBSTDC++)
10     # for PPP
11     #libppp.a
12 ;
```

Das Rautesymbol leitet dabei einen Kommentar ein und das Semikolon deutet das Ende einer Anweisung an. Ohne weiter in die Syntax und Semantik der Jam-Sprache eingehen zu wollen, sei der Augenmerk auf die erste Zeile gelegt. Die Anweisung `Server` bekommt als ersten Parameter den Namen des zu erstellenden Servers übergeben. Nach dem ersten Doppelpunkt folgen alle Bestandteile, welche für den Bau des `net_server`s benötigt werden. Der Doppelpunkt ist dabei das Trennsymbol der einzelnen Parameter, ähnlich

[29] Enthält unter anderenm die öffentliche objektorientierte Netzwerkschnittstelle.
[30] Enthält unter anderem die öffentliche, POSIX-konforme Socketsschnittstelle.
[31] Enthält die Haiku-interne Netzwerkschnittstelle.
[32] Enthält den Hauptteil des Netzwerkstacks.
[33] Enthält die Netzwerktreiber.
[34] Enthält unter anderem die Implementierung der objektorientierten Schnittstelle.
[35] Enthält den Netzwerkkonfigurationsdienst net_server.
[36] Enthält unter anderem die Sockets-Systemrufschnittstelle.
[37] Jamfilepfad: `src/servers/net/Jamfile`

wie das Komma die Parameter einer C-Funktion trennt. Zeile 2-7 stellt somit einen weiteren Parameter dar, der eine Liste aller benötigten Quellcodedateien enthält.

Zeile 9 beherbergt den dritten und letzten Parameter, der - wieder in Form einer Liste – die dynamisch hinzuzulinkenden Bibliotheken angibt. Die Angabe `$(TARGET_LIBSTDC++)` ist dabei eine Bezugnahme auf den Inhalt der Variable `TARGET_LIBSTDC++`, also der Standard-C++-Bibliothek von Haiku. Der Namensbestandteil `TARGET` ist ein Hinweis darauf, dass nicht die Standard-C++-Bibliothek der Buildumgebung (zum Beispiel GNU-Linux) sondern des Zielsystems (also Haiku) verwendet wird. Die Zeilen 10-11 sind auskommentiert und das Semikolon in Zeile 12 schließt die Serveranweisung ab.

Ein Vorteil von jam gegenüber dem bekannteren make-Buildsystem ist die automatische Ermittlung der benötigten Headerdateien. Deswegen „fehlen" diese Angaben im obigen Listing.

3.3.2.4. Beispielanalyse: Socketerzeugung

Zu Beginn der Netzwerkstackanalyse – wie bei der Analyse jeder unbekannten Komponente – stellen sich folgende Fragen:

1. In welcher Komponente soll mit der Ausführung begonnen werden?
2. Wo in der Komponente beginnt die Ausführung?
3. Welche Randbedingungen[38] liegen zum Ausführungsbeginn vor?
4. Wie lässt sich der weitere Aufrufpfad im Quellcode verfolgen?

Anhand der Erzeugung eines neuen Sockets wird demonstriert, wie diese Fragen unter Verwendung der Kopfsimulation (Abschnitt 3.3.2.1) beantwortet werden können.

Ausgangspunkt

Für die Erzeugung eines neuen Sockets existiert die Funktion `socket()`. Dies kann zum Beispiel dem Quellcode des Netzwerkdienstes `net_server` entnommen werden. Das Quellcodelisting zeigt dabei einen Ausschnitt aus der Funktion `NetServer::_BringUpInterfaces()`[39] des `net_servers`. Die beiden Doppelpunkte vor dem `socket()`-Funktionsaufruf (Zeile 4) definieren dabei den Namensraum, innerhalb dessen der Funktionsname Gültigkeit besitzt. Andernfalls ließe sich der Funktionsname `socket` für den Kompiler nicht von der lokalen Variable `socket` unterscheiden.

```
1 NetServer::_BringUpInterfaces()
2 {
3     // we need a socket to talk to the networking stack
4     int socket = ::socket(AF_INET, SOCK_DGRAM, 0);
5     if (socket < 0) {
```

[38] Werte globaler Variablen, Werte von übergebenen Parametern der Startfunktion, ...
[39] Datei: `src/servers/net/NetServer.cpp`

```
6           fprintf(stderr, "%s:␣The␣networking␣stack␣doesn't␣seem␣to␣be␣"
7               "available.\n", Name());
8           Quit();
9           return;
10      }
```

Die Entscheidung, das Beispiel mit dieser Funktion des `net_servers` zu beginnen, erscheint zunächst willkürlich. Die Fragen 1 und 2 wurden schließlich implizit so beantwortet, dass die Ausführung in der Komponente `net_server` und der Funktion `_BringUpInterfaces()` begonnen wird. Für das Beispiel würde es zu weit gehen, die Herleitung der Antworten auf die beiden Fragen zu demonstrieren. Soviel sei gesagt, den Fragen 1 und 2 ist eine Kopfsimulation des Bootvorgangs von Haiku vorangegangen. Die Zielstellung bei dieser Simulation war, allgemein zu verstehen, wann Haiku das erstemal Netzwerkfunktionalitäten beansprucht. Dabei hat sich ergeben, dass dies eben genau an der besagten Quellcodestelle mit dem Erzeugen eines neuen Sockets geschieht.

Randbedingungen

Die dritte Frage nach den Randbedingungen lässt sich zunächst nur unvollständig beantworten. So kann zwar die Frage nach den Startparametern leicht[40] beantwortet werden, da `socket()` drei Konstanten[41] übergeben bekommt, doch es lässt sich noch nicht abschätzen, welche anderen Randbedingungen von Interesse sein könnten. Solche Randbedingungen werden meist erst während der weiteren Simulation deutlich, was dann zu neuen Simulationssträngen[42] führen kann. In dieser Unvollständigkeit der Startbedingungen liegt also ein schwer abschätzbarer Quellcodeanalyseumfang verborgen.

Es bleibt noch die Beantwortung der vierten Frage. Diese Frage betont dabei den Aspekt der Quellcodeanalyse, der letztendlich das Verständnis für die Funktionsweise der untersuchten Komponente liefert. Zur Verfolgung des Aufrufpfades wird von nun an die Entwicklungsumgebung Eclipse verwendet, da deren Quellcodenavigation (zum Zeitpunkt dieser Masterarbeit) sich gegenüber derer von QT Creator als ausgereifter erwies. Diese Überlegenheit wird bei der Suche nach der Datei, in der `socket()` implementiert ist, deutlich. Klickt man in QT Creator auf den Link werden lediglich alle Vorkommen der lokalen Variable `socket` hervorgehoben. In Eclipse jedoch erscheint der Auswahldialog der Abbildung 3.9.

[40] Im Vergleich zu Variablen, deren Inhalt vorher noch ermittelt werden müsste.
[41] Konvention von C/C++: Konstantennamen werden stets mit Großbuchstaben geschrieben.
[42] Zur Wertermittlung der neuen Randbedingungen.

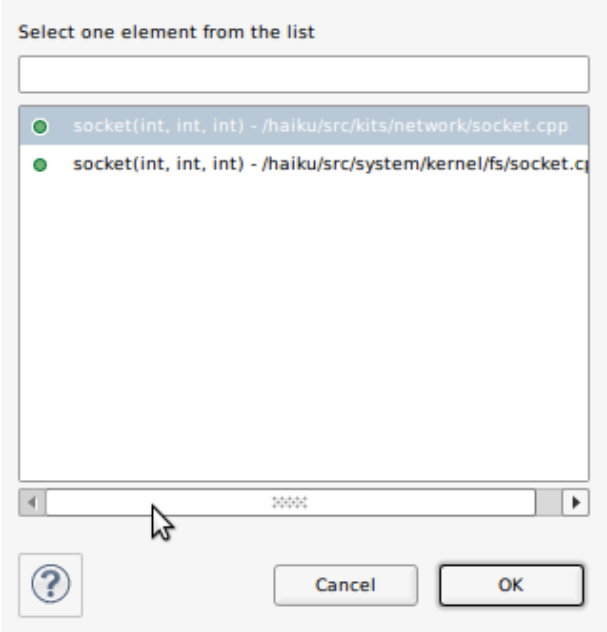


Abbildung 3.9.: Auswahldialog für die Implementierung von socket()

Bestimmung der Implementierung von socket()

Welche der beiden Dateien enthält die `socket()`-Implementierung, welche von `_BringUpInterfaces()` tatsächlich aufgerufen wird? Hier liefert das Buildsystem einen wichtigen Hinweis. Im `Jamfile`-Listing des Abschnitts 3.3.2.3 sind ja schließlich alle Bestandteile von `net_server` aufgelistet. Der entscheidende Hinweis ist in Zeile 9 zu finden, in dem die dynamische Bibliothek `libnetwork.so` als Bestandteil der `net_server`-Komponente ausgewiesen ist. Die Bauanweisung – also das nachfolgende auszugsweise gelistete Jamfile[43] – für die Bibliothek `libnetwork.so` beinhaltet auch die zur Auswahl stehende Datei `src/kits/network/socket.cpp`.

```
1  SubDir HAIKU_TOP src kits network ;
2
3  UsePrivateHeaders app libroot net shared ;
4  UsePrivateSystemHeaders ;
5
6  SharedLibrary libnetwork.so :
7      init.cpp
8      interfaces.cpp
9      notifications.cpp
10     socket.cpp
11     r5_compatibility.cpp
12     :
```

[43] Pfad: `src/kits/network/Jamfile`

Damit die in Zeile 10 aufgeführte `socket.cpp`-Datei als die zu wählende `src/kits/network/socket.cpp` Datei identifiziert werden kann, ist etwas Hintergrundwissen über die Arbeitsweise des jam-Buildsystems notwendig. Dieses ermittelt den Pfad der Quellcodedateien mittels einer globalen Variable `SEARCH_SOURCE`, welche die zu durchsuchenden Pfade enthält. Der Befehl `SubDir` in der ersten Zeile manipuliert diese globale Variable.

Dabei wird zunächst der an `SubDir` übergebene Pfadparameter „`HAIKU_TOP src kits network`" zu `src/kits/network` aufgelöst, wobei `HAIKU_TOP` eine Variable ist, die den Pfad zum Toplevel-Verzeichnis (Abschnitt 3.3.2.2) angibt und hier als leer angenommen wird. `SubDir` trägt diesen Pfad nun in `SEARCH_SOURCE` ein, so dass der Pfad die oberste Priorität bei der Pfadbestimmung der Quellcodedateien bekommt.

Implementierung von socket()

Somit befindet sich die Implementierung der von `_BringUpInterfaces()` aufgerufenen `socket()`-Funktion in der Datei `src/kits/network/socket.cpp`. Das nachfolgende Listing enthält die vollständige Implementierung der `socket()`-Funktion, anhand derer die Simulation fortgesetzt werden kann.

```
1 socket(int family, int type, int protocol)
2 {
3     if (check_r5_compatibility())
4         convert_from_r5_socket(family, type, protocol);
5
6     RETURN_AND_SET_ERRNO(_kern_socket(family, type, protocol));
7 }
```

Die beiden Funktionsaufrufe der Zeilen 3 und 4 können mit Hilfe der Codenavigation problemfrei verfolgt werden. Ihre Aufgabe lässt sich auch ihrem Namen ableiten – Bereitstellung von BeOS R5 Kompatibilität – und ist somit eine Umsetzung der gesteckten Ziele des Haiku-Projektes.

Besonderheit bei Systemrufen

Schwieriger gestaltet sich die Simulation der Zeile 6. Die Aufgabe des Makros `RETURN_AND_SET_ERRNO` lässt sich noch mittels Codenavigation ermitteln[44], doch zur Funktion `_kern_socket()` lässt sich keine Implementierung finden. Diese Funktion ist ein sogenannter Systemruf (Syscall) – wie alle Funktionen die mit Namensbestandteil `_kern` beginnen – und besitzt keine Implementierung unter diesem Namen.

Dieses Wissen lässt sich entweder über eine Anfrage auf der haiku-developer-Mailinglist in Erfahrung bringen, oder besser durch vorhergehendes Studium der bereitgestellten Dokumentation. Dort wird in [Syscalls] ausführlich der Syscall-Mechanismus mit dem Hinweis beschrieben, dass die implementierenden Funktionen statt dem Präfix `_kern` das Präfix `_user` tragen. Es handelt sich hierbei also um den Übergang vom Userland ins Kernelland, welcher an den Funktionsnamen kenntlich gemacht wird.

[44] Setzen des Rückgabewertes der Funktion `_kern_socket()` in der POSIX-Fehlervariable `errno`.

Im Kernelland

Eine suche in Eclipse nach `_user_socket` führt schließlich zu einem eindeutigen Treffer in der Datei `src/system/kernel/fs/socket.cpp`. Die vollständige Implementierung von `_user_socket()` ist im folgenden Listing dargestellt.

```
1  _user_socket(int family, int type, int protocol)
2  {
3      SyscallRestartWrapper<int> result;
4      return result = common_socket(family, type, protocol, false);
5  }
```

Die Bedeutung der Variable `result` und ihrer Klasse `SyscallRestartWrapper` ist bei der weiteren Simulation des Ausführungspfades uninteressant, es sei aber darauf hingewiesen, dass hier templatebasierte[45] Objektorientierung im Kernel Anwendung findet.

Der Ausführungspfad wird mit der Implementierung von `common_socket` fortgesetzt, welche innerhalb der selben Datei zu finden ist. Ein Auszug aus der Implementierung wird nachfolgend gegeben.

```
1  common_socket(int family, int type, int protocol, bool kernel)
2  {
3      if (!get_stack_interface_module())
4          return B_UNSUPPORTED;
```

Auch die Funktion `get_stack_interface_module()` befindet sich in der selben Quellcodedatei und beginnt mit den folgenden Zeilen:

```
1  get_stack_interface_module()
2  {
3      MutexLocker _(sLock);
4
5      if (sStackInterfaceInitialized++ == 0) {
6          // load module
7          net_stack_interface_module_info* module;
8          // TODO: Add driver settings option to load the userland net stack.
9          status_t error = get_module(NET_STACK_INTERFACE_MODULE_NAME,
10             (module_info**)&module);
```

Die Aufgabe der Zeile 3 leiten wir direkt aus dem Namen der Klasse ab – Absicherung eines kritischen Abschnitts mittels eines Mutex' `sLock`[46] – ohne uns der Quellcodenavigation zu bemühen, da dies hier zu sehr ins Detail gehen würde.

Erweiterung der Randbedingungen

Die Inhaltsprüfung der Variablen `sStackInterfaceInitialized` in Zeile 5 erfordert die Kenntnis einer Randbedingung, ohne die die Simulation nicht fortgeführt werden kann.

[45] Templates sind eine Form der Generischen Programmierung und sind durch die spitzen Klammern gekennzeichnet.

[46] Die Definition dieser Variable befindet sich am Anfang der selben Quellcodedatei.

Dieses Stückchen Quelltext ist somit ein Beispiel für die anfangs[47] erwähnte Wissenslücke der notwendigen Randbedingungen. Für die Ermittlung des Wertes von `sStackInterfaceInitialized` muss ein zweiter Simulationsstrang gestartet werden, für den zunächst wieder alle Fragen[48] beantwortet werden müssten.

Ausführungspfadeingrenzung

Doch es gibt noch einen anderen Weg den Wert der neuen Randbedingung zu ermitteln. Dabei kann auf ein erneutes Beantworten der ersten drei Fragen verzichtet werden. Dieser Weg lässt sich auch als Ausführungspfadeingrenzung bezeichnen. Dabei werden schrittweise – ausgehend von der Randbedingung – alle Simulationspfade bestimmt, welche Einfluss auf den Wert der gesuchten Randbedingung haben können. Die resultierenden Ausführungspfade können in Form eines Aufrufgraphen (Callgraph) visualisiert werden. Am Beispiel von `sStackInterfaceInitialized` soll dies nun demonstriert werden.

In Eclipse wird die Ausführungspfadeingrenzung durch eine Funktionalität namens `Open Call Hierarchy` unterstützt. Dazu ist es notwendig `sStackInterfaceInitialized` zu selektieren und im zugehörigen Kontextmenü `Open Call Hierarchy` auszuwählen. In Abbildung 3.10 wird dabei die Anwendung (①) dieser Funktionalität auf `sStackInterfaceInitialized` und das Ergebnis (②) – der Callgraph – dargestellt.

[47] Abschnitt Randbedingungen, Seite 48
[48] Abschnitt 3.3.2.4

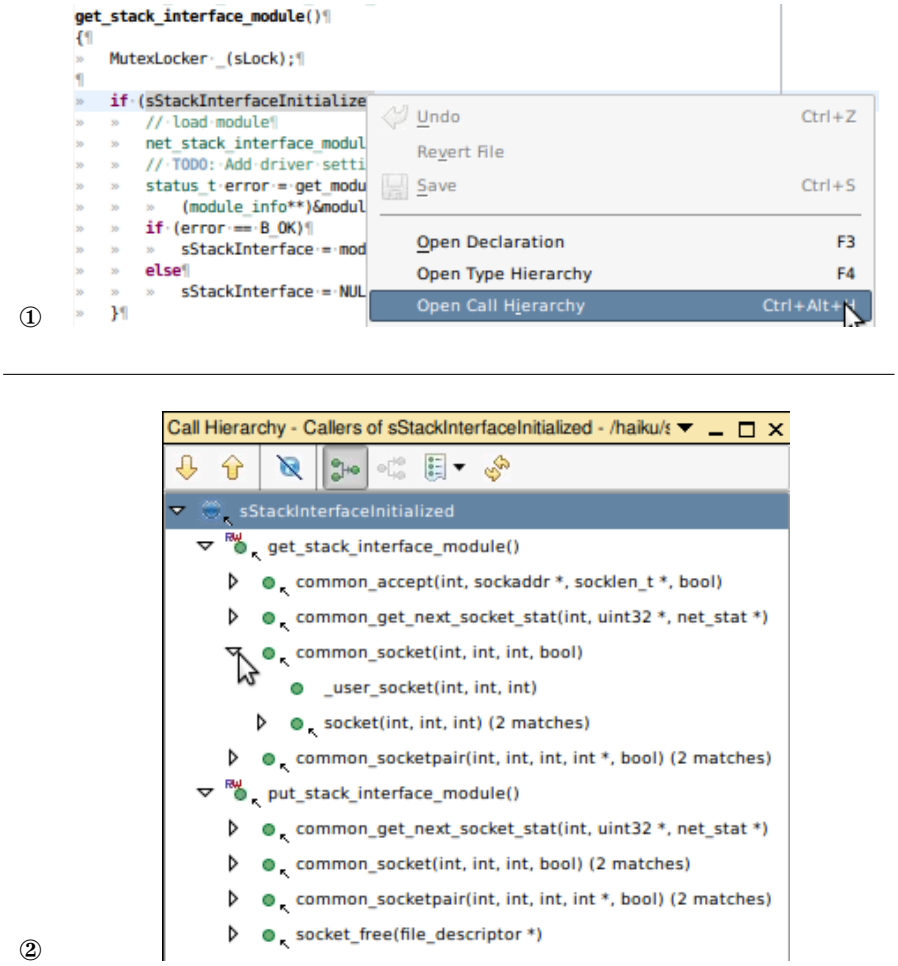


Abbildung 3.10.: Ausführungspfadeingrenzung mittels Callgrapherzeugung

Der Callgraph weist dabei eine Baumstruktur auf, an deren Wurzel `sStackInterfaceInitialized` steht. Die einzelnen Äste des Baumes lassen sich dabei bis zu den Blättern weiterverfolgen. In der Abbildung wird dies für den Ast `sStackInterfaceInitialized` ▷ `get_stack_interface_module()` ▷ `common_socket()` ▷ `_user_socket()` demonstriert. Der Callgraph kann dabei aufgrund der Systemrufgrenze nur die Kernelland-Aufrufpfade abdecken. Doch für die Bestimmung des Wertes von `sStackInterfaceInitialized` ist das vollkommen ausreichend.

Wissenskombination

Der Wert von `sStackInterfaceInitialized` muss also Null sein. Dies lässt sich durch die Kombination des Callgraphen mit den Ausgangsüberlegungen[49] zur Socketerzeugung

[49] Abschnitt Ausgangspunkt, Seite 47

herleiten. Dort wurde gesagt, dass die Funktion `_BringUpInterfaces()` des `net_server`s den ersten Zugriff auf Haikus Netzwerkstack durchführt. Es kann bis jetzt also nur der Ast im Callgraph beschritten worden sein, auf dem die Ausgangssimulation zur Variable `sStackInterfaceInitialized` gelangt ist.

Weiterhin zeigt ein Blick auf die Definition von `sStackInterfaceInitialized` im nachfolgenden Listing[50], dass diese Variable mit Null initialisiert wird. Deswegen kann nur dieser Wert für die weitere Durchführung der Ausgangssimulation zulässig sein.

```
1 static vint32 sStackInterfaceInitialized = 0;
```

Abschluss

Dank der Ausführungspfadeingrenzung und der anschließenden Wissenskombination ist nun klar, dass die Bedingung in der Funktion `get_stack_interface_module()` erfüllt ist. Als nächstes würde die Simulation daher mit dem Aufruf von `get_module()` fortfahren. Bis zur eigentlichen Erzeugung eines Sockets ist es allerdings noch ein weiter Weg.

An dieser Stelle soll die Simulation abgebrochen werden. Alle wesentlichen Analysetechniken – Codenavigation, Spezialwissen aus externer Dokumentation, Callgrapherstellung und Wissenskombination – wurden exemplarisch vorgestellt. Bei der Behandlung der Bedarfsinitialisierung des Netzwerkstacks im Abschnitt 3.3.3.1 wird die Socketerzeugung erneut aufgegriffen. Dabei wird diese vollständiger behandelt als in diesem Abschnitt, allerdings auch auf einer geringeren Detailstufe.

Aus dem bisherigen Ausführungspfad lässt sich abschließend das in Abbildung 3.11 dargestellte Sequenzdiagramm erstellen.

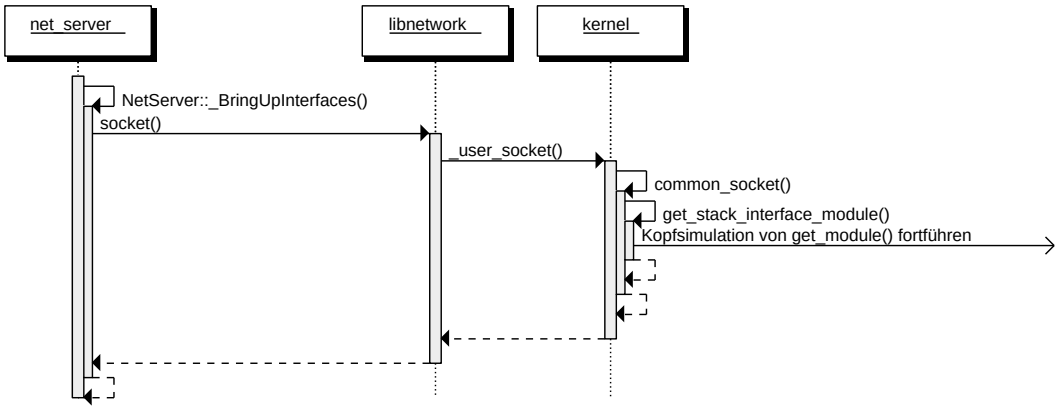


Abbildung 3.11.: Ausführungspfad als Sequenzdiagramm

[50] nach wie vor in der Quelldatei `src/system/kernel/fs/socket.cpp`

Zusammenfassung

Es wurde am Beispiel der Socketerzeugung der Einsatz verschiedener Analysetechniken demonstriert, die bei der Kopfsimulation behilflich sind. Dabei wurde die Anwendung der Analysetechniken Codenavigation und Callgrapherstellung unter Verwendung der Entwicklungsumgebung Eclipse vorgeführt.

3.3.3. Funktionsweise

Im Folgenden werden die Interaktionen der einzelnen Schichten von Haikus Netzwerkstack (Abschnitt 3.3.1) – beruhend auf dem Quellcodestudium (Abschnitt 3.3.2 auf Seite 42) – näher beleuchtet. Dafür werden die beiden Hauptfunktionalitäten Senden und Empfang von Daten und zusätzlich die Bedarfsinitialisierung des Netzwerkstacks in den entsprechenden Abschnitten behandelt.

Da es jedoch keine Literatur über die Funktionsweise gibt, wurde diese durch Studieren des Quellcodes hergeleitet. Die dabei eingesetzten Verfahren wurden im Abschnitt 3.3.2 vorgestellt.

3.3.3.1. Bedarfsinitialisierung

Haikus Netzwerkstack wird erst dann initialisiert wenn er benötigt wird (Lazy-Initialisierung). Mit anderen Worten: Die erste Verwendung einer der in Abschnitt 3.3.1 vorgestellten Schnittstellen, löst die Initialisierung aus. Dabei wird die Netzwerkfunktionalität zuerst vom Systemdienst net_server benötigt, welcher direkt im Bootscript gestartet wird. Der net_server selbst stellt typische Netzwerkdienste wie dynamisches (DHCP) und statisches Konfigurieren der Netzwerkschnittstellen, bereit.

Protokoll- und Datalinkschicht

Da im Grunde egal ist, welche Anwendung zuerst auf den Netzwerkstack zugreift, beginnt die folgende Diskussion direkt mit der `socket()`-Funktion. Diese muss von jeder Anwendung direkt[51] oder indirekt[52] zuerst aufgerufen werden, wenn sie den Netzwerkstack verwenden möchte. Somit stellt `socket()` den Ausgangspunkt für die Lazy-Initialisierung dar. In Abbildung 3.12 wird schließlich in Form eines (bereits vereinfachten) Sequenzdiagramms gezeigt wie die ersten Netzwerkstackschichten genutzt und damit auch initialisiert werden.

[51] mittels Socketschnittstelle
[52] mittels Network-Kit-Schnittstelle

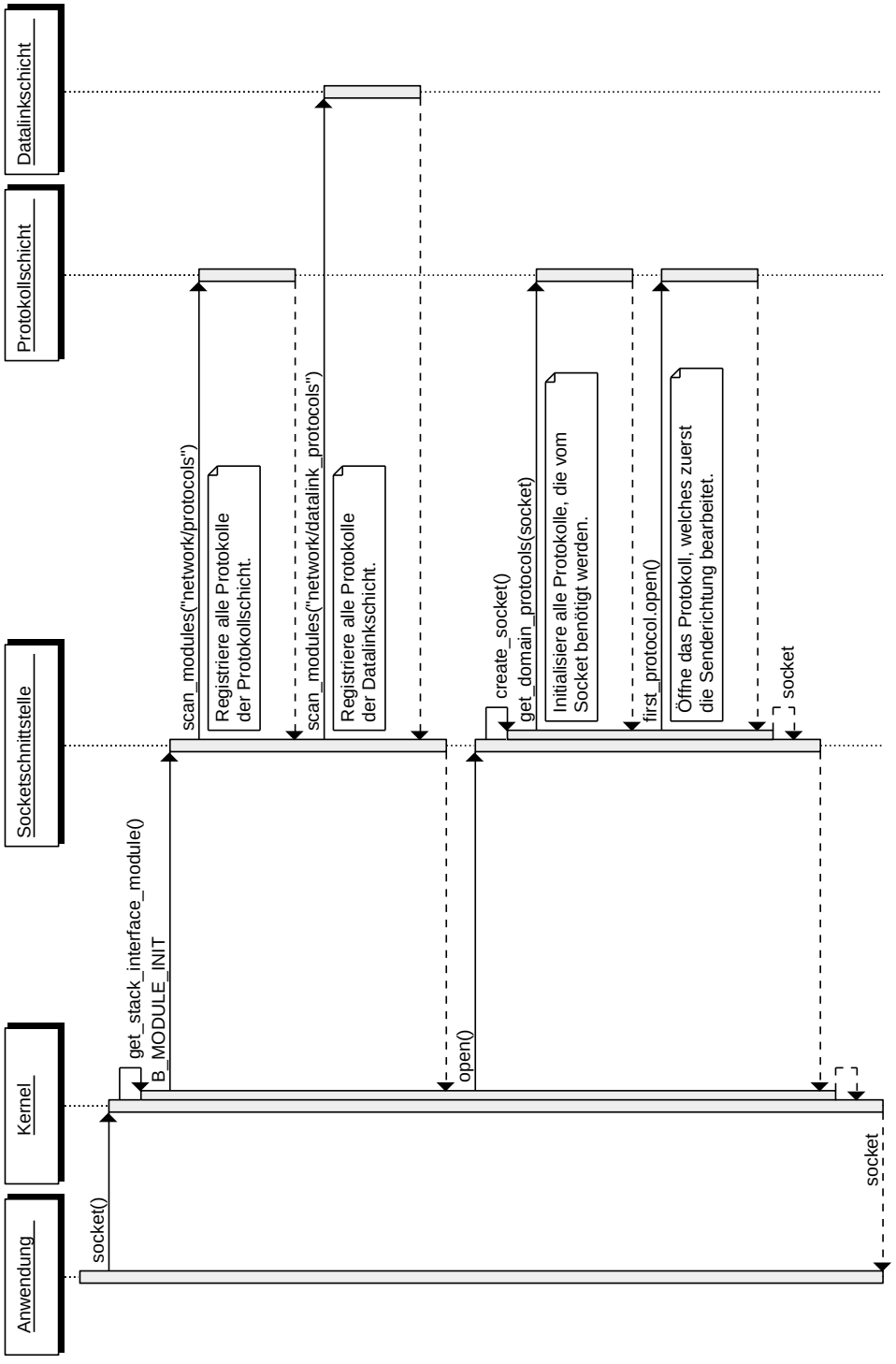


Abbildung 3.12.: Lazy-Initialisierung der Protokoll- und Datalinkschicht

Nach Beendigung der `socket()`-Funktion wurden somit die Protokoll- und die Datalinkschicht initialisiert. Die Geräte- und die Treiberschicht blieben unberührt. Daran wird der Lazy-Initialisierungsansatz des Netzwerkstacks nocheinmal deutlich, denn die Erzeugung eines neuen Sockets benötigt noch keine Geräte- und Treiberschichtenfunktionalitäten.

Geräte- und Treiberschicht

Die Bedarfsinitialisierung von Geräte- und Treiberschicht ist ein mehrstufiger Prozess, der folgende Stufen umfasst:

1. Treiberschichtinitialisierung,
2. Geräteschichtinitialisierung und
3. Verbindung von Geräte- und Treiberschicht

Dieser Vorgang wird im Folgenden anhand der Netzwerkgeräteerkennung, wie sie vom net_server-Dienst vorgenommen wird, diskutiert.

Die **Treiberschichtinitialisierung** (Abbildung 3.13) erfolgt durch den erstmaligen Zugriff auf den Dateisystempfad `/dev/net`. Dadurch wird eine Kettenreaktion in Gang gesetzt, die mit dem Laden und Initialisieren (`init_driver()`) aller Netzwerktreiber beginnt. Findet der Netzwerktreiber die zugehörige Hardware vor, meldet er das an die Treiberschicht, andernfalls wird er wieder entladen.

Treiber, die ihre Hardware vorgefunden haben, werden schließlich von der Treiberschicht gefragt (`publish_devices()`), unter welchem Pfad sie angesprochen werden möchten. Beispielsweise wird eine sis900-Netzwerkkarte hier den Pfad `/dev/net/sis900/0` angeben.

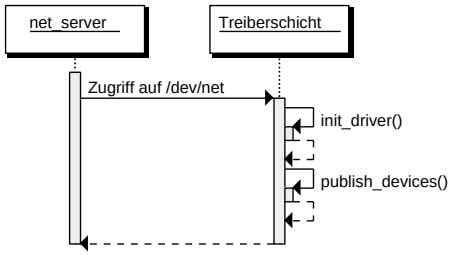


Abbildung 3.13.: Bedarfsinitialisierung der Treiberschicht

Die **Geräteschichtinitialisierung** (Abbildung 3.14) benötigt einen bereits erzeugten Socket. Dieser wird beim Aufruf der Socketschnittstellen-Funktion `ioctl()`, neben `SIOCAIFADDR`[53], als weiterer Parameter übergeben. Der in Abbildung 3.14 verwendete Pfad `/dev/net/sis900/0` gibt das Gerät an, welches dem Netzwerkstack hinzugefügt werden soll.

[53] Socket I/O Add Interface Address: Füge Netzwerkgerät dem Stack hinzu.

Dieser Funktionsaufruf landet in der Datalinkschicht. Diese fragt schließlich alle Komponenten der Geräteschicht ab (`init()`), ob das Gerät unterstützt wird. Im Falle von `/dev/net/sis900/0` wird die Ethernetkomponente eine positive Antwort liefern und dadurch der Ansprechpartner für die Datalinkschicht (bezüglich dieses Gerätes) werden. Somit stellt die Geräteschichtinitialisierung auch die Verbindung von Datalinkschicht und Geräteschicht her.

Für das Verständnis des Datenempfangs wurde in der Abbildung 3.14 auch die Erzeugung des Demultiplexers mit eingezeichnet. Dieser verteilt empfangene Daten auf die entsprechenden Komponenten der Datalinkschicht und wird im Abschnitt 3.3.3.3 weiter behandelt.

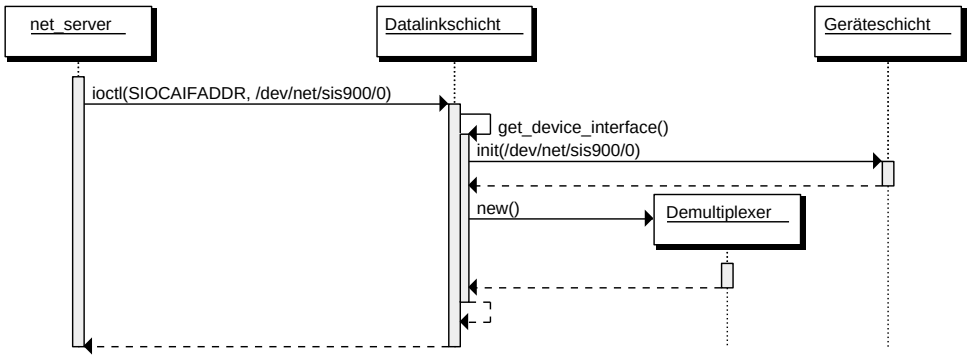


Abbildung 3.14.: Bedarfsinitialisierung der Geräteschicht

Damit auch Daten zwischen Geräte- und Treiberschicht ausgetauscht werden können, wird im letzten Schritt noch die **Verbindung zwischen Geräte- und Treiberschicht** hergestellt. Hierfür wird wieder der – aus der Geräteschichtinitialisierung bekannte – Socket benötigt. Es kommt auch die selbe Funktion `ioctl()` zum Einsatz, wobei diesmal allerdings der Parameter `SIOCSIFFLAGS`[54] verwendet wird, um das Flag `IFF_UP`[55] auf dem Netzwerkgerät `/dev/net/sis900/0` (in Abbildung 3.15 nicht mit angegeben) zu setzen.

Die Datalinkschicht aktiviert (`up()`) darauf die Geräteschicht (genaugenommen die Ethernetkomponente). Diese wiederum aktiviert zunächst die Treiberschicht[56] (`open()`) und führt anschließend einige Konfigurationsaktionen (`ioctl()`) durch. Damit ist die Verbindung zwischen Geräte- und Treiberschicht hergestellt und die Initialisierungsdiskussion im Wesentlichen abgeschlossen.

Der Deframer ist später für das Verständnis des Datenempfangs (Abschnitt 3.3.3.3) notwendig. Seine Aufgabe ist das Entfernen der netzwerktechnologiespezifischen Frames von den empfangenen Daten.

[54] Socket I/O Control Set Interface Flags: Setze Flags des Netzwerkgerätes.
[55] Interface Up: Aktiviere Netzwerkgerät.
[56] Genaugenommen den sis900 Treiber.

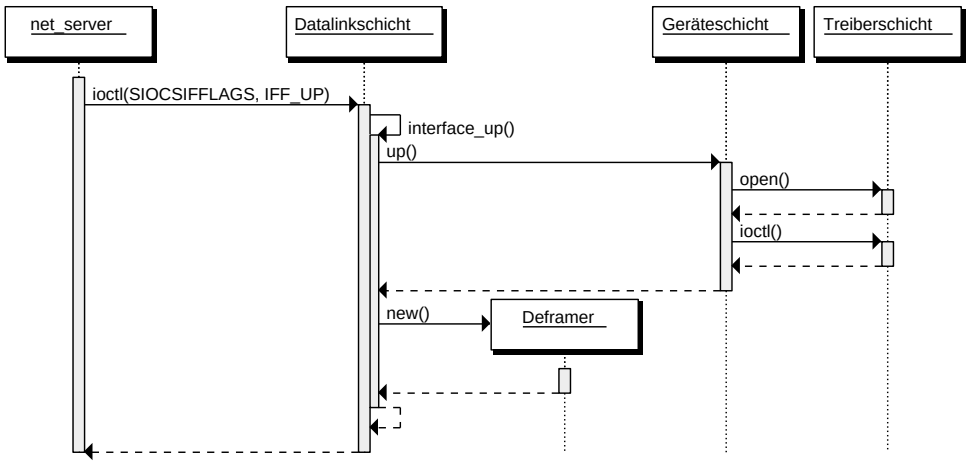


Abbildung 3.15.: Verbindung von Geräte- und Treiberschicht

3.3.3.2. Senden

Das Senden von Daten über ein Netzwerk wird anhand einer TCP/IP-Verbindung erklärt. Abbildung 3.16 zeigt dabei einen (vereinfachten) Aufrufpfad, den ein Datenpaket innerhalb des Netzwerkstacks durchläuft. Auslöser der Datenaussendung ist eine Anwendung, die durch Erzeugen eines Sockets und der Herstellung einer TCP/IP-Verbindung – beides nicht in der Abbildung 3.16 dargestellt – bereits den Aufrufpfad festgelegt hat. Unter anderem wird hier also festgelegt, welche Route die Datenaussendung innerhalb der Datalinkschicht und Geräteschicht nehmen wird.

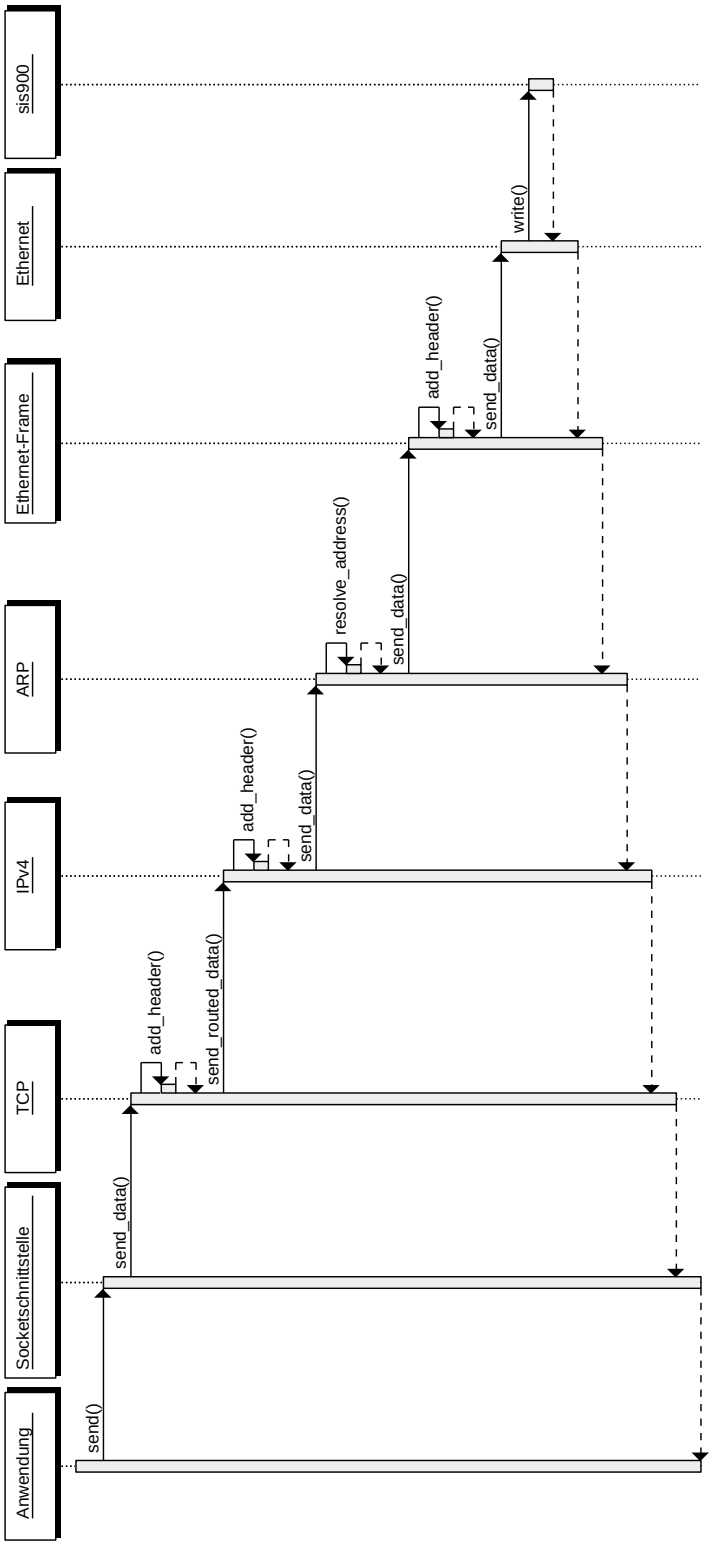


Abbildung 3.16.: Senden von Daten über TCP/IP-Verbindung

Die Funktion `send()` veranlasst die Socketschnittstelle dazu, die `send_data()`-Funktion der im Aufrufpfad ersten Komponente (`TCP`) aufzurufen. Diese fügt den übergebenen Daten einen TCP-Header hinzu und übergibt diese an die `send_routed_data()`-Funktion der nächsten Komponente (`IP`). Wie der Funktionsname `send_routed_data()` andeutet, wird der Funktion auch der Routingparameter übergeben, der später von der Datalinkschnittstelle verwendet wird, um den weiteren Aufrufpfad zu bestimmen.

Die, in der Abbildung 3.16 dargestellte, Geradlinigkeit der Datenaussendung ist natürlich der Vereinfachung geschuldet. So wird beispielsweise das ARP-Module nicht immer eine IP-Adresse durch alleinigen Zugriff auf seinen Adresscache auflösen können. Die Kernfunktion dieser Abbildung ist somit die Darstellung der beteiligten Netzwerkstackkomponenten. Mit Hilfe dieser Übersicht lässt sich dann in Abschnitt 4.5 die Eingliederung des objektorientierten WLAN-Stacks in Haiku besser verstehen.

3.3.3.3. Empfang

Der Datenempfang ist ein gutes Beispiel für den Einsatz von Multithreading in Haiku. Für das leichtere Verständnis wurde die Diskussion dabei auf mehrere Unterabschnitte aufgeteilt. Im Unterabschnitt „Threadklassen" wird eine Übersicht der beteiligten Threads und ihrer Aufgaben gegeben. In „Threadkommunikation" wird der Datenaustausch zwischen den Threads und in „Threadkontext" wird die Abgrenzung des Wirkungsbereichs der einzelnen Threads thematisiert. Abschließend stellt „Datenverarbeitung" den Weg eines einzelnen Datenpaketes über alle Schichten und Threads hinweg dar.

Threadklassen

Bei Betrachtung der Abbildung 3.17 wird verdeutlicht, dass es die drei Threadklassen

- Anwendung
- Demultiplexer
- Deframer

für den Empfang und die Verarbeitung der Daten gibt, welche eine Verarbeitungshierarchie bilden. Zur Klasse der Anwendung gehören alle Userland-Threads, deren Funktion natürlich von der Anwendung[57] abhängt. Die Aufgaben der Demultiplexer- und der Deframerklasse wurden bereits bei der Initialisierung der Geräte- und Treiberschicht im Abschnitt 3.3.3.1 vorgestellt.

Weiterhin ist auch zu sehen, dass von jeder Klasse gleichzeitig mehrere Instanzen (Threads) laufen können. Wobei jedem Netzwerkgerät jeweils genau eine Demultiplexerthreadinstanz und eine Deframerthreadinstanz zugeordnet ist. Letztendlich sind also zwei Threads pro Netzwerkgerät aktiv.

[57] zum Beispiel: Webbrowser

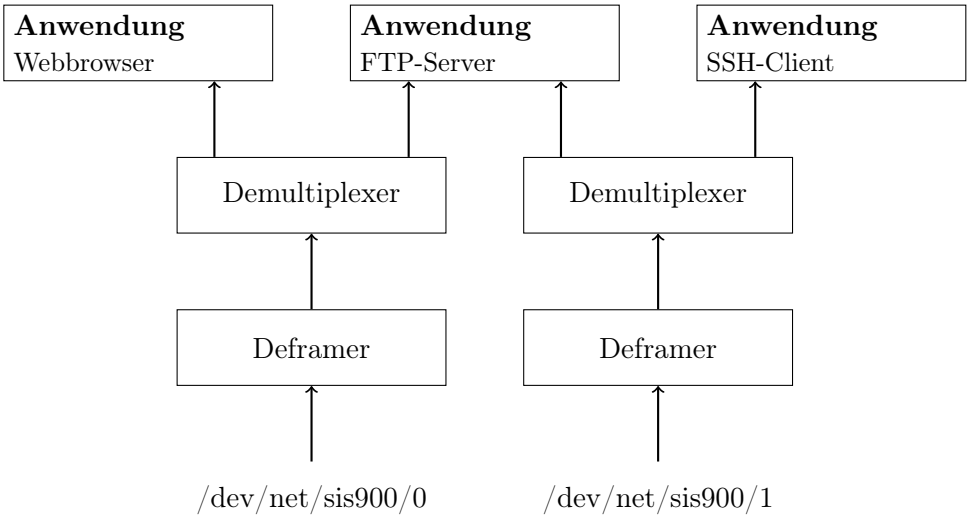


Abbildung 3.17.: Übersicht der drei Empfangsthreadklassen

Eine Anwendung besteht standardmäßig aus genau einem Thread. Zwar kann dieser mehrere Sockets gleichzeitig geöffnet haben[58], jedoch kann zu jedem Zeitpunkt stets nur ein Socket bedient werden. Die Anwendung hat aber immer die Möglichkeit, neue Threads zu erzeugen, um mehrere Sockets parallel bedienen zu können.

Zusammenfassend lässt sich sagen, dass am Empfang eines Datenpakets drei verschiedene Threads beteiligt sind.

Threadkontext

Zur einfacheren Unterscheidung der am Empfang beteiligten Threads und ihres Wirkungsbereiches (Kontext), werden die Sequenzdiagramme in den nachfolgenden Abbildungen entsprechend des Threadkontextes eingefärbt. Mit anderen Worten, wird die gleiche Farbe für alle Funktionsaufrufe (Aktivitäten) verwendet, die im Kontext ein und desselben Threads ausgeführt werden.

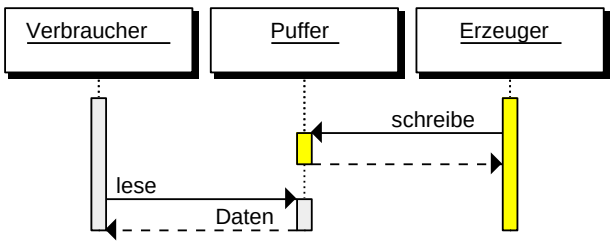


Abbildung 3.18.: Visualiserung des Threadkontexts von Aktivitäten

[58] Anwendungsfall: Webserver

Die Abbildung 3.18 demonstriert diese Einfärbungsregel dabei an einer Erzeuger-Verbraucher-Interaktion. Darin tauschen[59] beide Threads Daten über einen gemeinsamen Puffer aus, wobei die Aktivitäten des Puffers einmal im Kontext des Erzeugerthreads und einmal im Kontext des Verbraucherthreads ausgeführt werden.

Threadkommunikation

Beim Thema Threadkommunikation geht es im wesentlichen um den Datenaustausch zwischen Threads. Dass dieser Austausch auch in irgendeiner Form synchronisiert erfolgen muss, wird hier nicht weiter diskutiert. Deswegen konzentriert sich die Abbildung 3.19 auch ausschließlich auf den Datenaustauschaspekt. Darin sind die bereits bekannten Threadklassen und die noch unbekannten Datenaustauschknoten TCP-Endpunkt und Empfangspuffer dargestellt.

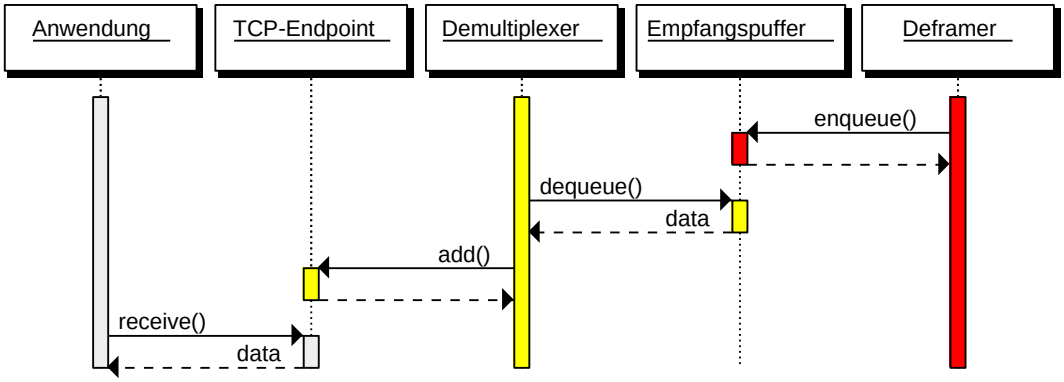


Abbildung 3.19.: Datenaustausch zwischen den Empfangsthreadklassen

Der TCP-Endpunkt ist Teil der TCP-Komponente des Netzwerkstacks und beherbergt einen Großteil der TCP-Funktionalität. Er besitzt einen Empfangspuffer, der im Threadkontext des Demultiplexers befüllt und im Threadkontext der Anwendung entleert wird. Die Betonung liegt hierbei auf Threadkontext, denn der Demultiplexer befüllt den TCP-Endpunkt natürlich nicht direkt, sondern nur indirekt über mehrere dazwischenliegende Komponenten (beispielsweise die IP-Komponente). Hingegen wird der abgebildete Empfangspuffer – im jeweiligen Threadkontext – direkt vom Demultiplexer entleert und vom Deframer befüllt.

Für die Abbildung 3.19 wurde die Annahme getroffen, dass beide Datenaustauschknoten stets Daten enthalten. Andernfalls würde der erste Leseversuch in einer Blockierung des Aufruferthreads resultieren. Das Sequenzdiagramm verschweigt auch, woher der Deframer seine Daten bekommt, was im nachfolgenden Abschnitt „Datenverarbeitung” (64) nachgereicht wird.

[59] In diesem Fall ist es ein einseitiger Tauschvorgang.

Datenverarbeitung

In der Abbildung 3.20 wird der Pfad eines empfangenen TCP/IP-Pakets durch die einzelnen Netzwerkstackkomponenten visualisiert. Dabei wird die gewohnte Schichtenreihenfolge beibehalten, so dass der Abstraktionsgrad – beginnend bei der Anwendungsschicht – von links nach rechts abnimmt, bis als letztes schließlich die Treiberebene (`sis900`) erreicht wird. Der hier diskutierte Datenpfad erstreckt sich somit diagonal von rechts oben nach links unten.

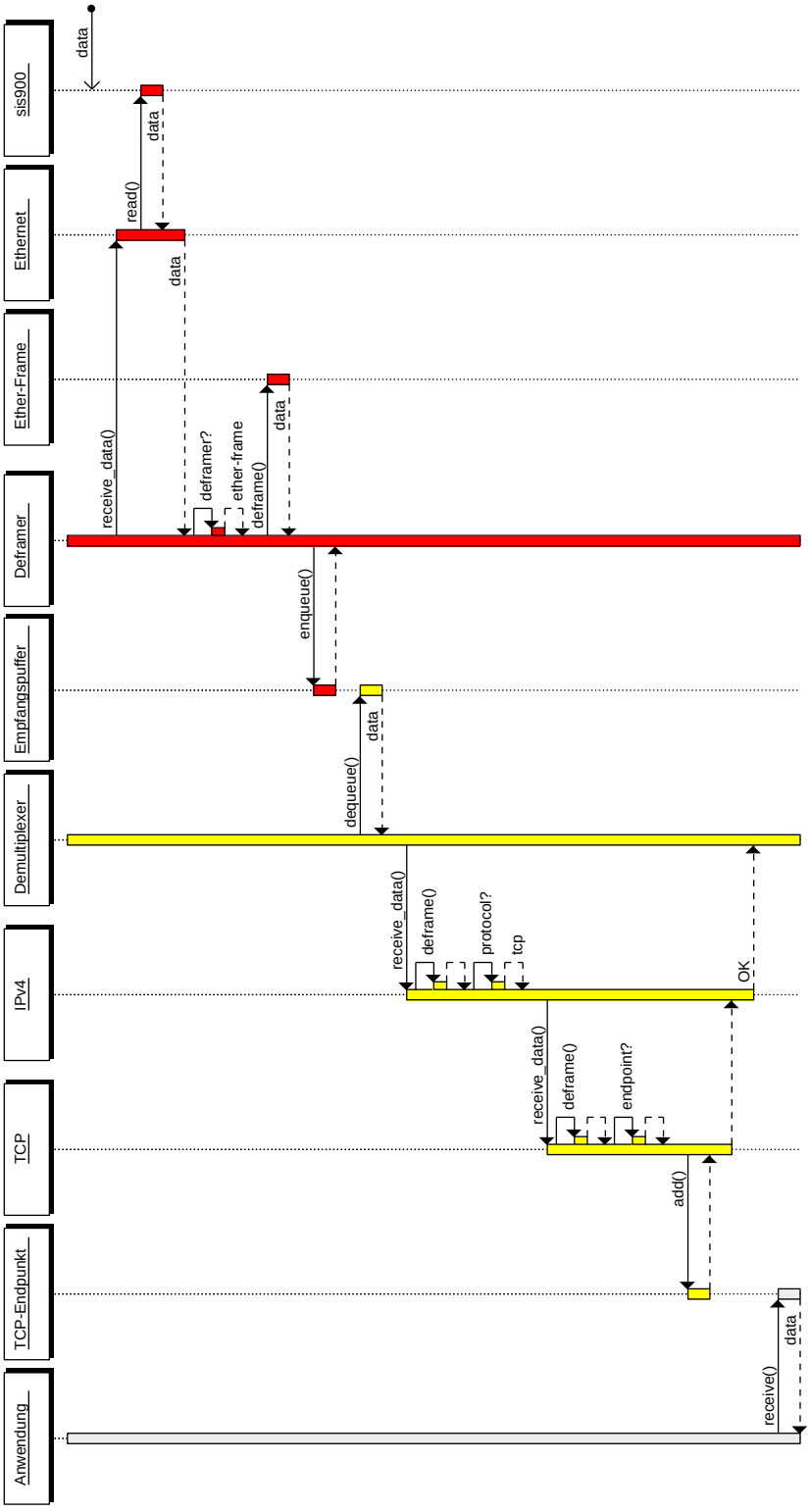


Abbildung 3.20.: Verarbeitungspfad eines empfangenen TCP/IP-Paketes

Zur Reduktion der Abbildungs- und Diskussionskomplexität, wurden die Randbedingungen so gesetzt, dass ein Lesezugriff auf einen Datenaustauschknoten nie blockiert. Es sind also stets Daten im jeweiligen Knoten vorhanden.

Die Treiberebene stellt den Ausgangspunkt für den Datenempfang dar. Die Hardware informiert den Treiber[60] über die Ankunft eines neuen Datenpakets (Frame). Dieses wird anschließend direkt vom Deframer unter Verwendung der Geräte[61]- und Treiberschicht[62] abgerufen. Der Deframer ermittelt nun die Stackkomponente, welche die Zuständigkeit für den empfangenen Frametyp angemeldet hat (nicht dargestellt) und veranlasst das Entfernen des netzwerktechnologiespezifischen Frames. Abschließend übergibt der Deframer das bearbeitete Datenpaket an den Empfangspuffer.

Der Demultiplexer entnimmt die Daten aus dem Empfangspuffer in der selben Reihenfolge, wie diese vom Deframer hinzugefügt wurden. Anschließend fragt er alle bei ihm registrierten Netzwerkprotokolle ab (nicht dargestellt), ob sie sich für das Datenpaket zuständig fühlen. Im vorliegenden Fall übernimmt die IPv4-Komponente die weitere Verantwortung für das Datenpaket, was durch den Rückgabewert `OK` an den Demultiplexer mitgeteilt wird.

Die IPv4-Komponente entfernt nun zunächst den IP-Header. Danach ermittelt sie – unter Verwendung der Headerinformation – aus allen bei ihr registrierten Netzwerkprotokollen die nächste Komponente in der Empfangskette. Dabei hat sich das IP-Header-bereinigte Datenpaket als ein TCP-Paket herausgestellt, weswegen es nun von der TCP-Komponente weiterverarbeitet wird.

Die TCP-Komponente entfernt nun ebenfalls den eigenen Protokollheader und verwendet anschließend die darin enthaltenen Informationen zur Bestimmung des zuständigen TCP-Endpunktes. Dabei existiert pro TCP-Verbindung genau ein TCP-Endpunkt, welcher dann das Header-bereinigte Datenpaket überreicht bekommt.

Als letztes Glied in der Verarbeitungskette steht nun nur noch die Applikation, welche die Daten mittels des `recv()`-Befehls über die Socketschnittstelle (nicht eingezeichnet) beim TCP-Endpunkt abholt.

3.4. FreeBSD-Kompatibilitätsschicht

Diese Schicht erlaubt es Ethernet-Netzwerktreiber von FreeBSD direkt unter Haiku zu verwenden. Die Kompatibilität ist dabei einerseits auf die Quellcodeebene bezogen, so dass ein FreeBSD-Treiber lediglich neu kompiliert werden braucht, ohne dabei Änderungen an seinem Quellcode vornehmen zu müssen. Andererseits bezieht sich die Kompatibilität aber auch auf die darüberliegende Geräteschicht, für die der FreeBSD-Treiber so erscheint, als wäre er ein nativer Haiku-Treiber.

[60] In Form eines Interrupts.
[61] `receive_data()`
[62] `read()`

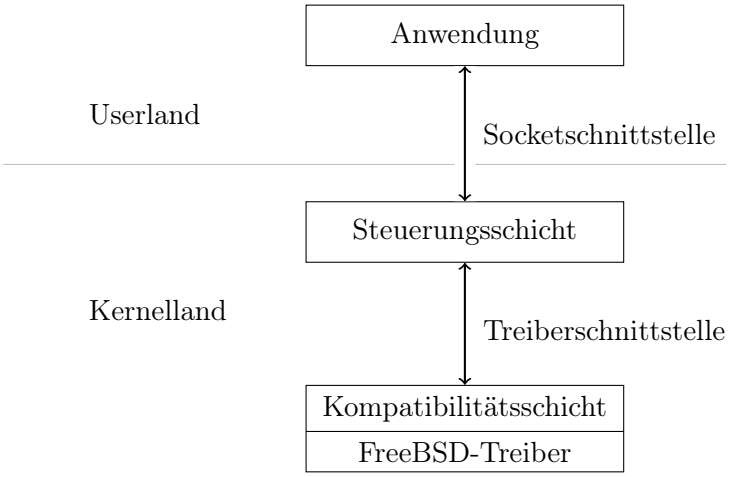


Abbildung 3.21.: Einordnung der FreeBSD-Kompatibilitätsschicht in Haikus Netzwerkarchitektur

Wie in Abbildung 3.21 zu sehen, ist die Kompatibilitätsschicht innerhalb der Treiberschicht von Haikus Netzwerkstackarchitektur[63] einzuordnen. Dort fungiert sie als Übersetzer zwischen der Treiberschnittstelle von Haiku und derjenigen von FreeBSD. Die Kompatibilitätsschicht ist dabei lediglich eine logische Konzeption. Sie stellt also keine eigenständige Netzwerkstackkomponente dar. Stattdessen wird sie in Form der statischen Bibliothek libfreebsd_network.a direkt mit dem jeweiligen FreeBSD-Treiber zu einer einzigen Binärdatei gelinkt.

Diese Umsetzung der Kompatibilitätsschicht hat den Vorteil, dass der Netzwerkstack unverändert bleiben kann. Es muss also keine neue Schnittstelle eingeführt werden und es müssen keine bestehenden Schnittstellen angepasst werden. Diese transparente Implementierungsvariante hat allerdings auch die Nachteile, dass die Treiber mehr Speicher[64] verbrauchen und dass eine Änderung an der Kompatibilitätsschicht zwangsweise eine Neukompilierung aller FreeBSD-Treiber nach sich zieht.

3.5. Portierung des FreeBSD-WLAN-Stacks

Wie bereits erwähnt, besaß Haiku noch keinen WLAN-Stack zu Beginn dieser Masterarbeit. Dies war schließlich auch einer der Gründe einen objektorientierten WLAN-Stack für Haiku zu entwickeln. Einen WLAN-Stack von Grund auf neu zu entwerfen, hat sich schnell als schwierig herausgestellt, da allein der IEEE Std 802.11 schon aus weit über 1000 Seiten besteht. Und in wie weit ein solcher Stack dann auch den Anforderungen stand hält, ist bei Einhaltung des IEEE Std 802.11 auch noch nicht sicher.

[63] Abschnitt 3.3.1
[64] In Desktoprechnern vernachlässigbar.

Das Portieren eines bereits erprobten und bewährten WLAN-Stacks schien daher die beste Variante, um gleich mehrere Hürden überwinden zu können. Es ist eine intensive Auseinandersetzung mit dem IEEE Std 802.11, mit dem Kernel, dem Netzwerkstack und der FreeBSD-Kompatibilitätsschicht von Haiku sowie der existierenden WLAN-Hardware notwendig, damit eine solche Portierung gelingt. Dabei konnten gleichzeitig auch die Nachteile der strukturorientierten Implementierung des FreeBSD-WLAN-Stacks herausgearbeitet werden, welche ja in dieser Arbeit durch Einsatz moderner objektorientierter Konzepte behoben werden sollen. Als Ausgangspunkt für die Portierung wurde dabei der WLAN-Stack von FreeBSD 8 verwendet.

Als wesentliche Nachteile haben sich dabei die geringen Parallelitäten zum IEEE Std 802.11 sowie die Funktionalitätenüberfrachtung einzelner Funktionen herauskristallisiert. So ist zum Beispiel die, im IEEE Std 802.11 beschriebene, Managementkomponente im FreeBSD-WLAN-Stack nicht klar zuordenbar.

Der FreeBSD-WLAN-Stack arbeitet auf der selben Netzwerkschichtebene wie die Kompatibilitätsschicht (Abschnitt 3.4) und stellt somit ein Ergänzung dieser Schicht dar. Im Folgenden wird kurz dargestellt, wie der WLAN-Stack bei der Portierung zur Kompatibilitätsschicht hinzugefügt worden ist.

1. **Den Stack in einen kompilierfähigen Zustand überführen.**

   In diesem Schritt werden fehlende Funktionen der Kompatibilitätsschicht deutlich, da der Kompiler deren Abwesenheit meldet. Die fehlenden Funktionen werden schließlich im FreeBSD 8 Quellcode lokalisiert und als Funktionsprototypen in die entsprechenden Headerdateien der Kompatibilitätsschicht eingefügt.

2. **Den Stack in einen linkfähigen Zustand überführen.**

   Nachdem sich die einzelnen Quelldateien des Stacks kompilieren lassen, gilt es, die so erstellten Objektdateien, zu einer ausführbaren Gesamtdatei zu verbinden. Der Fachbegriff dafür ist linken[65] und wird von dem Linker `ld` übernommen. Der Linker beanstandet schließlich fehlende Funktionsimplementierungen der Kompatibilitätsschicht, welche dann entweder per „Kopieren und Einfügen" vom entsprechenden FreeBSD 8 Quellcode oder per Haiku-spezifischer Neuimplementierung umgesetzt werden.

   Am Ende dieses Schrittes ergibt sich eine libfreebsd_wlan.a-gennannte, statische Bibliothek[66], welche gemeinsam mit der Bibliothek libfreebsd_network.a[67] die erweiterte WLAN-fähige Kompatibilitätsschicht bildet.

3. **Den Stack in einen lauffähigen Zustand überführen.**

   Damit die so erstellte Bibliothek libfreebsd_wlan.a auch ausführbar ist, muss sie von einem WLAN-Treiber genutzt werden. Dazu muss zunächst ein FreeBSD-

[65] von Englisch: to link – verknüpfen, verbinden

[66] Statische Bibliotheken werden bereits zur Kompilierzeit in ein Programm/Treiber eingebunden. Im Gegensatz dazu werden dynamische Bibliotheken erst während der Laufzeit eines Programmes/Treibers eingebunden.

[67] Enthält FreeBSD-Netzwerkkompatibilitätscode; Abschnitt 3.4.

WLAN-Treiber[68] portiert werden, welches auch entsprechend der beiden vorhergehenden Schritte erfolgt.

Anschließenden kann die neue, erweiterte Kompatibilitätsschicht und der WLAN-Treiber gelinkt werden. Dies wird vom selben Linker wie im vorhergehenden Schritt durchgeführt, weswegen hier genauso neue Beanstandungen von fehlenden Funktionsimplementierungen auftauchen können, welche durch selbige Methoden nachzuholen sind.

4. **Den Stack ausführlich testen.**

   Dieser Schritt konzentriert sich auf die Fehlersuche und -behebung. Dabei kommen verschiedene Debugging- und Testszenarien zum Einsatz. Zunächst wird damit begonnen, eine Verbindung zum eigenen WLAN unter Verwendung der Testhardware aufzubauen. Hilfreich ist dabei sowohl die im Systemlog[69] aufgezeichneten Debugausgaben des WLAN-Stacks und des WLAN-Treibers, as auch die Kerneldebuggerfähigkeiten von Haiku.

   Der Systemlog ist besonders dann hilfreich, wenn Haiku zwar anstandslos läuft, eine Verbindung aber dennoch nicht zustande kommt.

   Der Kerneldebugger wird meist automatisch gestartet, wobei dann unter Verwendung des Backtracekommandos die Funktionsaufrufe rekonstruiert werden können, welche zum Stillstand des Systems geführt haben.

   Nachdem es im eigenen „Labor" funktioniere und problemlos mehrere 100 MB an Daten über das Funknetzwerk transportiert werden können, gilt es eine breitere Testbasis heranzuziehen.

   Im vorliegenden Falle wurde dafür ein Aufruf in der Haikugemeinschaft gestartet, mit der Bitte den WLAN-Stack auf der eigenen Hardware zu testen und festgestellte Schwierigkeiten zurückzumelden. Für das Sammeln, Auswerten und Beheben der so entdeckten Fehler wurde ein, von der Haikugemeinschaft bereitgestellter, Bugtracker verwendet.

   Jedoch können nicht alle Fehler nur durch die Kommunikation über den Bugtracker gelöst werden. Bei manch schwierigen Fehlern (solche bei denen der Kerneldebugger notwendig war) hilft nur die direkte Kommunikation mittels IRC[70] und E-Mail, wodurch die Fehlereingrenzung wesentlich schneller erfolgen kann.

5. **Die Stack-Portierung stabilisieren.**

   Nach dem vorigen, ausgiebigen Testen existiert nun ein WLAN-Stack und ein WLAN-Treiber. Nun gilt es den Stack weiter zu stabilisieren. Schließlich soll dieser zum Vergleichen und Testen des objektorientierten WLAN-Stacks herangezogen werden, wozu das Vertrauen in seine Stabilität unerlässlich ist. Daher wurden

[68] Der Treiber für die WLAN-Chipsätze von Atheros, in FreeBSD ath genannt.
[69] Eine Datei in der unter anderem alle Kernelmeldungen gespeichert werden.
[70] Internet Relay Chat: Eine Form der textuellen Echtzeitkommunikation.

auch die restlichen 10 FreeBSD-WLAN-Treiber portiert. Wobei noch einige Fehler im Port aufgedeckt werden konnten. Seit Anfang März 2010 sind alle portierten WLAN-Treiber in den täglich erstellten Haikutestversionen enthalten und somit einem breiten (Test-)Publikum ausgesetzt.

3.6. Zusammenfassung

In diesem Kapitel wurde das Betriebssystem Haiku näher vorgestellt. Dabei wurde – ausgehend vom Aufbau und der Funktionsweise des Netzwerkstacks – die Grundlage für die Einordnung des objektorientierten WLAN-Stacks in das System gelegt. Weiterhin wurde Hintergrundwissen zur FreeBSD-Kompatibilitätsschicht und der darauf aufbauenden Portierung des FreeBSD-WLAN-Stacks bereitgestellt. Dieses Wissen erleichtert das Verständnis des in Abschnitt 4.6.4.1 vorgestellten Aufzeichnens von Testdaten.

4. Objektorientierter WLAN Stack

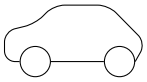
Das von IEEE Std 802.11 spezifizierte System trägt eine hohe Komplexität in sich. Diese in ein verständliches und funktionierendes Softwaresystem zu transformieren, ist eine Herausforderung.

Alle offenen Implementierungen[1] dieses Standards nehmen die Herausforderung mittels eines strukturierten Entwurfs an. Vermutlich aufgrund der Beschränkungen der verwendeten Programmiersprache[2]. Schwieriger gestaltet sich die Einschätzung bei den geschlossenen Implementierungen von Microsoft (bezogen auf Windows 7) und Apple (bezogen auf Mac OS X 10.6). Die Schnittstellen[3] der kommerziellen Stacks legen jedenfalls Nahe, dass es sich bei Microsoft ebenfalls um einen strukturorientierten bei Apple eventuell[4] sogar um einen objektorientierten Stack handelt.

Die quelloffenen Implementierungen erhöhen den Einarbeitungsaufwand, da sich die IEEE-802.11-Architektur[5] darin nicht wiederfindet. Anders ausgedrückt: Sie verwenden nicht die Sprache der IEEE-802.11-Domäne. Ein Manko, welches der vorliegende WLAN-Stack-Entwurf vermeiden möchte. In Abschnitt 4.2 wird auf diese Behauptung näher eingegangen.

Als Verbesserung dieser Situation wird hier der objektorientierte Entwurf angesehen. Dieser hat sich für die Entwicklung komplexer Softwaresysteme durchgesetzt, weswegen es nahe liegt, IEEE 802.11 ebenfalls damit zu implementieren. Dadurch können Systemkomponenten und ihr Zusammenspiel besser als mit strukturorientierten Entwürfen modelliert werden. Die Implementierung eines objektorientierten Modells mit einer objektorientierten Programmiersprache[6] mündet damit idealer Weise in eine Übereinstimmung von Theorie und Praxis. Mit anderen Worten formuliert, bleibt die Implementierung der Sprache und den Aussagen des Modells treu.

[1] BSD und Linux
[2] C ist strukturorientiert.
[3] Für Windows 7 siehe [Win7], für Mac OS X 10.6 siehe [MacOSX]
[4] Mac OS X bietet zwar eine objektorientierte Schnittstelle, jedoch gibt es Anzeichen ([Leffler], „Desktop/Laptop users: [...], Apple“, Seite 20), dass der WLAN-Stack intern den FreeBSD-Code verwendet und nur die Schnittstelle objektorientiert gehalten ist.
[5] Abschnitt 2.3 und Abschnitt 2.4
[6] C++ in dieser Masterarbeit



[a]Die Räder auf der gegenüberliegenden Autoseite sind verdeckt.

Abbildung 4.1.: Seitenansicht eines Autos

Der modellgetriebene Entwurf bietet eine Regelsammlung, welche bei der Annäherung an dieses Ideal hilft. Dabei wird von Anfang darauf geachtet, dass Modell und Implementierung übereinstimmen. Jedoch wird zusätzlich zum rein objektorientierten Entwurf auch noch ein großer Wert auf das durchgängige Verwenden der Sprache des Anwendungsbereichs (der Domäne des Softwaresystems) gelegt. Jedoch ist die hier vorgelegte Arbeit keine Demonstration einer modellgetriebenen Softwareentwicklung. Dies würde die automatische Generierung des Quellcodes aus dem Systemmodell erfordern, was im Rahmen dieser Arbeit zu weit geht[7]. Die Übereinstimmung von Modell und Implementierung wird also „per Hand“ realisiert. Ein Umstand der von [Modell, Seite 3], in Abgrenzung zum modellgetriebenen, auch als modellbasierter Entwicklungsprozess bezeichnet wird.

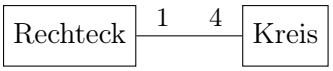


Abbildung 4.2.: UML-Notation eines Autos, wie es beim objektorientierten Entwurf abstrahiert werden darf

Die (stark vereinfachte) Zeichnung eines Autos (Abbildung 4.1) könnte im objektorientierten Entwurf wie in der Abbildung 4.2 repräsentiert werden. Zur Veranschaulichung der Gefahren des objektorientierten Entwurfs, ist der Abstraktionsgrad für das Chassis und die Räder dabei stark übertrieben gewählt. Das Auto ist nicht mehr zu erkennen. Es ist, als müsste man eine neue Sprache lernen und das liegt nicht an der eingesetzten UML-Notation. Bei bloßer Betrachtung des daraus generierten Quellcodegerüsts (Algorithmus 4.1) ist unersichtlich, dass es sich um die Repräsentation eines Autos handeln soll.

[7]Der Mangel an Werkzeugen für die automatische Generierung unter Haiku ist hier der ausschlaggebende Punkt.

Algorithmus 4.1 Implementierungsvariante des objektorientierten Autoentwurfs mit C++

```
class Rechteck {
        Kreis kreis1;
        Kreis kreis2;
        Kreis kreis3;
        Kreis kreis4;
};

class Kreis {
        Rechteck rechteck;
};
```

Viel verständlicher erscheint hingegen das domänenbeachtende Automodell in der Abbildung 4.3. Es ist sofort klar, welches Autoteil durch welche Modellkomponente repräsentiert wird, obwohl die gleichen geometrischen Formen wie in Abbildung 4.2 verwendet werden!

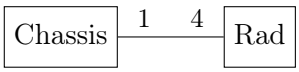


Abbildung 4.3.: Das Automodell in UML nach Verwendung des modellbasierten Entwurfs

Greift man sich in diesem Modell die Komponente Rad heraus, dann erkennt jeder Autoliebhaber sofort, welche Eigenschaften und Fähigkeiten (Informatiker sagen Attribute und Funktionen) diese Komponente besitzen muss. Dass der daraus resultierende Quellcode ebenfalls verständlicher ist, demonstriert Algorithmus 4.2. Es findet also eine geschickte Vereinigung von Anwendungsbereich, Modell und Implementierung statt. Dies sind natürlich keine neuen Erkenntnisse dieser Arbeit, sondern vielmehr die Bestandteile eines modellgetriebenen Entwurfs, wie sie in [Modell] ausführlich behandelt werden.

Algorithmus 4.2 Implementierungsvariante des modellbasierten Autoentwurfs mit C++

```
class Chassis {
        Rad Rad1;
        Rad Rad2;
        Rad Rad3;
        Rad Rad4;
};

class Rad {
        Chassis chassis;
};
```

Die Anwendung der damit verbundenen Entwurfsprinzipien auf das WLAN-Stack-Design wird in Abschnitt 4.1 vorgestellt. Anschließend wird das daraus resultierende Systemmodell in Abschnitt 4.2 beschrieben. Mit einer darauffolgenden, differenzierteren Betrachtung der einzelnen Systemkomponenten erhält der Leser in Abschnitt 4.3 schließlich ein tieferes Verständnis für den Aufbau des Stacks. Die Klassendiagramme in Abschnitt 4.4 stellen schließlich die letzte und detaillierteste Stufe der Entwurfsphase dar.

Abrundend wird im Abschnitt 4.5 die Einbindung des erstellten Entwurfs in die Betriebssystemumgebung behandelt und in Abschnitt 4.6 ein Plan zur Überprüfung der Gültigkeit des Entwurfs diskutiert.

4.1. Entwurfsprinzipien

Der bloße Einsatz von objektorientierten Entwurfsmethoden führt noch nicht zu einem verständlichen (beziehungsweise funktionierendem) Softwaresystem. Sie stellen viel mehr die Werkzeuge des Entwicklers dar, welche er verwendet, um sein Werkstück zu formen. Der effektive Umgang mit ihnen Bedarf Erfahrung und einer genauen Vorstellung vom Endprodukt.

Die für den WLAN-Stack-Entwurf maßgeblich eingesetzten Entwurfsprinzipien beruhen somit auf den Erfahrungen vieler Entwickler und Forscher. Wo die Vorstellung vom Endprodukt noch schwammig ist, bedarf es zusätzlich einer eingehenden Systemanalyse, welche im vorliegenden Fall jedoch überflüssig ist. Das zu erstellende Werkstück ist schließlich durch den IEEE Std 802.11 bereits sehr genau spezifiziert.

Damit die Menge an verwendeten Entwurfsprinzipien überschaubar bleibt, konzentriert sich der Entwurf auf vier Kriterien:

1. Verständlichkeit[8]

[8] Wird in [SoftEng, Seite 220] „Understandability“ genannt.

2. Anpassbarkeit[9]
3. Testbarkeit
4. Geschwindigkeit

Diese Reihenfolge spiegelt gleichzeitig die Priorisierung ihrer Anwendung wider. Den höchsten Stellenwert beim WLAN-Stack-Entwurf nimmt somit die Verständlichkeit ein. In den nachfolgenden Abschnitten wird diese, genauso wie die anderen drei Prinzipien, näher erläutert.

4.1.1. Qualität

Das Kriterium Qualität ist implizit in den vier Prinzipien enthalten und wird deswegen nicht extra aufgelistet. Diese Behauptung wird im Folgenden begründet. Als Grundlage für den Begriff Qualität dient hierbei die Definition in [Analyse, Seite 176], welche 5 Bewertungsdimensionen vorschlägt:

- Kopplung
- Kohäsion
- Zulänglichkeit
- Vollständigkeit
- Einfachheit

Die Verständlichkeit einer Stack-Komponente ist umso höher, je weniger sie von anderen Komponenten abhängig ist; also je weniger sie an andere Komponenten gekoppelt ist. Ebenso lässt sich eine Stack-Komponente leichter testen, wenn die **Kopplung** zu anderen Komponenten gering ist. Die Testbarkeit nimmt zu, wenn die Kopplung abnimmt. Genauso lässt sich eine Komponente leichter an neue Anforderungen anpassen, wenn dabei weniger Abhängigkeiten zu anderen Stack-Komponenten zu beachten sind. In diesem Fall steigt die Anpassbarkeit bei sinkender Kopplung.

Eine Stack-Komponente lässt sich für den Entwickler leichter verstehen, wenn alle Teilstücke dieser Komponente in einem logischen Zusammenhang zueinander stehen. Ein gutes Zeichen dafür ist eine enge Interaktion der Teilstücke untereinander. In diesem Fall spricht man von einer hohen **Kohäsion** innerhalb der Stack-Komponente.

Die Verständlichkeit einer Stack-Komponente hängt auch von ihrer **Zulänglichkeit** ab. Fehlt der Komponente eine Funktion, die vom Entwickler an dieser Stelle erwartet wird[10], so ist sie unzulänglich und besitzt eine geringe Zulänglichkeit. So sollte zum Beispiel ein WLAN-Gerät neben der Funktion Senden auch die Funktion Empfangen bereitstellen.

Eine Stack-Komponente kann ihre Verständlichkeit auch durch das andere Extrem, der überbordenden Funktionalität, verringern. Dazu gehören Funktionen, die zwar im Zusammenhang zum Komponentennamen stehen aber sich semantisch nur wenig von Anderen

[9] Wird in [SoftEng, Seite 221] „Adaptability“ genannt.
[10] Zum Beispiel weil der Name es so vermittelt.

unterscheiden. Dies wird durch den Begriff **Vollständigkeit** zum Ausdruck gebracht: Eine Komponente bietet nur die Funktionalität an, die noch nicht abgedeckt ist. Einfacher ausgedrückt: Konzentration auf das Wesentliche.

Ein schöner[11] Entwurf, der zu einer langsamen[12] Systemreaktion führt, wird Niemandem nutzen. Diesem Qualitätsaspekt widmet sich das Kriterium der **Einfachheit**. Es empfiehlt, häufig genutzte Funktionen (in [Analyse] als Primitive bezeichnet) möglichst einfach zu implementieren. Sinngemäß wird in [Analyse] gesagt, dass eine effektive Implementierung von grundlegenden Funktionen nur durch den sparsamen Einsatz von Indirektionen möglich ist.

4.1.2. Verständlichkeitsprinzip

Das Einarbeiten in ein neues Themengebiet erfordert stets Zeit, ein Gut welches im Opensource-Bereich genauso knapp und wertvoll ist wie in der Wirtschaft. Es gilt also: Je weniger Einarbeitungszeit notwendig ist, desto eher finden sich neue (Opensource-)Entwickler und desto schneller können Firmen neue Angestellte einarbeiten.

Das Verständlichkeitskriterium zielt genau auf diese Reduktion ab. Dabei beruht die Grundidee darauf, bereits erworbenes Wissen weiterzuverwenden. Ist der Entwickler erst einmal mit dem IEEE Std 802.11 vertraut, so soll er dieses Wissen idealerweise auch eins zu eins auf den WLAN-Stack übertragen können. Mit anderen Worten: Die Einarbeitungszeit fällt dann nur einmal an. Natürlich wird auch für den WLAN-Stack Einarbeitungszeit anfallen. Jedoch fällt diese gegenüber einer rein strukturorientierten Implementierung geringer aus.

Eine starke Behauptung, welche jedoch auf den Erfahrungen des domänengetriebenen Systementwurfs, wie er in [DDD] beschrieben wird, basiert. Die Kernidee dabei ist, den objektorientierten Entwurf so auszurichten, dass er die Sprache des Systemmodells (Domäne) verwendet. Die Namen für Klassen und Pakete werden dabei entsprechend der Domäne gewählt. In [Analyse, Seite 29] bringt Booch einen treffenden Vergleich aus der Luftfahrt dazu an: „Wenn der Pilot bereits weiß, wie ein bestimmtes Flugzeug zu fliegen ist, ist es für ihn viel einfacher zu erkennen, wie ein ähnliches Flugzeug geflogen wird."

Ein weiterer Ansatzpunkt zum Wiederverwenden von Entwicklerwissen ist der Einsatz von Entwurfsmustern. Diese erfreuen sich seit dem Erscheinen von [Muster] großer Beliebtheit und werden heutzutage standardmäßig in der Informatikerausbildung gelehrt. Dementsprechend wird sich der Einsatz eines Musters auch im jeweiligen Klassennamen widerspiegeln. Zusätzlich verbindet sich damit auch ein wichtiger Aha-Effekt für den Entwickler.

Der Mensch ist nach wie vor das A und O in der Entwicklung und Betreuung komplexer Software. Deswegen wird dem Verständlichkeitsprinzip eine große Bedeutung beim Ent-

[11] Im Sinne von: Die ersten vier Qualitätskriterien beachtend.

[12] Zum Beispiel werden Frames verworfen, wenn sie nicht innerhalb eines vorgegebenen Zeitfensters versendet werden.

wurf des WLAN-Stacks zugemessen. Indirekt profitieren auch die anderen drei Entwurfsprinzipien davon, denn je besser der Entwickler das System versteht, desto zuverlässiger kann er es anpassen, desto zielgerichteter kann er auf mögliche Fehler testen und desto deutlicher erschließen sich ihm Optimierungsmöglichkeiten.

4.1.3. Anpassbarkeitsprinzip

Der WLAN-Stack ist, wie alle anderen Softwaresysteme auch, einer ständigen Weiterentwicklung unterworfen. Sei es, um gefundene Fehler zu beseitigen oder die neuesten IEEE-802.11-Erweiterungen einzuarbeiten. Wie aufwändig es sich gestaltet, ein Softwaresystem zu modifizieren, wird unter dem Begriff Anpassbarkeit zusammengefasst.

Ein Softwaresystem ist leicht anpassbar, wenn sich einzelne Komponenten austauschen lassen, ohne dadurch andere Komponenten verändern zu müssen. Unter dem sicheren Gefühl das Softwaresystem funktionstüchtig zu halten, kann der Entwickler somit eine alte Komponente durch eine mit neuer Funktionalität oder Eine mit weniger Fehlern ersetzen.

Der Begriff Anpassbarkeit vereint somit die Prinzipien Änderbarkeit, Erweiterbarkeit und Wiederverwendbarkeit. Dies sind Merkmale, welche sich direkt durch die Zerlegung des Stacks in einzelne Komponenten ergeben. Eine Bestätigung dieser Behauptung findet sich mit anderer Wortwahl in [SoftArch, Seite 145].

Das Zerlegen erfolgt dabei entsprechend der in Abschnitt 4.1.1 genannten Qualitätskriterien, wobei insbesondere die Kopplung zwischen, beziehungsweise die Kohäsion innerhalb, der Module eine zentrale Rolle spielen.

Die Anpassbarkeit kann durch den Einsatz von Entwurfsmustern zusätzlich erhöht werden. Diese Muster zielen direkt auf die Wiederverwendbarkeit ab, wie es im Einführungskapitel von [Muster] zum Ausdruck gebracht wird. Dadurch erhöht sich neben der Anpassbarkeit auch die Verständlichkeit, welches im Abschnitt 4.1.2 näher erläutert wurde.

Ein anpassbares System wird auch gerne angepasst werden. Dadurch wird zum Einen das Altern des Systems im Sinne von[Balzert, Seite 1090] verhindert und zum Anderen fördert es Innovationen und Reaktionsgeschwindigkeit[13].

4.1.4. Testbarkeitsprinzip

Softwaresysteme neigen dazu, Fehler zu enthalten. Diese sind entweder von Anfang an enthalten oder werden durch Anpassungen hinzugefügt. Das Testen der Software auf Fehler ist daher von großer Bedeutung. Wie leicht oder schwer ein System für Tests geeignet ist, wird in der Testbarkeit ausgedrückt.

[13] In kurzer Zeit auf Konkurrenzprodukte oder Kundenwünsche reagieren zu können, ist wirtschaftlich gesehen von Vorteil.

Testen sollte leicht sein. Doch wann ist der WLAN-Stack leicht und wann schwer testbar? Je komplexer die Schnittstellen, desto schwieriger gestaltet sich auch das Testen ihrer Implementierungen, antwortet [Clean] sinngemäß darauf. Gemeint sind damit die Schnittstellen der Stack-Komponenten, der einzelnen Klassen und der Klassenmethoden.

Beispielsweise bietet eine Methode `sendeFrame(Header, NettoDaten, IstVerschlüsselt)` eine komplexere Schnittstelle, als `sendeFrame(Frame)`. Es sind allein 8 Testfälle notwendig, nur um alle möglichen Kombinationen von NULL-Parameter-Fehlern zu überprüfen. Dem gegenüber steht ein Testfall bei `sendeFrame(Frame)`. Eine erhöhte Verständlichkeit[14] ist ein angenehmer Nebeneffekt von einfachen Schnittstellen.

Lassen sich Stack-Komponenten einzeln testen, kann die Testbarkeit weiter erhöht werden. Testfälle lassen sich dann einfacher entwickeln, wenn eine geringe Kopplung[15] zu anderen Komponenten besteht. Je geringer die Kopplung, desto weniger Testkomponenten, sogenannte Mock-ups, müssen für einen Test entwickelt werden.

Um es deutlich auszudrücken: Im Zuge der Masterarbeit werden keine Testfälle implementiert, viel mehr wird die Grundlage bereitet, das Integrieren von Tests zu erleichtern. Dabei hilft der portierte FreeBSD-WLAN-Stack genauso, wie die Beachtung des Testbarkeitsprinzips während des objektorientierten Entwurfs.

4.1.5. Geschwindigkeitsprinzip

Es gibt zeitliche Vorgaben, deren Einhaltung notwendig für einen verwendbaren Stack sind. Beispielsweise würde der Hardwareempfangspuffer überlaufen, wenn der WLAN-Stack diesen zu langsam entleert. Somit ist eine Optimierung der Verarbeitungsgeschwindigkeit erstrebenswert. Nachdem die vorhergehenden Prinzipien die Belange des Entwicklers in den Mittelpunkt stellten, werden hier nun die Belange des Anwenders thematisiert.

Doch was nützt ein schnelles System, dass regelmäßig abstürzt? Nichts, es muss erst funktionieren bevor es auf Geschwindigkeit optimiert wird. Und dazu gehört es, das System maßgeblich unter den Gesichtspunkten der ersten drei Prinzipien zu entwerfen. Somit stehen diese dem „Schneller-ist-besser"-Motto ausgleichend gegenüber.

Es ist wichtig, einen Stack zu entwerfen, der einen Kompromiss zwischen diesen Standpunkten findet. Sparsame Optimierung könnte man den hier gewählten Kompromiss nennen. Anders ausgedrückt: Der Entwurf entscheidet sich hauptsächlich für die ersten drei Prinzipien. Müssen diese bei bestehendem Optimierungspotential nicht über Bord geworfen werden, so wird die schnellere Variante gewählt.

Je länger der Weg zum Ziel ist, desto länger dauert dessen Begehung. Der WLAN-Stack sollte also möglichst kurze Wege bereitstellen. Lange Wege können sich während des Entwurfs durch die Verwendung vieler aufeinanderaufbauender Komponenten (Schichtung) ergeben. Jede Schicht stellt damit eine weitere Abstraktion von darunterliegender

[14] Abschnitt 4.1.2
[15] Abschnitt 4.1.1

Funktionalität dar und trägt zur Erhöhung des **Abstraktionsgrades** bei. Hier greift das Qualitätskriterium der Einfachheit[16] ein, womit letztendlich der Abstraktionsgrad reduziert wird.

Auch verzweigte Wege sorgen für eine erhöhte Verarbeitungszeit. Jede Abzweigung erfordert eine Entscheidung und von diesen sollte der WLAN-Stack weitestgehend befreit sein. Die Verarbeitungszeit verringert sich und die Verarbeitungsgeschwindigkeit wird erhöht. Einige der in [Muster] behandelten **Entwurfsmuster** zielen direkt auf derartige Befreiungsaktionen ab und finden ihre Anwendung daher auch im Stack-Entwurf.

4.2. Systemmodell

Ein neues Themengebiet lässt sich leichter erschließen, wenn zunächst eine allgemein gehaltene Übersicht gegeben wird. Das IEEE Std 802.11 wäre dabei in einigen Punkten noch verbesserungsfähig. Beispielsweise ist die Suche nach einer Einführung in die Interaktion der einzelnen 802.11 Funktionalitäten vergeblich. Auch die mir bekannte Literatur bietet keine Abhilfe. Deswegen wird dies mit der vorliegenden Masterarbeit nachgeholt. Der Entwurf des allgemeinen Systemmodells wird dabei schrittweise in diesem Abschnitt demonstriert. Die Aufgaben der Systemkomponenten und ihre Beziehung zu IEEE Std 802.11 werden in Abschnitt 4.3 beschrieben.

Die Referenzimplementierung einer Client-/Ad-hoc-Station im Anhang C.3[17] von IEEE Std 802.11 ist vermutlich der beste Weg, das Zusammenspiel von Funktionen wie Authentisierung, Fragmentierung und Distribution-Coordination-Function zu verstehen. In Abbildung 4.4 sind dabei die Komponenten dargestellt, wie sie die Referenzimplementierung verwendet.

Zur Einordnung in die Stationenarchitektur (Abschnitt 2.3) sind in Abbildung 4.4 die Komponenten der angrenzenden Schichten angedeutet. Die Pfeile zwischen den Komponenten zeigen dabei die Datenaustauschs- und Signalisierungsbeziehungen an. Das `STA` am Namensende von Komponenten wie `MLME_STA` weist auf eine Client-/Ad-hoc-spezifische Komponente hin. Fehlt der `STA`-Zusatz, so ist die Komponente für alle drei Betriebsmodi[18] ausgelegt.

[16] Abschnitt 4.1.1
[17] Seite 836 ff.
[18] Ad-hoc, Client, Access-Control-Mode; Abschnitt 2.2

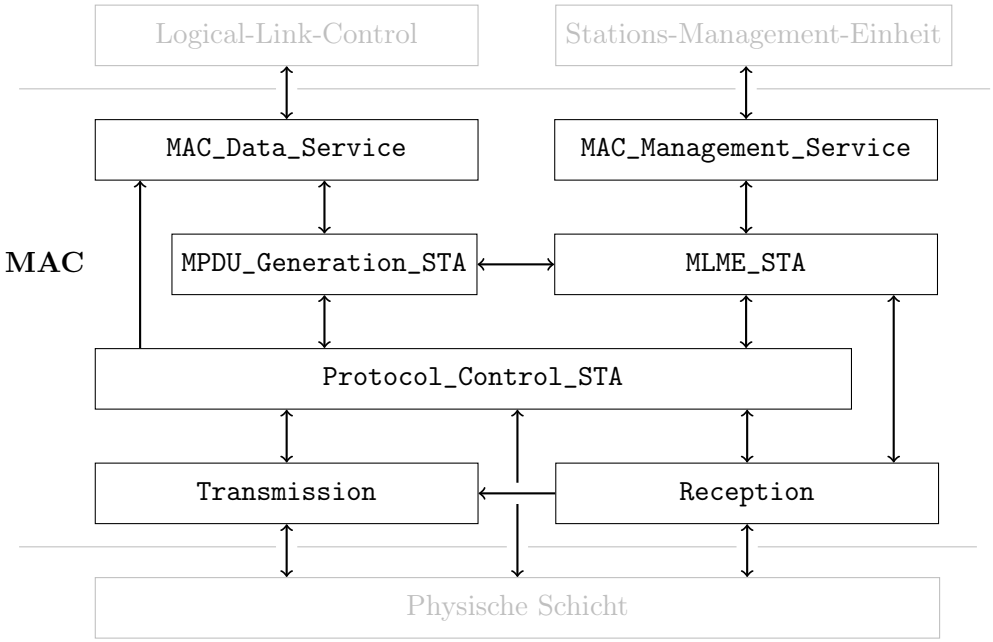


Abbildung 4.4.: Komponenten der Referenzimplementierung nach IEEE Std 802.11

Jedoch ist in der Abbildung 4.4 nicht zu erkennen, wie die IEEE-802.11-Funktionen auf die einzelnen Komponenten verteilt sind. Diese Zuordnung realisiert die Referenzimplementierung mittels Anmerkungen zu der jeweiligen Komponente, welche in Tabelle 4.1 aufgeführt sind.

Komponente	Aufgaben
`MAC_Data_Service`	Hinzufügen und Entfernen des MAC-Headers Anfrageprüfung
`MAC_Management_Service`	Management-Datenhaltung (MIB) Prüfung von `MLME_STA`-Dienstanfragen Prüfung von `MLME_STA`-Dienstrückmeldungen
`MLME_STA`	Netzwerksuche Synchronisation (Join) Authentisierung, Deauthentisierung Assoziation, Deassoziation, Reassoziation Starten eines unabhängigen Netzwerkes (IBSS) Überwachen und Steuern des Energiesparmodus'
`MPDU_Generation_STA`	Verschlüsselung Fragmentierung Duplikatsschutz Warteschlange für Energiesparmodus
`Protocol_Control_STA`	Distribution Coordination Function Controlframes
`Transmission`	Prüfsummengenerierung Zeitstempelgenerierung
`Reception`	Prüfsummenprüfung Entschlüsselung Duplikatsfilter Defragmentierung

Tabelle 4.1.: Zuordnung von IEEE-802.11-Funktionen zu Komponenten entsprechend der Referenzimplementierung in IEEE Std 802.11

Dabei fehlen aber Funktionalitäten, die seit der ersten Version von IEEE Std 802.11 hinzugekommen sind (Abschnitt 2.7). Beispielsweise fehlen die neuen Verschlüsselungsverfahren genauso wie die Quality-of-Service-Erweiterung. Deren Einarbeitung in die WLAN-Stackarchitektur ist kein Bestandteil dieser Arbeit.

Eine Überführung dieses Ausgangsmodells in die WLAN-Stackarchitektur wird im Folgenden schrittweise verdeutlicht.

4.2.1. Vereinheitlichung der Betriebsmodi

Zur Erhöhung der Verständlichkeit soll das Modell neben Ad-hoc- und Clientmodus auch den Zugriffskontrollmodus (Access-Control-Mode) behandeln können. IEEE Std 802.11[19] bietet dabei zusätzlich eine Referenzimplementierung für einen kompletten Access-Point an. Also für eine Station im Zugriffskontrollmodus mit zusätzlicher Anbindung an ein

[19] [Std80211, Anhang C.4, Seite 913 ff.]

Distributionssystem (DS). Daraus kann die Funktion einer Zugriffskontrollstation extrahiert werden, indem die auf `AP`-endenden Komponenten mit ihren `STA`-Gegenstücken verglichen werden.

Es läuft somit auf eine Funktionalitätserweiterung der Komponenten `MPDU_Generation`[20], `MLME`[21] und `Protocol_Control`[22] hinaus. Das Zwischenspeichern von Frames, welche in der wettbewerbsfreien Medienzugriffsphase (Abschnitt 2.4.7) ausgesendet werden, erweitert die `MPDU_Generation`-Komponente um eine weitere Warteschlange. Ein Infrastrukturnetzwerk zu starten, ist nun unter Verwendung der `MLME`-Komponente möglich. Das Regeln der wettbewerbsfreien Phase, fließt in die `Protocol_Control`-Komponente in Form der Point-Coordination-Function ein. Die angesprochenen Änderungen sind in Tabelle 4.2 fett hervorgehoben.

Komponente	Aufgaben
`MLME`	Netzwerksuche Synchronisation (Join) Authentisierung, Deauthentisierung Assoziation, Deassoziation, Reassoziation Starten eines unabhängigen Netzwerkes (IBSS) **Starten eines Infrastrukturnetzwerkes** Überwachen und Steuern des Energiesparmodus'
`MPDU_Generation`	Verschlüsselung Fragmentierung Warteschlange für Energiesparmodus **Warteschlange für wettbewerbsfreie Phase**
`Protocol_Control`	Distribution Coordination Function **Point Coordination Function** Controlframes

Tabelle 4.2.: Vereinheitlichung der Betriebsmodi im Systemmodell

Die Abbildung 4.4 verändert sich lediglich hinsichtlich der Komponentennamen, deren `STA`-Endung entfällt.

4.2.2. Vereinigung der Managementkomponenten

Eine Verbesserung der Verständlichkeit und Testbarkeit lässt sich durch das Zusammenfassen der `MAC_Management_Service`- und `MLME`-Komponente erreichen. Die neue `Mac-Management`-Komponente wird entsprechend Abbildung 4.5 in das Systemmodell eingefügt. Die Verwendung des Namens `Mac-Management` anstelle von `MLME` begründet

[20] vormals: `MPDU_Generation_STA`
[21] vormals: `MLME_STA`
[22] vormals: `Protocol_Control_STA`

sich durch die bessere Lesbarkeit und ist somit ein Beitrag zur Verbesserung der Verständlichkeit.

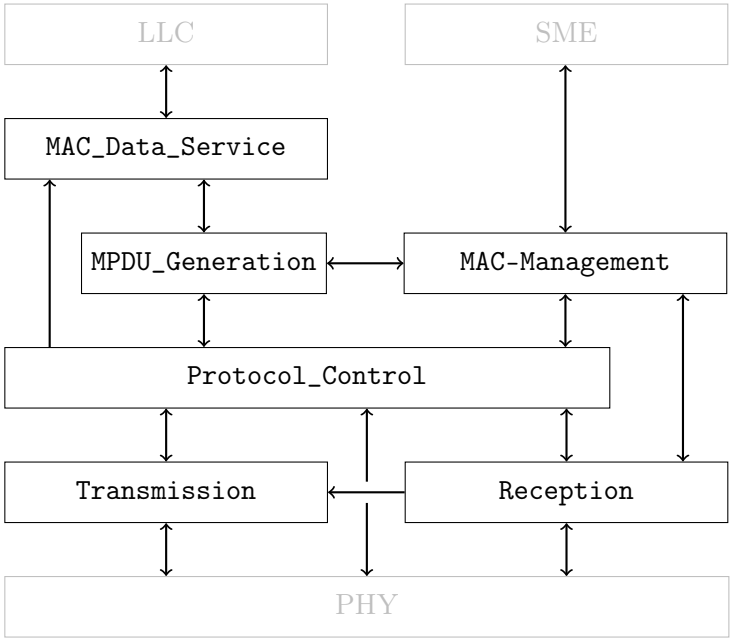


Abbildung 4.5.: Systemmodell im Anschluss an die Vereinigung der Managementkomponenten.

Desweiteren ist nun die eindeutige Zuordnung, der im IEEE Std 802.11 beschriebenen Funktionalität, zur neuen `Mac-Management`-Komponente möglich. Es entfällt der Aufwand, zwei Komponenten verstehen zu müssen und die Zuordnung von IEEE-Std-802.11-Spezifikationen zur jeweiligen Komponente durchführen zu müssen.

Statt zweier Komponenten nur noch eine Testen zu müssen, verringert den Testaufwand ebenso und steigert die Testbarkeit.

4.2.3. Funktionalitätensymmetrisierung

Der Entwickler erwartet, dass symmetrische Funktionalitäten in der selben Komponente zu finden sind. Die Aufteilung von Fragmentierung und Defragmentierung, Verschlüsselung und Entschlüsselung sowie Duplikatsschutz und Duplikatsfilter auf die Komponenten `MPDU_Generation` und `Reception` überrascht. Für die Verständlichkeit ist es daher besser, Entschlüsselung, Defragmentierung und Duplikatsfilter in die `MPDU_Generation`-Komponente zu verschieben. Der Komponentenname wird nun aber nicht mehr der enthaltenen Funktionalität gerecht. Besser beschreibt der Name `MPDU-Coordination` die neue Aufgabe.

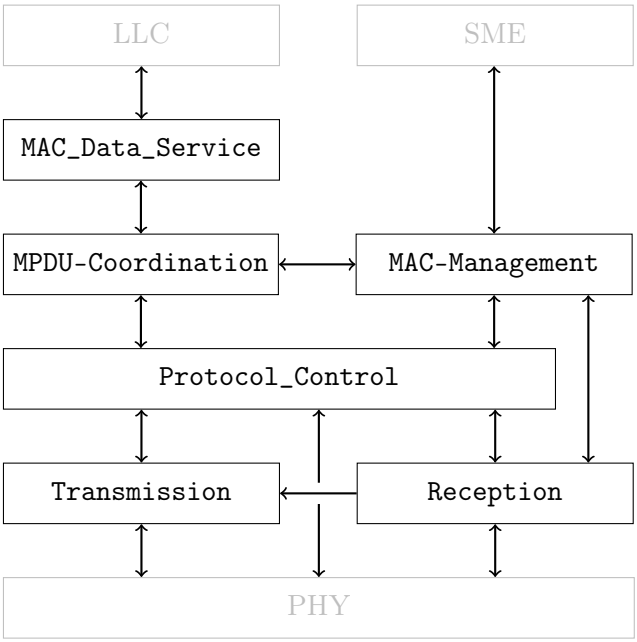


Abbildung 4.6.: Systemmodell im Anschluss an die Funktionalitätensymmetrisierung

Die Testbarkeit erhöht sich durch die Verlagerung ebenfalls, da nun die Verbindung zwichen `Reception` und `MAC_Data_Service` entfällt, welche vorher für die Auslieferung der empfangenen Datenframes an `MAC_Data_Service` zuständig war. Somit ergibt sich das symmetrisierte Systemmodell der Abbildung 4.6 und die neue Funktionenzuordnung der Tabelle 4.3.

Komponente	Aufgaben
`MPDU_Coordination`	Verschlüsselung, **Entschlüsselung** Fragmentierung, **Defragmentierung** Duplikatsschutz, **Duplikatsfilter** Warteschlange für Energiesparmodus
`Reception`	Frameprüfung

Tabelle 4.3.: Funktionenzuordnung nach erfolgter Symmetrisierung

4.2.4. Blackboxing

Die drei Komponenten `Protocol_Control`, `Transmission` und `Reception` lassen sich zu einer einzigen Komponente `Device` zusammenschließen. Diese beinhaltet dann alle Funktionen, welche in Hardware realisiert sind und somit als Blackbox behandelt werden können. Die in Abbildung 4.7 verbleibenden Verbindungen zur `Device`-Komponente dienen somit dem Datenaustausch und dem Devicemanagement.

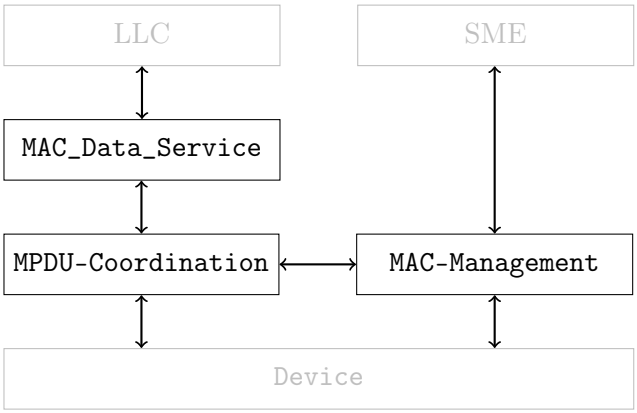


Abbildung 4.7.: Systemmodell im Anschluss an das Blackboxing

Die `Device`-Komponente wird aufgrund ihrer Blackboxnatur innerhalb dieser Masterarbeit nicht weiter berücksichtigt, sondern von dieser als gegeben vorausgesetzt. Aus diesem Grund wird das `Device` in der Abbildung 4.7 auch nur angedeutet.

4.2.5. Modellerweiterungen

Bis hierher wurde das ausgängliche Systemmodell verallgemeinert und Komponenten zusammengefasst. Es wurden Funktionalitäten hinzugefügt und neu verteilt, wobei allesamt aus den beiden Referenzimplementierungen und damit direkt dem IEEE Std 802.11 entnommen wurden.

Hingegen beruht die `Station-Management`-Komponente der Abbildung 4.8 auf anderen Quellen. Das `Station-Management` repräsentiert dabei die in IEEE Std 802.11 angedeutete aber nicht spezifizierte Stations-Management-Einheit (SME). Deren Funktionalität kann somit von der WLAN-Stack-Implementierung selbst definiert werden. Die dabei bereitgestellten Funktionen werden in Abschnitt 4.3 mit behandelt.

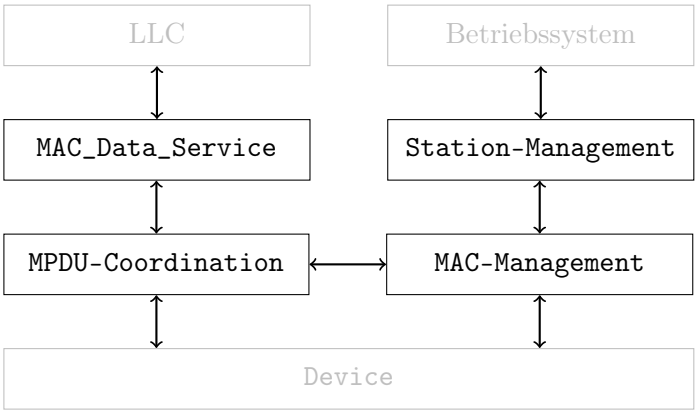


Abbildung 4.8.: Erweiterung des Systemmodells

4.2.6. Endfassung

Abschließend wird noch etwas Feinschliff an der `MAC_Data_Service`-Komponente betrieben. Der Namensbestandteil Service vermittelt Komponenteneigenschaften[23], die so nicht von IEEE Std 802.11 gefordert werden: Eine Umbenennung in `MAC-Data` berücksichtigt das und führt zur in Abbildung 4.9 dargestellten Endfassung des Systemmodells.

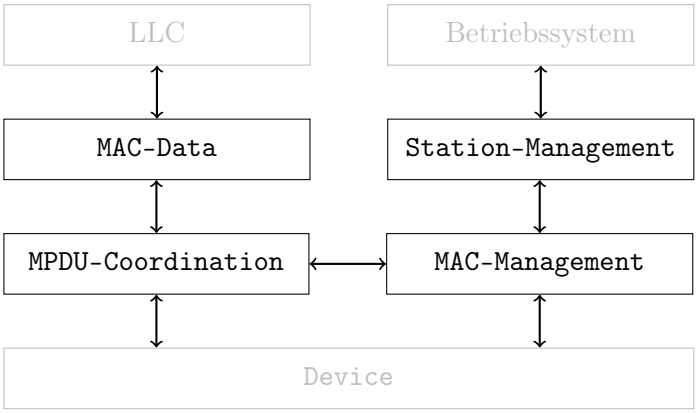


Abbildung 4.9.: Endfassung des Systemmodells

Es wurde Schritt für Schritt gezeigt, wie aus der Referenzimplementierung des IEEE Std 802.11 das Systemmodell (Domänenmodell) gewonnen wurde. Dabei hat sich eine Struktur herausgebildet, welche eine klare Zuordnung der Komponentenfunktionalitäten zum Standard ermöglicht. Diese Zuordnung wird in Abschnitt 4.3 ausführlich dargelegt.

Eine grobe Einordnung der Komponentenzuständigkeiten lässt sich bei Betrachtung der verarbeiteten Frametypen bewerkstelligen. Rund um Datenframes dreht es sich bei der

[23] Ein Service besitzt keinen Zustand.

`MAC-Data`-Komponente. Alle Managementframes werden von der `MAC-Management`-Komponente erzeugt beziehungsweise verarbeitet. Funktionen, die auf fragmentierte Frames[24] angewendet werden, finden sich in `MPDU-Coordination`. Das Senden und Empfangen von Fragmenten, sowie die dafür benötigten Controlframes, sind schließlich Aufgabe der `Device`-Komponente.

Die `Station-Management`-Komponente verarbeitet als einzige Systemkomponente keine Frames. Sie verwendet ausschließlich die von `MAC-Management` bereitgestellten primitiven Funktionen, um daraus komplexere Funktionen, wie etwa den Verbindungsaufbau zu einem Infrastrukturnetzwerk, zu formen.

4.3. Systemkomponenten

In Abschnitt 4.2 wurde die Herleitung der im Folgenden vorgestellten Systemkomponenten demonstriert und deren Aufgaben grob beschrieben. In den nächten Abschnitten werden diese Komponenten näher charakterisiert.

Das Ziel dabei ist, das Verständnis für die jeweilige Komponente zu erhöhen. Dies wird unter anderem erreicht, indem die Verbindung zu den jeweiligen Kapiteln des IEEE Std 802.11 hergestellt wird.

4.3.1. MAC-Data

In Senderichtung nimmt diese Komponente die Daten von Haikus Netzwerkstack entgegen. Anschließend verpackt sie die Daten in einen Datenframe und übergibt diesen an die `MPDU-Coordination`-Komponente.

In Empfangsrichtung nimmt `MAC-Data` die empfangenen Datenframes von der `MPDU-Coordination`-Komponente entgegen, entfernt den Frame und übergibt die Rohdaten an Haikus Netzwerkstack.

Die Aufgaben der `MAC-Data`-Komponente werden in Kapitel 6 von IEEE Std 802.11 spezifiziert.

4.3.2. Station-Management

Diese Komponente nimmt Managementanfragen von Haikus Netzwerkstack entgegen, löst deren synchrone Bearbeitung aus und liefert anschließend das Ergebnis zurück. Für die Bearbeitung der Anfragen verwendet die `Station-Management`-Komponente Funktionalitäten, welche von der `MAC-Management`-Komponente bereitgestellt werden.

Für asynchrone Ereignisse – beispielsweise Fund eines neuen Netzwerkes – wird ein Benachrichtigungsmechanismus bereitgestellt, an dem sich interessierte Komponenten registrieren können.

[24] Von IEEE Std 802.11 MPDU genannt.

Wie bereits erwähnt, existiert für die `Station-Management`-Komponente keine Spezifikation innerhalb von IEEE Std 802.11. Aus diesem Grund orientiert sich die Masterarbeit an der Managementfunktionalität des FreeBSD-WLAN-Stacks. Diese wird von FreeBSD in der Datei ieee80211_ioctl.c implementiert.

4.3.3. MAC-Management

Die `MAC-Management`-Komponente stellt das Backend zur `Station-Management`-Komponente dar. Es werden die Basisfunktionalitäten bereitgestellt, die zum Management einer WLAN-Station benötigt werden.

Ihre Aufgabe ist auch das Erzeugen und Auswerten aller Managementframes. So verarbeitet `MAC-Management` beispielsweise Authentication-Frames, welche den Beitritt zu einem Infrastrukturnetzwerk ermöglichen (Abschnitt 2.4.1.2).

Erzeugte Managementframes werden an die `MPDU-Coordination`-Komponente zur weiteren Verarbeitung übergeben. Die `MPDU-Coordination`-Komponente ist schließlich auch die Quelle für empfangene Managementframes.

Die Aufgaben der `MAC-Management`-Komponente werden in Kapitel 10 von IEEE Std 802.11 spezifiziert. Da die `MAC-Management`-Komponente auch Funktionen des PHY-Managements beansprucht ist die Zuordnung der Devicekomponentenfunktionalitäten zum IEEE Std 802.11 in Abschnitt 4.3.5 ebenfalls hilfreich.

4.3.4. MPDU-Coordination

Dies ist die zentrale Komponente für die geräteunabhängige Aufbereitung von auszusendenden und von empfangenen Frames. Dazu zählen die Ver- und Entschlüsselung sowie die Fragmentierung und Defragmentierung. Die aufbereiteten Frames werden schließlich an die Devicekomponente übergeben beziehungsweise von dieser entgegengenommen.

Weiterhin ist die `MPDU-Coordination`-Komponente auch für das Verteilen der empfangenen Frames auf die entsprechenden Komponenten zuständig. Nach erfolgter Defragmentierung und Entschlüsselung werden Datenframes schließlich an die `MAC-Data`-Komponente und Managementframes an die `MAC-Management`-Komponente übergeben.

Die Aufgaben der Ver-/Entschlüsselung sind in den Kapiteln 8.2 und 8.3, die der Fragmentierung/Defragmentierung in den Kapiteln 9.4 und 9.5 von IEEE Std 802.11 spezifiziert.

4.3.5. Device

Auch wenn diese Komponente als Blackbox behandelt wird (Abschnitt 4.2.4), ist eine Zuordnung ihrer Funktionalität zum IEEE Std 802.11 hilfreich. Dadurch kann der interessierte Entwickler schneller in Erfahrung bringen, welche Funktionen im einzelnen von der `Device`-Komponente bereitgestellt werden.

Die `Device`-Komponente umfasst die Funktionen der verschiedenen Medienzugriffsverfahren (Abschnitt 2.4.7), wie sie in den Kapiteln 9.2, 9.3 und 9.9 von IEEE Std 802.11 beschrieben sind. Diese Funktionalitäten werden von der `MPDU-Coordination`-Komponente verwendet.

Desweiteren werden auch Managementfunktionalitäten der physischen Schicht (Physical Layer) bereitgestellt, welche über die Kapitel 12 - 19 von IEEE Std 802.11 verteilt sind. Eine Auflistung, geordnet entsprechend der Übertragungstechnologie (Abschnitt 2.5) findet sich in Tabelle 4.4. Die Managementfunktionen werden von der `MAC-Management`-Komponente beansprucht.

Technologie	Kapitel
FHSS	14.4, 14.8
DSSS	15.3
Infrarot	16.4
OFDM	17.4
HR/DSSS	18.3
ERP	19.8

Tabelle 4.4.: Detaillierte Zuordnung der Managementfunktionalitäten der `Device`-Komponente zu IEEE Std 802.11

4.4. Klassendiagramm

Das Klassendiagramm besteht aus vier Hauptpaketen, welche die jeweiligen in Abschnitt 4.3 beschriebenen Systemkomponenten repräsentieren. Damit ergibt sich in Abbildung 4.10 für die Pakete das gleiche Beziehungsgeflecht, wie für die Systemkomponenten (Abbildung 4.9 auf Seite 86). Weiterhin ist in Abbildung 4.10 ein Paket `interfaces` zu sehen, welches Schnittstellespezifikationen enthält, die von den vier Hauptpaketen gemeinsam verwendet werden.

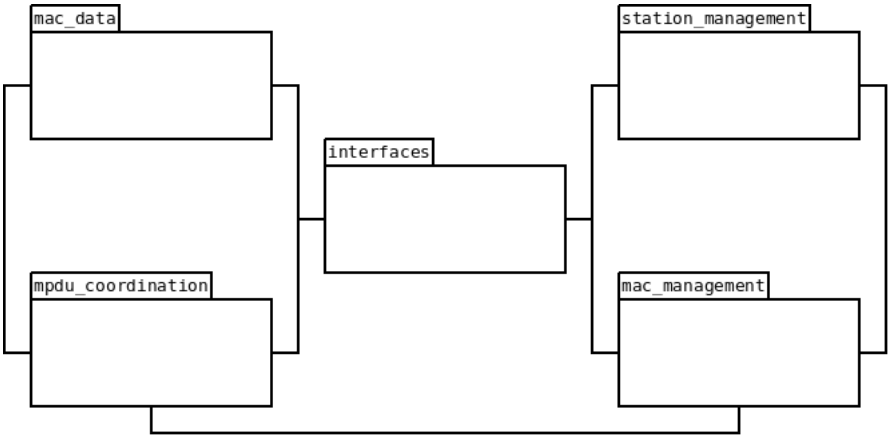


Abbildung 4.10.: Beziehungsgeflecht zwischen den Hauptpaketen

Die Klassendiagramme beschränken sich dabei auf eine Station im Clientmodus, welche weitestgehend nur die Funktionalität des originalen IEEE Std 802.11-1997 unterstützt. Dadurch halten sich die Klassendiagramme in einem überschaubaren Umfang. Die einzige Abweichung von diesem Prinzip findet sich in den Klassendiagrammen zur MPDU-Coordination-Komponente (Abschnitt 4.4.4), welche auch die aktuellen Verschlüsselungsverfahren TKIP und CCMP (Abschnitt 2.4.5) berücksichtigen.

Trotz dieser Einschränkungen ist es unmöglich ein zusammenhängendes Klassendiagramm aller Komponenten auf einer Seite unterzubringen. Daher werden die Klassendiagramme nachfolgend paketweise behandelt. Dabei ist es empfehlenswert zu erst den Abschnitt 4.4.1 zu lesen, da dieser am ausführlichsten auf die Klassendiagramme eingeht. Die Vorstellung der verbleibenden drei Hauptpakete wird weniger intensiv betrieben, da die Pakete viele – gewollte – Gemeinsamkeiten besitzen.

4.4.1. mac_data

Das Paket `mac_data` besteht aus einer Klasse `MacData::Roster` und vier weiteren Unterpaketen (Abbildung 4.11). Die Klasse `MacData::Roster` stellt dabei den zentralen Zugriffspunkt auf die Funktionalitäten der `MAC-Data`-Komponente dar. Somit ist dies eine Anwendung des Fassadenmusters, welches die Verwendung dieser Komponente erleichtert. Der Namensbestandteil `Roster` ist eine Haiku-spezifische Notation und eine andere Bezeichnung für „Manager“. Die Beschaffenheit der vier Unterpakete wird etwas später in der Abbildung 4.12 vorgestellt.

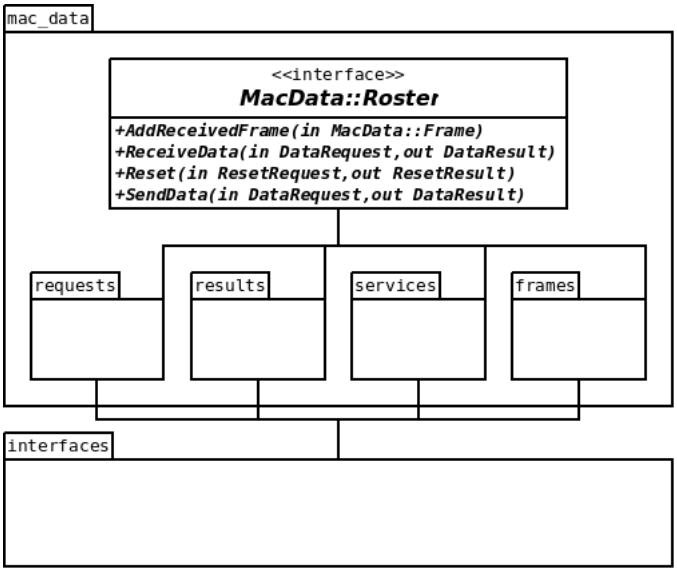


Abbildung 4.11.: Klassendiagramm des `mac_data`-Pakets

Der Zweck der Methoden, welche von `MacData::Roster` bereitgestellt werden, ist durch ihre Namensgebung selbsterklärend[25]. Die Verwendung der Methoden bedarf jedoch einer Erklärung. Im Folgenden wird dies anhand der Methode `MacData::Roster::ReceiveData()` diskutiert.

Diese Methode besitzt zwei Parameter. Jeder Parameter wird dabei für eine Datenflussrichtung verwendet. Der `MacData::DataRequest`-Parameter enthält dabei die Daten, welche vom Aufrufer an die Methode[26] übergeben werden. `MacData::DataResult` enthält schließlich das Ergebnis des Methodenaufrufs und enthält daher die Daten, welche von der Methode an den Aufrufer zurückgeliefert[27] werden. Ein Methodenparameter, der ausschießlich zur Ergebnisrückgabe verwendet wird, wird auch als Ausgabeparameter bezeichnet.

Die Verwendung eines dedizierten Ausgabeparameters verdient eine weitere Erklärung. Normalerweise werden Ergebnisse über einen speziellen Mechanismus – den Rückgabeparameter – an den Aufrufer zurückgeliefert[28]. Bei der prototypischen Implementierung des WLAN-Entwurfs hat sich jedoch gezeigt, dass es verständlicher ist, den Rückgabeparameter als Ersatz für die fehlende Exceptionsunterstützung (Abschnitt 3.2) zu verwenden. Somit kann anhand des Rückgabeparameters stets überprüft werden, ob die Methode fehlerfrei ausgeführt werden konnte oder ob ein Fehler aufgetreten ist. Die Rückgabeparameter wurden in allen Klassendiagrammen weggelassen, da sie keinen Informationsgewinn darstellen.

[25] Anwendung des Verständlichkeitprinzips
[26] Angedeutet durch die vorangestellte Annotation „`in`“.
[27] Angedeutet durch die vorangestellte Annotation „`out`“.
[28] In C++ mittels der `return`-Anweisung.

Die vier Unterpakete `requests`, `results`, `services` und `frames` (Abbildung 4.12) bündeln die von `MacData::Roster` benötigte Funktionalität in logische Einheiten. Dabei enthalten die Unterpakete `requests` und `results` die Klassen, welche für die Ein- und Ausgabeparameter verwendet werden. Im Unterpaket `services` befinden sich die Klassen, welche die Hauptarbeit verrichten. Dies wird auch durch den Namensteil „Service“ zum Ausdruck gebracht. Das Unterpaktet `frames` enthält schließlich die Klassen, welche einen objektorientierten Zugriff auf zu sendende und auf empfangene Datenframes ermöglichen.

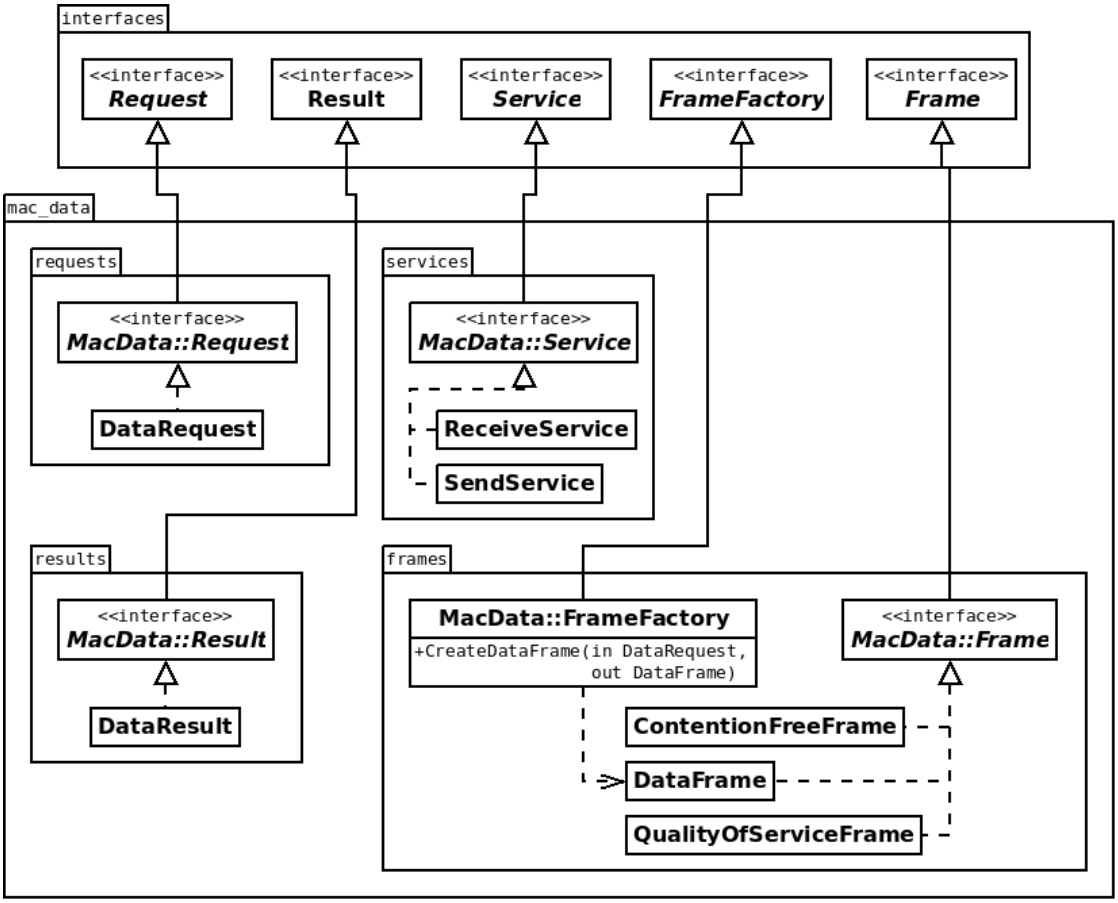


Abbildung 4.12.: Aufbau der Unterpakete von `mac_data`

Für das Erzeugen von Datenframes wird eine abgewandelte Form des Entwurfsmusters „Abstrakte Fabrik“[29] eingesetzt. Dies kommt maßgeblich der Testbarkeit der `MAC-Data`-Komponente zu gute, da nur die Konstruktionsklasse `MacData::FrameFactory` durch eine Testklasse ersetzt werden brauch und nicht sämtliche Datenframeklassen. Desweiteren ist das Hinzufügen einer neuen Methode zu `MacData::FrameFactory` leichter, als das Er-

[29] [Muster, Seite 107 ff.]

zeugen einer neuen Datenframeklasse. Dadurch wird die inkrementelle Implementierung des WLAN-Stack-Entwurfs erleichtert.

Das Unterpaket `frames` enthält zwar drei verschiedene Datenframeklassen, jedoch kann `MacData::FrameFactory` nur die Klasse `MacData::DataFrame` erzeugen[30]. Dies ist die einzige Datenframevariante, welche von einer Station im Clientmodus erzeugt werden kann. Die anderen beiden Klassen müssen dennoch vorhanden sein, da eine Station generell jeden Frametyp empfangen kann.

Abschließend sei noch bemerkt, dass der Namensbestandteil „MacData“ in den Klassendiagrammen – zur Platzersparnis – nur bei einigen ausgewählten Klassen mit eingefügt wurde. Korrekterweise müsste dies für jede Klasse im Unterpaket `mac_data` angegeben werden.

4.4.2. station_management

Das Paket `station_management` besteht aus einer Klasse `StationManagement::Roster` und zwei weiteren Unterpaketen (Abbildung 4.13). Die Klasse `StationManagement::Roster` stellt dabei den zentralen Zugriffspunkt auf die Funktionalitäten der `StationManagement`-Komponente dar. Es wird hierbei das gleiche Entwurfsmuster wie bei der MAC-Data-Komponente angewandt. Es lässt sich generell die Feststellung treffen, dass sich die Klassendiagramme der vier Hauptkomponenten des WLAN-Stack-Entwurfs ähneln. Dadurch wird die Verständlichkeit erhöht. Hat sich der Entwickler erst einmal in eine Komponente eingearbeitet, so kann er die anderen drei Komponenten leichter verstehen.

[30] Erkenntlich durch den Pfeil und die einzige Methode `MacData::FrameFactory::CreateDataFrame()`.

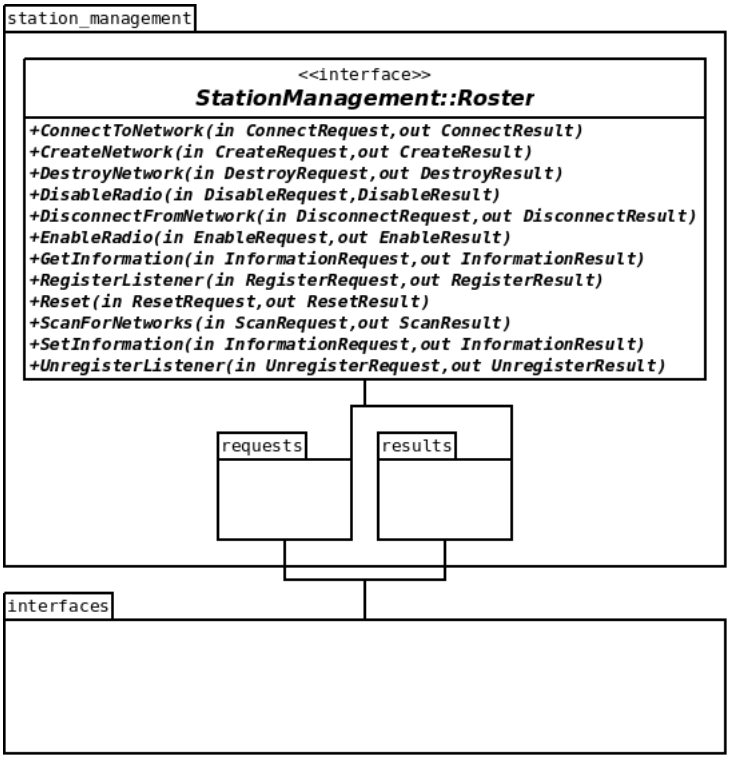


Abbildung 4.13.: Klassendiagramm des `station_management`-Pakets

Die Methode `StationManagement::Roster::CreateNetwork()` sei extra erwähnt. Sie erzeugt ein neues Drahtlosnetzwerk – wie der Name vermuten lässt – und kann damit nur von Stationen im Access-Control-Mode oder im Ad-Hoc-Modus realisiert werden. Diese Methode wird hier dennoch[31] aufgeführt, da somit zum Einen eine Modus-übergreifende, einheitliche Schnittstelle bereitgestellt wird und zum Anderen diese Methode von einer Station im Clientmodus sinnvoll implementiert werden kann. Dazu kann sie den Ausgabeparameter `StationManagement::CreateResult` dermaßen setzen, dass der Aufrufer die Aussage „Erzeugen eines Drahtlosnetzwerkes nicht unterstützt“ als Fehlermeldung auffassen kann.

Hinter der Einhaltung einer einheitlichen Schnittstelle steckt auch der Gedanke, das Startegieentwurfsmuster ([Muster, Seite 373 ff.]) einsetzen zu können. Mit diesem Muster lässt sich für jeden Modus[32] eine Modus-spezifische Implementierung realisieren. Dadurch erlangt der WLAN-Stack-Entwurf einen höheren Verständlichkeitsgrad, denn die Implementierungen der einzelnen Modi sind klar getrennt. Durch diese Trennung besteht auch dass Potenzial eine höhere Verarbeitungsgeschwindigkeit zu erzielen. Gegenüber einer Modus-vermischenden Implementierung[33] können dabei die Bedingungsabfragen nach

[31] Es wurde in Abschnitt 4.4 die Einschränkung getroffen, nur den Clientmodus zu betrachten.
[32] Ad-Hoc-Modus, Clientmodus, Access-Control-Mode
[33] Wie sie der FreeBSD-WLAN-Stack verwendet.

dem aktuellen Betriebsmodus entfallen. Mehr zur Idee des Strategieentwurfsmusters und dessen Umsetzung findet sich in Abschnitt 4.5.3 und in der darin enthaltenen Abbildung 4.22.

Die Deklaration der `StationManagement::Roster` Klasse als Schnittstelle (`<<interface>>`) ist dabei die Formalisierung einer einheitlichen Schnittstelle. Diese Deklaration findet sich auch in den vorhergehenden Klassendiagrammen des `mac_data`-Hauptpaketes, jedoch ist die Erklärung der Auswirkungen dieser Entwurfsentscheidung anhand von `StationManagement::Roster::CreateNetwork()` einfacher. Auch die beiden anderen Hauptpakete `mac_management` und `mpdu_coordination` verwenden die Schnittstellendeklaration für ihre `Roster`-Klasse aus demselben Grund.

Die Klassen der beiden Unterpakete `requests` und `results` werden in der Abbildung 4.12 dargestellt. Sie haben die gleiche logische Gliederung, wie die gleichnamigen Unterpakete von `mac_data` (Abschnitt 4.4.1) und werden daher an dieser Stelle nicht nochmals erläutert.

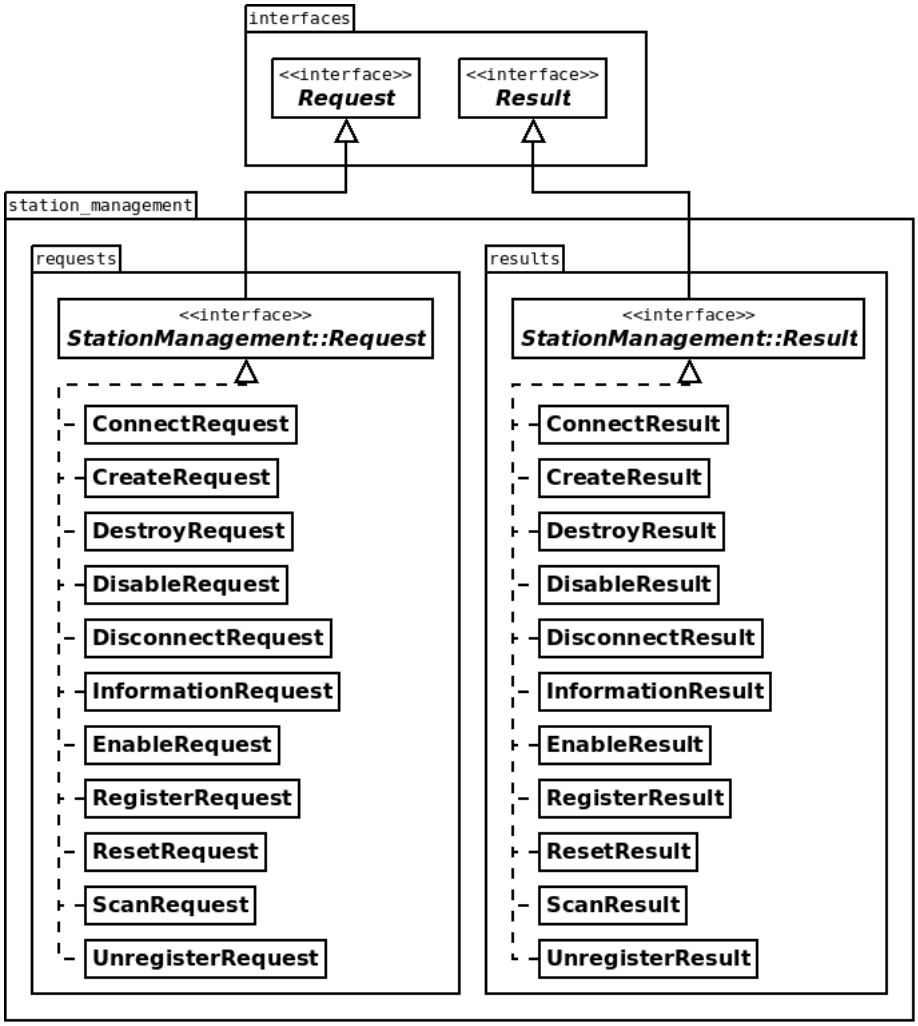


Abbildung 4.14.: Aufbau der Unterpakete von `station_management`

Der Namensbestandteil `StationManagement` wurde nur bei einigen ausgewählten Klassen mit eingefügt. Dies erfolgt aus den selben Gründen, wie beim `mac_data`-Hauptpaket (Abschnitt 4.4.1).

4.4.3. mac_management

Die Klassendiagramme des `mac_management`-Hauptpaketes (Abbildung 4.15 und 4.16) besitzen die gleiche Struktur und verwenden die selben Entwurfsmuster, wie sie im `mac_data`-Hauptpaket (Abschnitt 4.4.1) bereits diskutiert wurden.

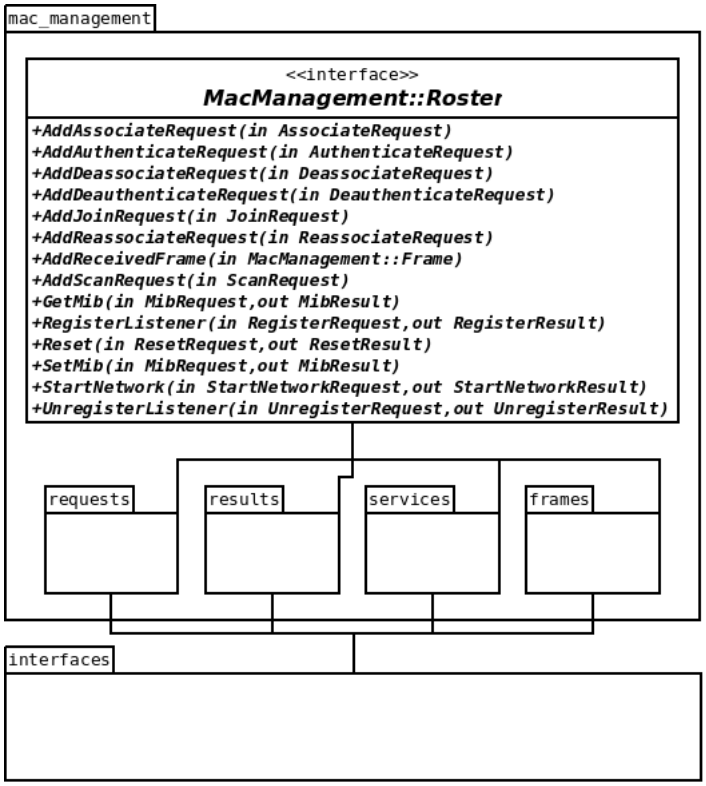


Abbildung 4.15.: Klassendiagramm des `mac_management`-Pakets

Der Namensbestandteil `MacManagement` wurde wieder nur bei einigen ausgewählten Klassen eingefügt. Die Begründung dafür wurde bereits im Abschnitt 4.4.1 beim `mac_data`-Hauptpaket gegeben.

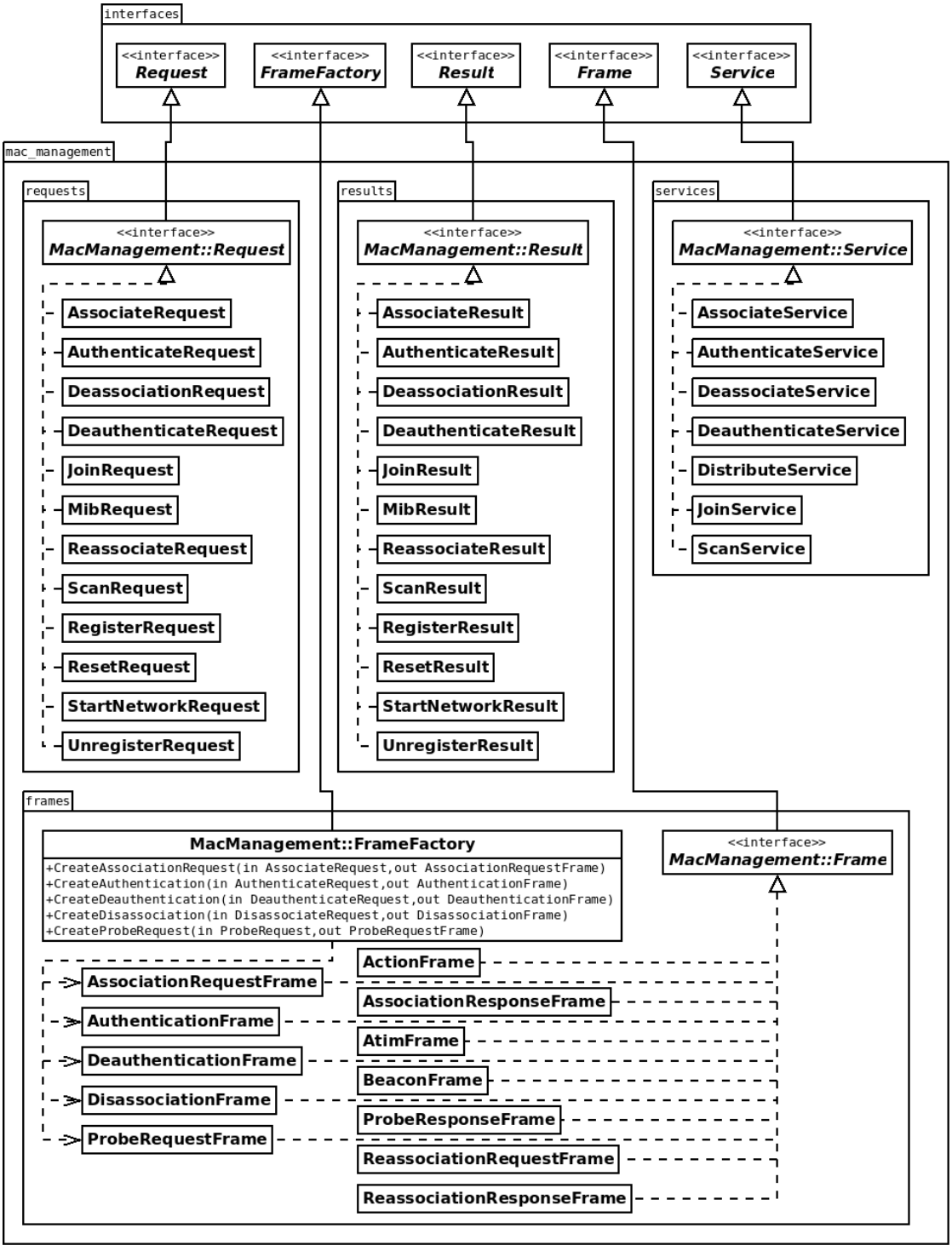


Abbildung 4.16.: Aufbau der Unterpakete von `mac_management`

4.4.4. mpdu_coordination

Das Klassendiagramm der Abbildung 4.17 enthält das Unterpaket `algorithms`, welches sich so nur in diesem Hauptpaket findet. Eine genauerer Übersicht der Bestandteile von `algorithms` findet sich in Abbildung 4.18.

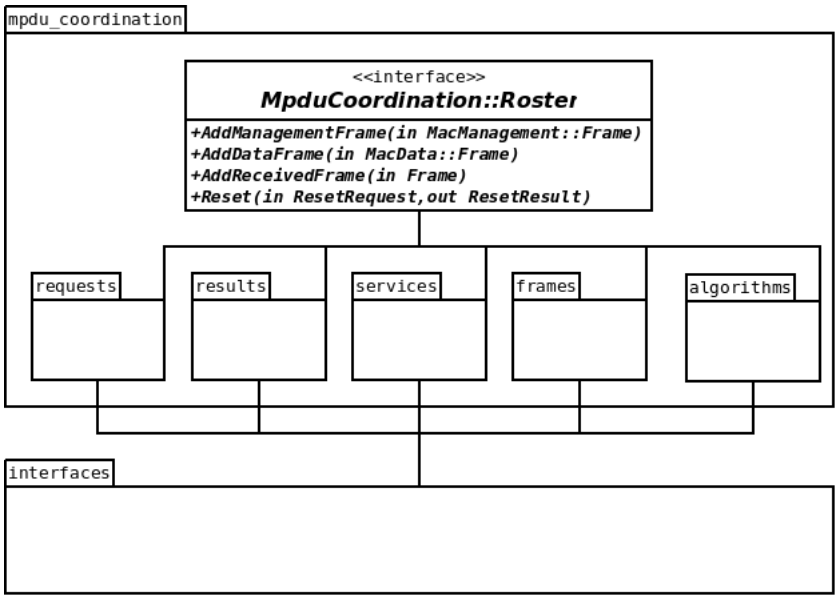


Abbildung 4.17.: Klassendiagramm des `mpdu_coordination`-Pakets

Darin werden die einzelnen Klassen deutlich. Jede Klasse implementiert dabei einen speziellen Verschlüsselungsalgorithmus (Abschnitt 2.4.5). Die Klasse `MpduCoordination::NullCipherAlgorithm` implementiert beispielsweise keinen Algorithmus. Somit wird diese Klasse immer dann verwendet, wenn keine Datenverschlüsselung benötigt wird.

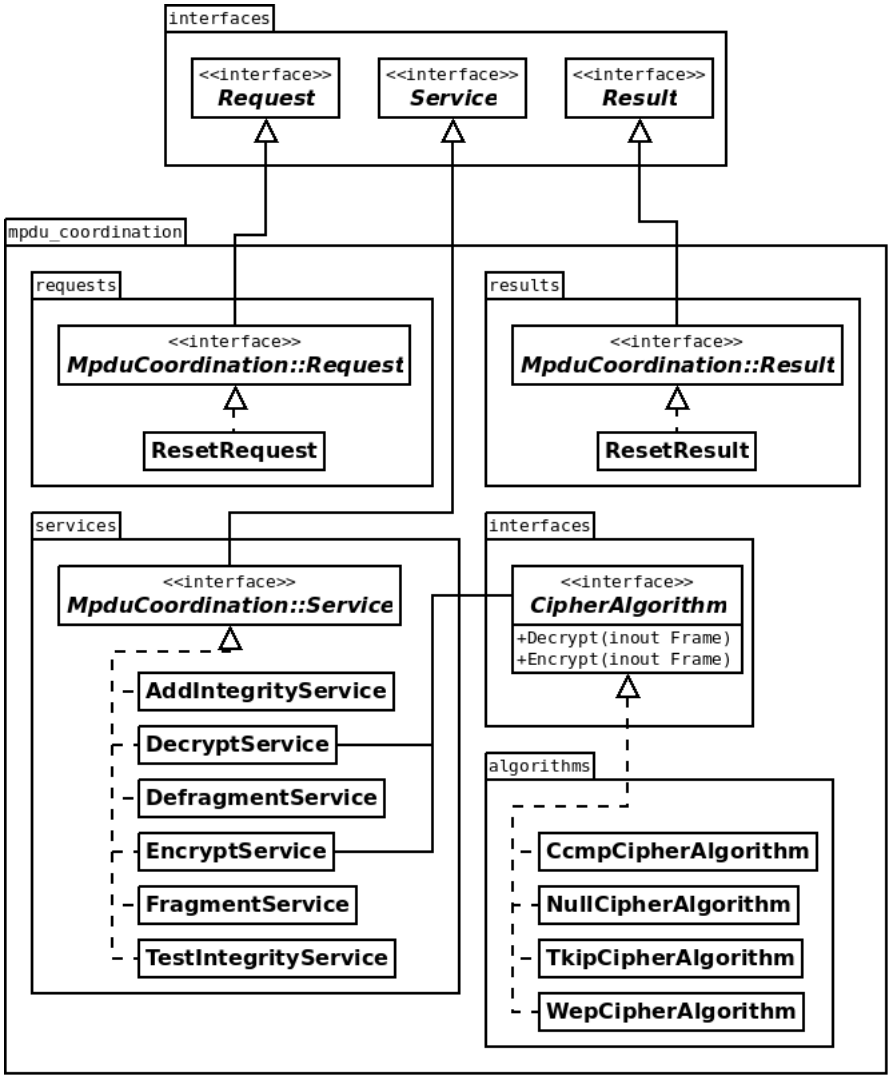


Abbildung 4.18.: Aufbau der Unterpakete von `mpdu_coordination`

Diese Verschlüsselungsklassen werden im Zusammenspiel mit den Serviceklassen aus dem `service`-Unterpaket eingesetzt. Dabei wird eine Kombination aus Schablonenmethoden- und Strategieentwurfsmuster ([Muster, Seite 366 ff. und Seite 373 ff.]) eingesetzt. Die beiden Serviceklassen `MpduCoordination::EncryptService` und `MpduCoordination::DecryptService` rufen dabei eine Methode der Klasse `MpduCoordination::CipherAlgorithm` an der richtigen Stelle im Quellcode auf (Schablonenmethodenmuster), wobei zuvor die gewünschte Verschlüsselungsklasse instanziiert wurde (Strategiemuster).

Die Kombination dieser beiden Entwurfsmuster erlaubt das einfache Hinzufügen neuer Verschlüsselungsalgorithmen. Desweiteren wird die Verständlichkeit erhöht, da jede Verschlüsselungsklasse genau einen Verschlüsselungsalgorithmus enthält. Auch die Test-

barkeit erhöht sich, da es leicht ist, eine Klasse `MpduCoordination::TestAlgorithm` anzulegen.

Der Namensbestandteil `MpduCoordination` wurde wieder nur bei einigen ausgewählten Klassen eingefügt. Die Begründung dafür wurde bereits im Abschnitt 4.4.1 gegeben.

4.4.5. Prototyp

Im Zuge der Ausarbeitung der Klassendiagramme wurden einige der Klassen in Form eines Prototypen (Anhang C) ansatzweise implementiert. Dieser Prototyp enthält weitere Ausführungen über die Bedeutung der einzelnen Klassen und ihrer Methoden, welche allerdings ausschließlich für Entwickler von Interesse sind. Aus diesem Grund sei hier nur darauf hingewiesen.

4.5. Integration in Haiku

Nachdem die Klassendiagramme des Abschnitts 4.4 einen detaillierten Überblick über den Entwurf des objektorientierten WLAN-Stacks gegeben haben, wird im Folgenden dessen Einbettung in Haiku diskutiert.

4.5.1. Ethernetimplementierung

Als Vorlage für die Integration des WLAN-Stacks wird dabei Haikus Ethernetimplementierung verwendet. Abbildung 4.19 zeigt die Komponenten, welche die Ethernettechnologie implementieren.

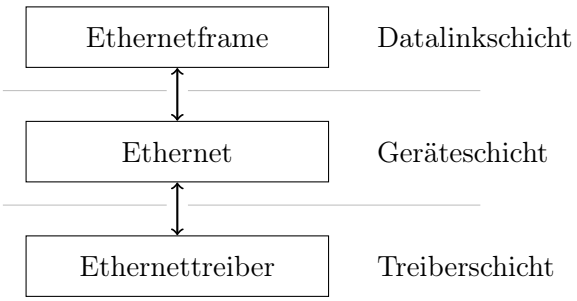


Abbildung 4.19.: Komponenten der Ethernettechnologie

Die Komponenten sind entsprechend der in Abschnitt 3.3.1 vorgestellten Schichtenarchitektur eingeordnet. Die Ethernetframe-Komponente ist dabei für das Hinzufügen beziehungsweise Entfernen der Ethernetframes zuständig. Von den vier definierten Ethernetframeformaten[34] ist diese Komponente ausschließlich für `ETHERNET_II`-Frames zuständig.

[34][Ether, Abschnitt 2.10.7 „Die verschiedenen Frametypen“, Seite 81 f.]

Die Ethernet-Komponente hat neben dem Empfangen und Senden der Frames auch die Aufgabe den jeweiligen Treiber zu verwalten. Dafür verwendet sie Ethernet-spezifische Steuerkommandos, welche in Tabelle 4.5 beispielhaft aufgelistet sind. Diese Steuerkommandos stellen damit – zusätzlich zur allgemeinen Treiberschnittstelle ([BeBook]) – eine spezielle Schnittstelle dar, welche alle Ethernettreiber implementieren müssen, damit sie vom Netzwerkstack verwendet werden können.

Kommando	Bedeutung
ETHER_GETADDR	Fordert die MAC-Adresse vom Treiber an.
ETHER_ADDMULTI	Fügt dem Treiber eine Multicast-Adresse hinzu.
ETHER_GET_LINK_STATE	Fordert Verbindungsinformationen (Geschwindigkeit, Qualität, Duplexmodus, ...) vom Treiber an.

Tabelle 4.5.: Ausgewählte Ethernetsteuerkommandos

Der Ethernet-Treiber ist schließlich für die Steuerung der zugehörigen Ethernethardware zuständig.

4.5.2. Implementierung von IEEE 802.11

In Anlehnung an die Ethernetimplementierung gliedert sich die Implementierung von IEEE 802.11 ebenfalls in drei Komponenten, welche in Abbildung 4.20 dargestellt sind.

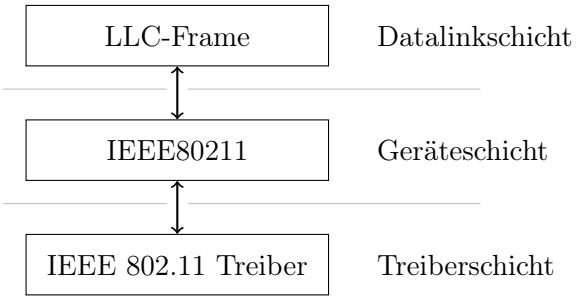


Abbildung 4.20.: Komponenten der IEEE 802.11 Technologie

Die Aufgabe der LLC[35]-Frame-Komponente besteht im Hinzufügen beziehungsweise Entfernen der LLC-Frames. Im Gegensatz zu Ethernet kennt IEEE 802.11 ausschließlich das LLC-Frame-Format. In der Terminologie von [Ether] handelt es sich dabei um das `ETHERNET_SNAP`-Format, welches ein spezielles LLC-Frameformat ist.

Für die Integration des objektorientierten WLAN-Stacks ist der genaue Aufbau dieses Frameformats irrelevant, weswegen es hier auch nicht näher vorgestellt wird. Wichtig

[35] Logical-Link-Control

zu wissen ist nur, dass die LLC-Frame-Komponente nicht von Haikus Netzwerkstack bereitgestellt wird. Bei einer Implementierung des WLAN-Stacks muss diese Komponente somit ebenfalls erst noch implementiert werden.

Die IEEE80211[36]-Komponente hat die Aufgabe, Frames zu senden und zu empfangen sowie den Treiber zu verwalten. Spezielle Kommandos, wie sie in der Ethernetkomponente zur Steuerung der Ethernettreiber verwendet werden, sind ebenfalls in dieser Komponente anzusiedeln, wurden jedoch im Rahmen dieser Masterarbeit nicht definiert.

Gleichzeitig stellt die IEE80211-Komponente auch die logische Stelle dar, innerhalb derer die Implementierung des objektorientierten WLAN-Stack-Entwurfs vorzunehmen ist. Hier werden schließlich alle netzwerktechnologiespezifischen und treiberunabhängigen Funktionalitäten gebündelt.

Der IEEE 802.11-Treiber ist schließlich für die Steuerung der zugehörigen IEEE 802.11-Hardware zuständig.

4.5.3. Integration des WLAN-Stacks in die IEEE80211-Komponente

Die Integration des objektorientierten WLAN-Stack-Entwurfs in die IEEE80211-Komponente erfordert dessen Anbindung an die prozedurorientierte Schnittstelle der Geräte- und Treiberschicht. Abbildung 4.21 veranschaulicht die drei Schichten der IEEE80211-Komponente.

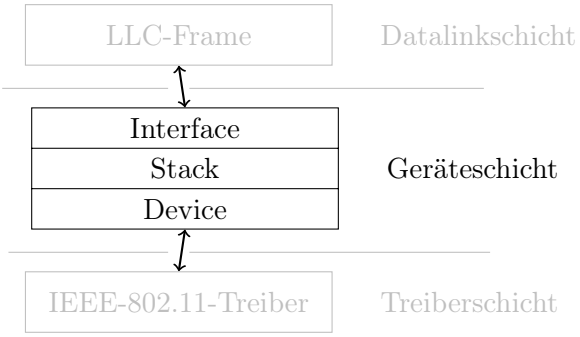


Abbildung 4.21.: Schichten der IEEE80211-Komponente

Das Interface implementiert dabei den Übergang von der prozedurorientierten Schnittstelle der Geräteschicht zur objektorientierten Schnittstelle des WLAN-Stacks (in der Abbildung 4.21 nur Stack genannt). Es bildet einen Adapter zwischen der von Haiku vorgegebenen Komponentenschnittstelle und der vom WLAN-Stack bereitgestellten Schnittstelle.

Die Stackschicht ist schließlich der Ort, an dem die Implementierung des WLAN-Stack-Entwurfs zu finden ist. Diese Schicht enthält somit rein objektorientierte Konzepte. Dabei

[36] Dies ist der Name der Komponente, so wie er auch im Quellcode verwendet wird.

kann die Stackschicht als eine Anwendung des Fassadenentwurfsmusters ([Muster, Seite 212 ff.]) betrachtet werden. Dieses Muster bietet die Vorteile, dass ein zentraler Zugriffspunkt auf die WLAN-Stack-Funktionalitäten besteht (Beachtung des Testbarkeitsprinzips[37]) und dass der Anwender nicht wissen muss, welche Komponente des objektorientierten Entwurfs er für die gewünschte Funktionalität anzusprechen hat (Beachtung des Verständlichkeitsprinzips[38]).

Weiterhin erlaubt ein zentraler Zugriffspunkt auch den transparenten Einsatz verschiedener Implementierungsstrategien. Damit lässt sich das Strategieentwurfsmuster ([Muster, Seite 373 ff.]) einsetzen, wobei jeder Stationsmodus[39] als eine eigene Strategie implementiert wird. Als Folge dessen ist es möglich den Modus einer Station im laufenden Betrieb[40] und damit ihr Verhalten zu ändern. In Abbildung 4.22 wird dieser Ansatz in Form eines Schalterdiagramms dargestellt. Der Stack ist dabei für das Umlegen des Schalters zuständig. Dabei ist auch zu sehen, dass stets nur ein Stationsmodus aktiv sein kann.

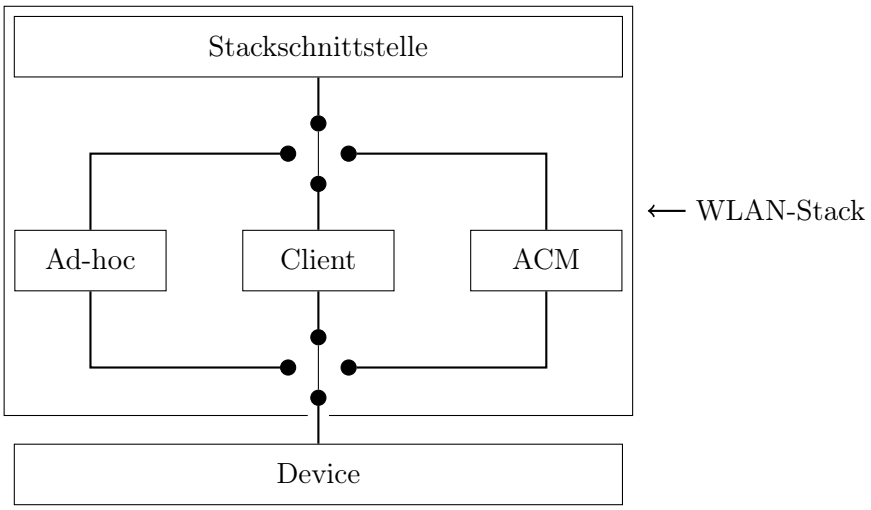


Abbildung 4.22.: Schalterdiagramm der dynamischen Modusänderung

Bei Betrachtung der Architektur einer solchen Modusstrategie (Abbildung 4.23), wird das in Abschnitt 4.2 vorgestellte objektorientierte Systemmodell ersichtlich. Die Konsequenz daraus ist, dass zwar jeder Modus seine eigene Implementierung besitzt, diese aber alle auf einem gemeinsamen Entwurf beruhen.

[37] Abschnitt 4.1.4
[38] Abschnitt 4.1.2
[39] Ad-hoc, Client, Access-Control-Mode
[40] ohne Reinitialisierung der IEEE80211-Komponente

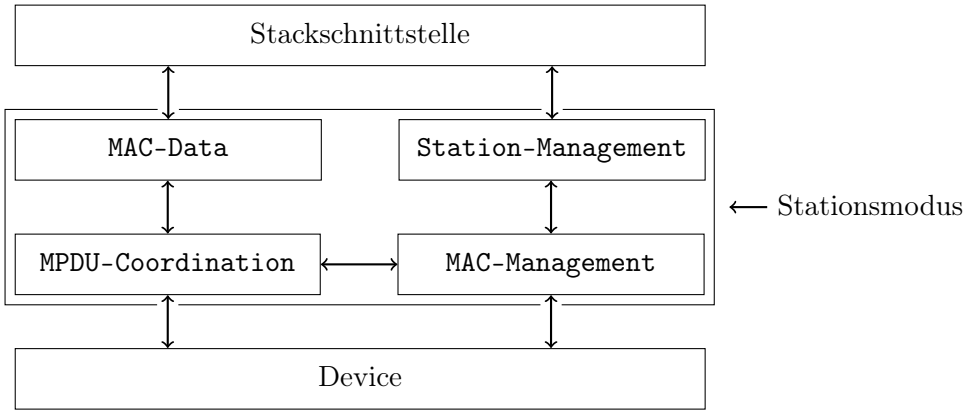


Abbildung 4.23.: Architektur einer Modusstrategie

Die Deviceschicht stellt schlussendlich die Verbindung zwischen Geräteschicht und Treiberschicht her. Sie bietet somit dem Stack eine objektorientierte Schnittstelle an und implementiert diese unter Verwendung der prozedurorientierten Schnittstelle der Treiberschicht.

Eine Teilimplementierung der hier vorgestellten IEEE80211-Komponente mit all ihren Unterkomponenten findet sich im Prototypen (Anhang C). Darin kann der interessierte Entwickler sehen, wie sich die Integration in Haiku umsetzen lässt.

4.6. Gültigkeit des Entwurfs

Nachdem der Entwurf und dessen Integration in Haiku bereits besprochen wurden, verbleibt nun noch die Frage inwieweit die vorgestellten Ideen auch gültig sind. Für die Überprüfung der Gültigkeit existieren dabei die beiden Lager der theorieorientierten und der praxisorientierten Verfahren.

Die theoretische Überprüfung beruht dabei im wesentlichen auf der Verwendung mathematischer Beweise. Der für diese Masterarbeit relevante Vorteil dieser Methode, liegt dabei in einer genauen Kontrolle der Rahmenbedingungen. Beispielsweise können Eigenheiten der Hardware ignoriert werden. Allerdings besteht hierbei auch die Gefahr, die Rahmenbedingungen an der Realität vorbei zu setzen, so dass am Ende zwar eine theoretisch korrekte Implementierung vorliegt, diese aber praktisch nicht einsetzbar ist.

Die praktische Überprüfung setzt besonders auf das Ausführen des Produktivcodes. Dabei wird der Produktivcode unter verschiedenen, wohldefinierten Szenarien (Testfälle) ausgeführt. Die dabei erzeugten Ergebnisse werden anschließend mit den Erwartungswerten verglichen und bewertet. Nur wenn alle Ergebnisse mit ihren Erwartungswerten übereinstimmen gilt der getestete Produktivcode als gültig. Der Vorteil der praktischen Überprüfung liegt vor allem darin, unter realen Bedingungen testen zu können. Dies birgt gleichzeitig aber auch den Nachteil des hohen Hardwareaufwands in sich.

Eine tatsächliche Überprüfung des objektorientierten WLAN-Stacks – über das Prototypenstadium hinaus – liegt außerhalb des Rahmens dieser Masterarbeit. Im Sinne einer Themenabrundung wird daher ausschließlich die weitere Vorgehensweise bei der Überprüfung des Entwurfs im Sinne eines Überprüfungsplans vorgestellt.

Die vorgestellten Verfahren beruhen dabei auf eigenen Überlegungen, da die mir bekannte Literatur keine Angaben über das Testen eines WLAN-Stacks enthält.

Hinweis: Nachdem das Testkonzept bereits ausgearbeitet wurde, hat sich gezeigt, dass in Linux ein ähnlicher Ansatz verwendet wird. Auch dort wird die WLAN-Simulation in einem Treiber implementiert. Im Unterschied zu dem Testkonzept der Masterarbeit, werden von Linux jedoch keine realen, aufgezeichneten Daten verwendet, stattdessen wird gleich ein komplettes WLAN-Netzwerk simuliert. Weitere Informationen dazu finden sich unter [Linux].

4.6.1. Überprüfungsaufwand

Für die Überprüfung des Entwurfs stützt sich der Überprüfungsplan maßgeblich auf die Verwendung praktischer Methoden. Dabei wird versucht, die Nachteile des hohen Hardwareaufwandes zu reduzieren. Weiterhin wird damit auch mehr Kontrolle über die Rahmenbedingungen der Überprüfung erreicht.

Der hohe Aufwand besteht in der Bereitstellung und der Konfiguration der für ein WLAN-Netzwerk notwendigen Hardware. Dazu gehören – bei einem Infrastrukturnetzwerk[41] – mindestens ein Client und ein Access-Point. Diese müssen vor jedem Test entsprechend der Testanforderungen konfiguriert werden, welches ein manuelles und somit langwieriges Unterfangen darstellt. Alternativ können mehrere identische Clients und Access-Points bereitgestellt werden, die sich nur in der Konfiguration unterscheiden, wobei jeder Testfall dann auf dem für ihn ausgelegten Hardwareaufbau ausgeführt wird.

4.6.2. Aufwandsreduktion

Die hier vorgestellte Reduktion des Testaufwandes beruht darauf, das WLAN-Netzwerk zu simulieren. Bei der Simulation besteht – ähnlich wie bei der theoretischen Überprüfung – die Gefahr die Randbedingungen unrealistisch zu setzen. Um diese Gefahr zu mindern werden bei der Simulation reale Daten verwendet, also solche Daten wie sie in einem realen Drahtlosnetzwerk auftreten.

WLAN-Hardware wird nur noch zum Sammeln dieser Daten benötigt. Anschließend kann sie für andere Zwecke verwendet werden. Die Vorteile liegen also darin, einen überschaubaren Hardwareaufwand zu haben, die langwierige und manuelle Konfiguration nur einmal pro Testszenario durchführen zu müssen und die späteren Simulationen automatisch, zeiteffizient und vor allem unter reproduzierbaren Bedingungen ausführen zu können.

[41] Abschnitt 2.2

Für die Datensammlung kommt dabei der bereits bewährte FreeBSD-WLAN-Stack zum Einsatz, welcher somit als Referenz dient. Für diesen Zweck wurde der FreeBSD-WLAN-Stack extra auf Haiku portiert (Abschnitt 3.5).

4.6.3. Randbedingungen des Überprüfungsplans

Der Überprüfungsplan beruht auf Prinzipien der testgetriebenen und iterativen Softwareentwicklung. Diese Prinzipien sind gekennzeichnet durch das Implementieren von Testfällen bevor der Produktivcode geschrieben wird (testgetrieben) und durch ein jederzeit ausführbares Codegerüst, welches nach und nach mit Produktivcode ausgefüllt wird (iterativ). Vergleichende Tests[42] mit anderen WLAN-Stacks werden in diesem Plan nicht berücksichtigt, obwohl – dank der FreeBSD-WLAN-Stack-Portierung – prinzipiell möglich.

Beim Testen des Produktivcodes konzentriert sich der Testplan dabei auf den Test der WLAN-Stack-Komponente IEEE80211 (Abschnitt 4.5.2). Insbesondere berücksichtigt der Testplan keine Systemintegrationstests, da dies hier zu weit führen würde. Es werden also Probleme, die im Zusammenspiel mit den restlichen Netzwerkkomponenten auftreten, nicht geprüft.

4.6.4. Überprüfungsplan

Der Weg zur Überprüfung der Korrektheit des objektorientierten WLAN-Stack-Entwurfs besteht aus den folgenden Hauptschritten:

1. Portierung des FreeBSD-WLAN-Stacks
2. Test der Portierung
3. Sammeln der Testdaten
4. Umsetzung der WLAN-Simulation
5. Implementierung des objektorientierten Entwurfs

Von diesen Schritten konnten im Rahmen der Masterarbeit Schritt eins und zwei umgesetzt werden. Eine nähere Beschreibung dieser beiden Schritte findet sich in Abschnitt 3.5.

Die ausstehenden Schritte werden in den nachfolgenden Abschnitten beschrieben. Dabei sollten diese nicht als streng-sequenziell[43] angesehen werden, sondern (ganz im Sinne der iterativen Softwareentwicklung) als iterativ-sequenziell. Mit anderen Worten: Diese Schritte werden für jedes weitere Testszenario erneut durchlaufen. Abbildung 4.24 veranschaulicht diese Unterscheidung von streng-sequenziell und iterativ-sequenziell nocheinmal grafisch.

[42] Durchsatztests, Reaktionsgeschwindigkeitstests, ...
[43] Der nächste Schritt wird erst nach vollständiger Abarbeitung des vorhergehenden begonnen.

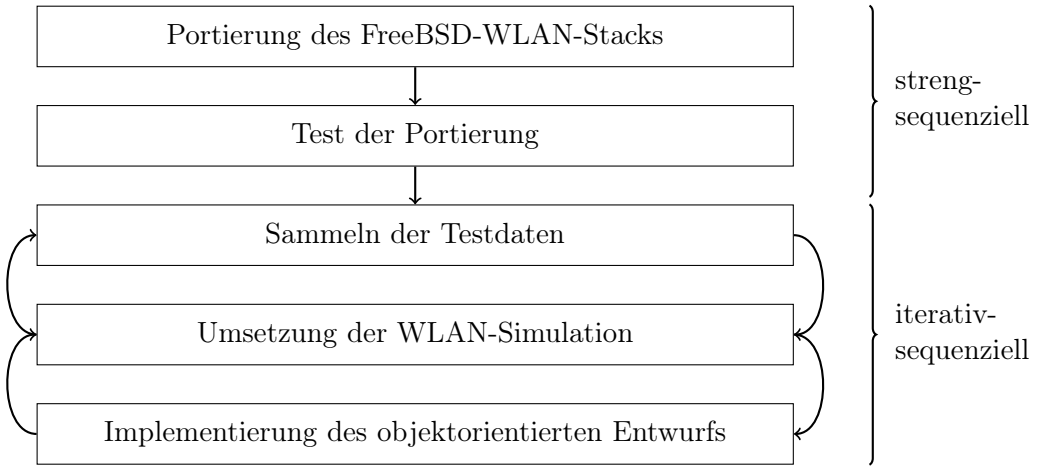


Abbildung 4.24.: Streng-sequenzielle und iterativ-sequenzielle Schritte des Überprüfungsplans

4.6.4.1. Sammeln der Testdaten

Zum Sammeln der Testdaten muss der portierte FreeBSD-WLAN-Stack zunächst so angepasst werden, dass ein Mitschneiden der gesendeten und empfangenen Daten-/Managementframes möglich wird. Der FreeBSD-WLAN-Stack selbst besitzt zwar in Form spezieller Schnittstellen[44] von Haus aus diese Möglichkeit, jedoch müssen diese Schnittstellen erst noch für Haiku bereitgestellt werden. Dafür ist eine Erweiterung der FreeBSD-Kompatibilitätsschicht notwendig.

Nachdem diese Vorbedingung erfüllt ist, können die Testszenarien entwickelt werden, welche zum späteren Testen der Implementierung des objektorientierten WLAN-Stack-Entwurfs verwendet werden sollen. Dabei bietet es sich an, die Testszenarien entsprechend der Einsatzfeldabdeckung zu sortieren. Anders gesagt, ist es sinnvoll, zunächst solche Testszenarien zu verwenden, welche einen breiten Nutzerkreis besitzen und seltene Szenarien erst später zu berücksichtigen.

Unter dieser Maßgabe werden anfangs nur Testszenarien für Infrastrukturnetzwerke berücksichtigt, da diese heutzutage am geläufigsten sind. Unter dem Gesichtspunkt der iterative Softwareentwicklung bietet es sich weiterhin an, diese Testszenarien entsprechend der Schrittfolge für den Beitritt zu einem solchen Netzwerk zu gliedern. Somit ergibt sich folgende Anordnung der Testszenarien:

1. Suchen
2. Synchronisieren
3. Authentifizieren

[44] bezeichnet als Berkeley Packet Filter

4. Assoziieren
5. Daten empfangen
6. Daten senden
7. Deassoziieren
8. Deauthentifizieren

Das Sammeln der Testdaten kann nun entsprechend des Fortschritts bei der Implementierung des objektorientieren WLAN-Stacks erfolgen. Die Testdaten können außer den bereits erwähnten Daten- und Managementframes auch andere Informationen enthalten. Zum Beispiel ist es sinnvoll die Zeit zwischen gesendeten und empfangenen Frames mitzuschneiden, damit auch Tests des Zeitverhaltens[45] möglich sind.

4.6.4.2. Umsetzung der WLAN-Simulation

Alle vorhergehenden Schritte waren auf das Bereitstellen der Testdaten für die WLAN-Simulation ausgerichtet. Dieser Schritt behandelt nun die Umsetzung dieser Simulation. Dafür ist es notwendig, die IEEE80211-Komponente von zwei Seiten aus mit Testkomponenten interagieren zu lassen. Abbildung 4.25 verdeutlicht diese Testinfrastruktur.

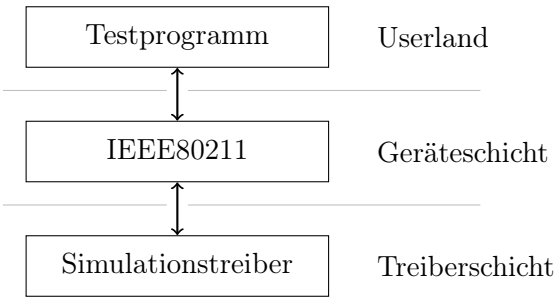


Abbildung 4.25.: Ansicht der Testinfrastruktur

Darin ist zu sehen, dass eine Seite von der Komponente „Testprogramm“ und die andere von der Komponente „Simulationstreiber“ besetzt wird. Der Entwurf dieser beiden Komponenten wird später behandelt. Weiterhin zeigt die Abbildung 4.25 auch, dass Haikus gesamter Netzwerkstack bei den Tests unbenutzt bleibt. Dadurch wird das Eingrenzen möglicher Fehlerquellen erleichtert, da der Netzwerkstack als potenzielle Fehlerquelle herausfällt.

Ein Nachteil dieser Testinfrastruktur liegt in der Untestbarkeit der korrekten Integration des objektorientierten Entwurfs in Haiku.

[45] Beispielsweise für das Testen der Reaktion auf Timeouts.

Entwurf des Testprogramms

Das Testprogramm ist für die Steuerung und Auswertung der einzelnen Tests zuständig. Es enthält dafür eine Liste der verfügbaren Testszenarien und der zugehörigen erwarteten Testergebnisse. Das Testprogramm verwendet für die Ergebnisprüfung eine integrierte Liste der Sollergebnisse. Dies ist praktikabel, da die Änderungen an einem Userlandprogramm vergleichweise[46] schnell angewendet werden können[47].

Entwurf der WLAN-Simulation

Die WLAN-Simulation erfolgt in einer eigenen Komponente. In Abbildung 4.26 wird dabei die Anbindung dieser Komponente an den objektorientierten WLAN-Stack veranschaulicht.

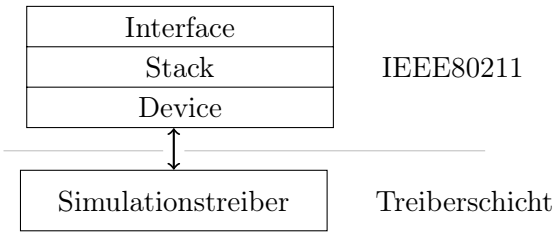


Abbildung 4.26.: Einbindung des WLAN-Simulationstreibers

Aus Sicht des WLAN-Stacks handelt es sich dabei also einfach um einen IEEE-802.11-Treiber. Dementsprechend wird die Simulationskomponente auch als Treiber implementiert. Dieser Treiber bindet anstelle von Hardware eine Datenbank ein (Abbildung 4.27), in welcher die Simulationsdaten enthalten sind. Diese Daten sind unveränderlich und wurden vorher durch den Prozess der Datensammlung/Datenaufbereitung (Abschnitt 4.6.4.1) in die Datenbank eingepflegt. Daten werden in der Datenbank dabei der jeweiligen, zu simulierenden Drahtlosnetzwerkkonfiguration zugeordnet. Die Datenbank besteht dabei aus einer Datei, welche ein – noch zu definierendes – menschenlesbares Format verwendet. Damit wird das Hinzufügen, Ändern und Löschen von Testkonfigurationen erleichtert. Natürlich ließen sich die Testdaten auch direkt in den Simulationstreiber integrieren, aber dann müsste der Treiber für jede zusätzliche Konfiguration neukompiliert und gestartet werden. Dies gestaltet sich aufgrund der Treiberausführung im Kernelland als zu aufwendig.

[46] Im Vergleich zu einem Kernellandprogramm, wie der Simulationstreiber eines darstellt.
[47] Neukompilierung und Neustart sind problemlos möglich.

Abbildung 4.27.: Anbindung der Datenbank an den Simulationstreiber

Gestartet wird die Simulation schließlich über ein Testprogramm, welches im Userland angesiedelt ist. Dieses Programm teilt dem Simulationstreiber zunächst mit, welche Drahtlosnetzwerkkonfiguration zum Testen verwendet werden soll und startet anschließend den Test der IEEE80211-Komponente. Der Simulationstreiber verhält sich aus Sicht der IEEE80211-Komponente nun annähernd wie ein reales Drahtlosnetzwerk. Abbildung 4.28 veranschaulicht den Ablauf eines solchen Tests am Beispiel der Scanfunktionalität.

Beispieltestlauf

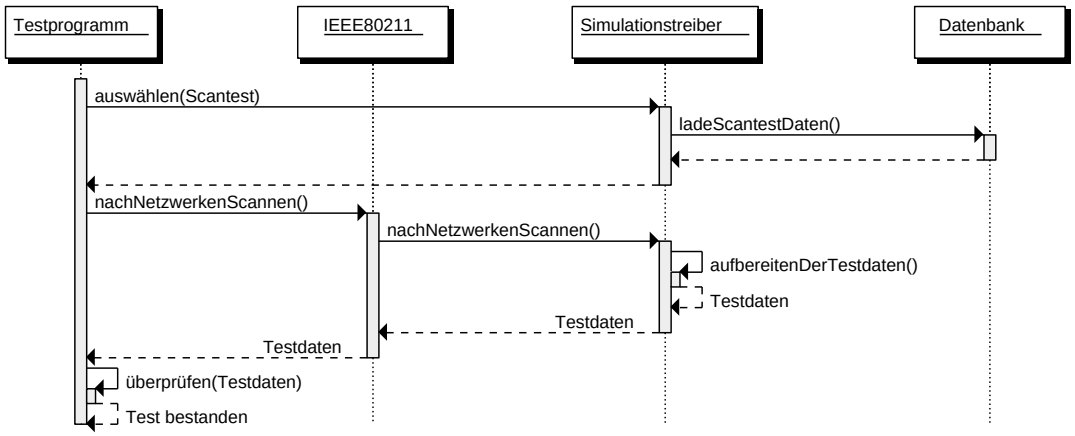


Abbildung 4.28.: Test der Scanfunktionalität

Zunächst wählt das Testprogramm den Scantest im Simulationstreiber aus. Der Treiber lädt daraufhin den entsprechenden Datensatz aus der Datenbank. Als nächstes beginnt das Testprogramm mit dem Testen der Testfunktionalität, indem es die IEEE80211-Komponente auffordert nach Drahtlosnetzwerken zu scannen. Diese Komponente fordert daraufhin den IEEE-802.11-Treiber – also in diesem Fall den Simulationstreiber – auf, nach Netzwerken zu suchen. Der Simulationstreiber liefert schließlich die in der Datenbank hinterlegten Drahtlosnetzwerke zurück. Das Testprogramm überprüft nun nur noch, ob die von der IEEE80211-Komponente zurückgelieferten Drahtlosnetzwerke den erwarteten entsprechen und teilt das Ergebnis dieses Tests mit.

4.6.4.3. Implementierung des objektorientierten Entwurfs

Der in den vorhergehenden Schritten betriebene Aufwand war notwendig, um die daran anschließende, iterative und testgetriebene Implementierung des objektorientierten Entwurfs zu ermöglichen. Dabei orientiert sich der Implementierungsprozess an der Reihenfolge der Bereitstellung der Testdaten. Schließlich werden in der testgetriebenen Softwareentwicklung zuerst die Tests implementiert und erst danach der Produktivcode. Und dann auch nur soviel Produktivcode, wie zum Bestehen der entsprechenden Tests notwendig ist.

4.7. Zusammenfassung

In diesem Kapitel wurde der Entwurfs eines objektorientierter WLAN-Stacks gezeigt, der sich am Referenzdesign einer WLAN-Station in IEEE Std 802.11 orientiert. Dabei wurde ein direkter Bezug zur Sprache von IEEE 802.11 hergestellt, wodurch sich der Einarbeitungsaufwand für neue Entwickler reduziert[48]. Desweiteren wurde die Integration des Entwurfs in Haiku demonstriert und ein Plan zur Überprüfung der Gültigkeit des Entwurfs wurde erstellt.

[48] Im Vergleich zu strukturorientierten WLAN-Stacks.

5. Auswertung und Ausblick

Die Verständlichkeit und Erweiterbarkeit eines WLAN-Stackentwurfs lässt sich durch Verwendung objektorientierter Konzepte steigern, wie es in dieser Masterarbeit demonstriert wurde. Das dabei zugrundeliegende Konzept, Komponenten entsprechend ihres Verhaltens zu modellieren und nicht entsprechend des Datenflusses, wie es im strukturorientierten Softwareentwurf der Fall ist, erlaubt eine leichte Zuordnung der Spezifikationen in IEEE Std 802.11 zur jeweiligen Komponente.

Dies liegt maßgeblich darin begründet, dass IEEE Std 802.11 Verhalten (zum Beispiel in Form von Protokollen) anstell von von Datenflüssen definiert. Zwar lassen sich strukturorientierte Entwürfe aus diesem Standard ableiten, wie die beiden Opensourcevertreter aus der BSD- und Linuxwelt deutlich demonstrieren, jedoch führt dies inhärent zu Schwierigkeiten, wenn es darum geht den Bezug zu IEEE Std 802.11 herzustellen. Dies ist keine Kritik an den Entwicklern dieser offenen WLAN-Stacks, welche im Rahmen ihrer strukturorientierten Möglichkeiten[1] eine hochwertige Umsetzung des WLAN-Standards geschaffen haben, sondern zeigt vielmehr die Vorteile auf, welche sich beim Einsatz objektorientierter Konzepte im Kernelland darbieten.

Der nächste logische Schritt ist die Umsetzung des zuvor besprochenen Überprüfungsplans (Abschnitt 4.6), was eine Implementierung und das Testen des objektorientierten WLAN-Stackentwurfs zur Folge hat. Daran anknüpfend sind Systemintegrationstests notwendig, welche im Überprüfungsplan nicht berücksichtigt werden und somit zunächst noch zu entwerfen sind. Für die Umsetzung solcher Systemintegrationstests ist es unter anderem notwendig die – in Haiku fehlende – LLC-Komponente (Abschnitt 4.5.2) zu implementieren. Als WLAN-Treiber kann dabei der vorgestellte WLAN-Simulationstreiber verwendet werden.

Interessant ist auch die Frage, wie ein objektorientierter WLAN-Stack im Vergleich mit den strukturorientierten Implementierungen abschneidet. Die Beantwortung dieser Frage erfordert dabei jedoch einen hohen Hardwareaufwand. Allerdings stellt dies die einzige Methode dar, realistische Vergleichsmessungen durchzuführen. Simulationen sind hier fehl am Platz.

[1] Bei den BSDs und bei Linux kann im Kernelland nur die strukturorientierte Programmiersprache C verwendet werden.

Glossar

Ack	Acknowledge. Bestätigung.
ACM	Spezieller Betriebsmodus einer Station. Dieser Modus erlaubt das Erzeugen eines Infrastrukturnetzwerkes.
Ad-hoc-Modus	Spezieller Betriebsmodus einer Station. Dieser Modus erlaubt die direkte Kommunikation einer Station mit allen Teilnehmern eines Drahtlosnetzwerkes.
AES	Advanced-Encryption-Standard. Als sicher geltender Verschlüsselungsstandard.
AP	Access-Point. Eine Station im Access-Control-Mode mit Zugriff auf das Distribution-System.
API	Application-Programming-Interface. Öffentliche Schnittstelle eines Betriebssystems oder einer Bibliothek, welche von Anwendungssoftware genutzt werden kann.
BeOS	Be Operating System. Von grundauf neuentwickeltes Betriebssystem der Firma Be; Entwicklung eingestellt.
BSD	Berkeley-Software-Distribution. Sammelbegriff für alle Betriebssysteme, die auf dem UNIX-Betriebssystem der Berkeley Universität basieren. Namhafte Vertreter sind FreeBSD, NetBSD und OpenBSD.
BSS	Basic-Service-Set. Allgemeine Bezeichnung für ein Drahtlosnetzwerk.
CBC-MAC	Cipher-Block-Chaining-Message-Authentication-Code. Erlaubt das Sicherstellen der Integrität und der Authentizität von Datenframes.
CCMP	CTR-with-CBC-MAC-Protocol. Aktuell sicherstes Verschlüsselungsverfahren von IEEE 802.11. Löst die WEP- und TKIP- Verschlüsselungsverfahren ab.
CDT	C/C++ Development Tooling. Erweiterung der Eclipse Entwicklungsumgebung um die Unterstützung der Programmiersprachen C und C++.

CF	Contention-Free. Zeitphase, während derer kein Wettbewerb um den Medienzugriff erfolgt.
Clientmodus	Spezieller Betriebsmodus einer Station. Dieser Modus erlaubt den Beitritt zu einem Infrastrukturnetzwerk.
CP	Contention-Phase. Zeitphase, während derer Wettbewerb um den Medienzugriff erfolgt.
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance. Protokoll, dass den Zugriff mehrerer Netzwerkteilnehmer auf das Medium regelt.
CSMA/CD	Carrier Sense Multiple Access with Collision Detection. Protokoll, dass den Zugriff mehrerer Netzwerkteilnehmer auf das Medium regelt.
CTR	Counter-Mode. Spezielles Verfahren bei Blockverschlüsselung. Wird bei CCMP eingesetzt.
DHCP	Dynamic-Host-Configuration-Protocol. Ein Netzwerkkonfigurationsprotokoll.
DS	Distribution-System. Das Verteilsystem, welches für den Datenaustausch innerhalb eines erweiterten Infrastrukturnetzwerkes zuständig ist. Besteht aus den Access-Points und dem Verteilermedium.
DSM	Distribution-System-Medium. Das Verteilermedium, welches mehrere Infrastrukturnetzwerke miteinander verbindet. Kann drahtlos aber auch drahtgebunden sein.
DSSS	Direct-Sequence-Spread-Spectrum. Frequenzspreizverfahren.
ERP	Extended-Rate-Phy. Kombination aus Frequenzspeiz- und Mehrträgerverfahren. Abwärtskompatibel zu bestehenden DSSS- und HR/-DSSS-Drahtlosnetzwerken.
ESS	Extended-Service-Set. Erweiterte Form eines Infrastrukturnetzwerkes. Besteht aus mehreren Infrastrukturnetzwerken, die allesamt die gleiche SSID verwenden und überlappende Ausleuchtungszonen besitzen.
FHSS	Frequency-Hopping-Spread-Spectrum. Frequenzsprungverfahren.
FreeBSD	Free-Berkeley-Software-Distribution. Opensource UNIX-Betriebssystem.
FTP	File Transfer Protokoll. Ein Netzwerkprotokoll zur Dateiübertragung.

GCC	GNU-Compiler-Collection. Quelloffene Softwaresammlung zum Kompilieren von Software.
gdb	GNU Debugger. Quelloffener Softwaredebugger des GNU-Projektes.
GNU	GNU is Not UNIX. Ein Projekt, welches sich für die Quelloffenheit von Software einsetzt.
Haiku	Opensource Betriebssystem. Basiert auf dem Design von BeOS.
HR/DSSS	High-Rate-Direct-Spread-Sprectrum. Frequenzspreizverfahren. Das selbe Verfahren wie bei DSSS nur mit höheren Datenraten.
IBSS	Independent-Basic-Service-Set. Spezielles Drahtlosnetzwerk, auch als Ad-Hoc-Netzwerk bezeichnet. Alle Teilnehmer werden im Ad-Hoc-Modus betrieben.
ID	Identität. Eine Zahl mit Eindeutigkeitscharakter.
IEEE	Institute of Electrical and Electronics Engineers. Berufsverband.
IP	Internet-Protocol. Ein Netzwerkprotokoll im Internet.
IPv4	Internet-Protocol version 4. Version 4 des Internet-Protokolls.
LAN	Local-Area-Network. Ein drahtgebundenes Netzwerk, welches Bestandteil der IEEE-802-Familie ist.
LLC	Logical-Link-Control. Eine Schicht und ein Protokoll der IEEE-802-Familie.
MAC	Media-Access-Control. Medienzugriffskontrolle.
MIB	Management-Information-Base. Standardisiertes Verfahren zur Beschreibung von Managementinformationen.
MIC	Message-Integrity-Code. Spezielle Prüfsumme, welche die Integritätsüberprüfung von Datenframes ermöglicht.
MLME	MAC-Layer-Management-Entity. Teilkomponente der MAC-Schicht, welche WLAN-Verwaltungsfunktionalität beinhaltet.
MPDU	MAC-Protocol-Data-Unit. Ein MAC-Frame, der zur Übergabe an die PHY vorgesehen ist. Ein MAC-Frame, wie ihn die PHY an die MAC übergibt.
OFDM	Orthogonal-Frequency-Division-Multiplexing. Mehrträgerverfahren.

OpenBSD	Open Berkeley Software Distribution. Opensource UNIX Betriebssystem mit Fokus auf Sicherheit.
OSI	Open-Systems.Interconnection. Schichtenmodell für Kommunikationsprotokolle.
PHY	Physical Layer. Bitübertragungsschicht.
PLCP	Physical-Layer-Convergence-Procedure. Teilschicht der Bitübertragungsschicht.
PLME	PHY-Layer-Management-Entity. Teilkomponente der PHY-Schicht, welche WLAN-Verwaltungsfunktionalität beinhaltet.
PMD	Physical-Medium-Dependent. Teilschicht der Bitübertragungsschicht.
Poll	Polling. Aktive Zustandsabfrage.
POSIX	Portable-Operating-System-Interface. Schnittstellenspezifikation mit dem Ziel Softwareportierungen zu erleichtern.
QoS	Quality-of-Service. Eine spezielle Funktion eines WLAN-Stacks. Datenpakete werden in verschiedene Prioriätenklassen eingeordnet.
R5	Release 5. Version 5 von BeOS.
SME	Station-Management-Entity. Managementkomponente des WLAN-Standards, welche auf die Funktionalität der MLME und PLME aufbaut. Stellt die Schnittstelle zu den Managementfunktionalitäten für das Betriebssystem bereit.
STA	Station. Allgemeine Bezeichnung für einen Teilnehmer in einem WLAN.
TCP	Transmission-Control-Protocol. Ein Internetnetzwerkprotokoll.
TKIP	Temporal-Key-Integrity-Protocol. Erweitert die WEP-Verschlüsselung um Integritätsschutz und tauscht den Schlüssel periodisch aus.
UML	Unified-Modeling-Language. Eine standardisierte Modellbeschreibungssprache.
WEP	Wired-Equivalent-Privacy. Verschlüsselungsverfahren der ersten Stunde. Mittlerweile veraltet, weil unsicher.
WLAN	Wireless-Local-Area-Network. Die Drahtlosvariante eines LAN.
WPA	Wi-Fi-Protected-Access. Kommerzielle Bezeichnung für Stationen mit TKIP-Verschlüsselung.
WPA2	Wi-Fi-Protected-Access 2. Kommerzielle Bezeichnung für Stationen mit CCMP-Verschlüsselung.

A. Testhardware

- WLAN-Access-Point
  - WLAN-Router
  - Hersteller: AVM Computersysteme Vertriebs GmbH
  - Produkt: FRITZ!Box WLAN 3030
  - Firmwareversion: 21.04.34
  - WLAN-Einstellungen:
    - WLAN-Kanal: 8
    - SSID: FRITZ!Box WLAN 3030
    - SSID wird periodisch gesendet
    - Technologie der physikalischen Schicht: IEEE 802.11g
    - Sicherheit: unverschlüsselt
- WLAN-Client
  - Laptop
  - Hersteller: IBM Corporations
  - Produkt: Thinkpad T41p, Type 2373, Model GEG
  - WLAN-Chipsatz:
    - Hersteller: Atheros Communications Inc.
    - Produkt: AR5212 802.11abg NIC, revision 01
- LAN-Client
  - Desktop-PC
  - Mainboardhersteller: ASRock Inc.
  - Mainboardprodukt: Penryn1600SLI-110dB
  - LAN-Chipsatz:
    - Hersteller: nVidia Corporation
    - Produkt: MCP51 Ethernet Controller (rev a3)

B. Werkzeuge

B.1. Entwicklungsumgebung

- Betriebssysteme:
  - Ubuntu 9.10 64-Bit
  - Haiku bis einschließlich Revision 35756 32-Bit
- Integrierte Entwicklungsumgebungen:
  - Eclipse 3.5
    - mit CDT Plugin Version 6
    - auf Java 6
  - QT Creator 1.2 (Haiku) und 1.3 (Ubuntu)
- Kompilierwerkzeuge:
  - Jam 2.5-haiku-20090626
  - GNU Compiler Collection 2.95.3-haiku-090629

B.2. Testumgebung

- Kapitel Testhardware auf der vorherigen Seite
- VirtualBox 3.1
  - Ubuntu 9.10 64-Bit als Host
  - OpenBSD 4.5 32-Bit als Guest
    - Stellt FTP-Server bereit
    - Zum Testen der Portierung des FreeBSD-WLAN-Stacks
- Haiku bis Revision 35756

B.3. Ausarbeitung

- Textwerkzeug:
  - LyX 1.6.5
  - TeX Live 2007-11
- Bildwerkzeuge:
  - Sequenzdiagramme: Quick Sequence Diagram Editor 3.1
  - Klassendiagramme: Dia 0.97
  - Übrige Grafiken: KTikZ 0.9

C. Prototyp

Im Folgenden wird der Quellcode zu einer prototypischen Implementierung des objektorientierten WLAN-Stack-Entwurfs aufgeführt. Der Prototyp implementiert dabei das Gerüst für den Anwendungsfall „Authentifizierung an einem Drahtlosnetzwerk". Dieser Anwendungsfall erfordert die Interaktion nahezu aller Komponenten[1] (Abschnitt 4.3) des Entwurfs. Somit bildet dieser ein gutes Studienobjekt für die Implikationen des Entwurfs auf seine Implementierung.

Die Quellcodedateien des Prototypen sind dabei auf verschiedene Verzeichnisse verteilt. Diese Verteilung folgt der Architektur der `IEEE80211`-Komponente und ihrer Schichten (Abbildung 4.21). So findet sich der Quellcode der `IEEE80211`-Komponente im Verzeichnis `ieee80211`. Der Quellcode der `Interface`-Schicht befindet sich direkt in diesem Verzeichnis, wobei die zugehörigen Dateien mit „ieee80211_" beginnen. Der Quellcode für die `Stack`-Schicht befindet sich im Verzeichnis `ieee80211/stack`/.

Wie in Abschnitt 4.5.3 bereits erklärt wurde, ist die `Stack`-Schicht die Stelle, in welcher der objektorientierte WLAN-Stack-Entwurf implementiert wird. Somit befinden sich hier alle Quellcodedateien, die den Entwurf implementieren. Dabei folgt die Verzeichniseinteilung der Komponenteneinteilung (Abbildung 4.9) des Entwurfs. Beispielsweise findet sich die `MAC-Management`-Komponente im Verzeichnis `ieee80211/stack/mac_management`/.

Die `Device`-Schicht ist der `Stack`-Schicht zugeordnet und befindet sich in `ieee80211/stack/device`/. Diese Zuordnung wurde primär getroffen, um im Verzeichniss `ieee80211/stack/` alle objektorientierten Implementierungen zu konzentrieren. Sollte sich diese Zuordnung als zu verwirrend erweisen (Abschnitt 4.1.2), könnte die `Device`-Schicht auch eine Verzeichnisebene nach oben verschoben werden.

In der Abbildung C.1 findet sich die Komponentenzuordnung innerhalb der Verzeichnisstruktur nochmals in visualisierter Form.

In den nachfolgenden Unterabschnitten werden ausschließlich die Quellcodedateien der drei Komponenten `Station-Management`, `MAC-Management` und `MPDU-Coordination` gelistet.

[1] Nur die `MAC-Data`-Komponente wird nicht benötigt.

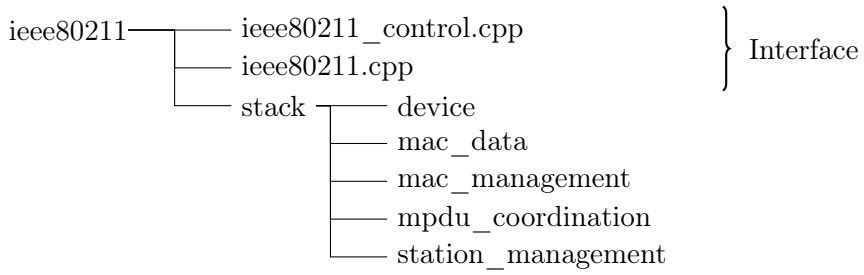


Abbildung C.1.: Komponentenzuordnung innerhalb der Verzeichnisstruktur des Prototypen

C.1. station_management/Roster.cpp

```
1  /*
2   * Copyright 2010 Haiku Inc. All Rights Reserved.
3   * Distributed under the terms of the MIT License.
4   *
5   * Authors:
6   *      Colin Günther, coling@gmx.de
7   */
8
9
10 /*! Implementation of the Station Management Entity (SME).
11     The term SME refers to the concept as described in the IEEE 802.11
12     documentation.
13     It is the central access point (roster) for all management related tasks,
14     needed for operating the wireless connections properly.
15
16     \sa IEEE Std 802.11-2007, Section 10 "Layer Management".
17  */
18
19
20 #include <station_management/Roster.h>
21
22
23 using namespace Ieee80211::StationManagement;
24
25
26 /*! Connects to a wireless network in a synchronous way.
27     After this method returns the station is either connected to the desired
28     network or it is not.
29     There may be some side effects depending on the completeness of information
30     you are passing in with the request. Rule of thumb is: the more information
31     contained within the connection request, the less side effects are needed.
32
33     For example a scan might be necessary when you are passing a SSID only, so
34     that the Station Management Entity can retrieve the respective BSSID of the
35     desired network.
36
37     \param request contains information needed to connect with the desired
38             wireless network.
39     \return B_OK Connection established.
40     \return B_ERROR Connection not established.
41  */
42 status_t
43 Roster::ConnectToNetwork(ConnectRequest* request)
44 {
45     if (request == NULL)
46         return B_BAD_VALUE;
47
48     status_t status = _RetrieveMissingConnectRequestData(request);
49     if (status != B_OK)
50         return status;
51
```

```
52      MacManagement::JoinRequest joinRequest(request);
53      status = fMacRoster.AddJoinRequest(&joinRequest);
54      if (status != B_OK)
55          return status;
56
57      MacManagement::JoinRequestResult* joinResult = NULL;
58      status = joinRequest.WaitForConfirm(&joinResult);
59      if (status != B_OK)
60          return status;
61      // TODO: Actually test the joinResult before proceed
62
63      MacManagement::AuthenticateRequest authenticationRequest(request);
64      status = fMacRoster.AddAuthenticateRequest(&authenticationRequest);
65      if (status != B_OK)
66          return status;
67
68      MacManagement::AuthenticateRequestResult* authenticateResult = NULL;
69      status = authenticationRequest.WaitForConfirm(&authenticateResult);
70      if (status != B_OK)
71          return status;
72      // TODO: Actually test the authenticateResult before proceed
73
74      return B_OK;
75  }
76
77
78  // #pragma mark - private functions
79
80
81  /*! Ensuring that all data necessary for connecting to a wireless network is
82      contained in the request.
83      Tries to retrieve missing data by issuing a scan request. If missing data
84      cannot be obtained this way the method will fail.
85
86      \param request to be checked and to be made up, if necessary.
87      \return B_OK Request contains all data (now).
88      \return B_ERROR Request misses data and the method failed in obtaining it.
89   */
90  status_t
91  Roster::_RetrieveMissingConnectRequestData(ConnectRequest* request)
92  {
93      return B_OK;
94  }
```

C.2. mac_management/Roster.cpp

```
/*
 * Copyright 2010 Haiku Inc. All rights reserved.
 * Distributed under the terms of the MIT License.
 *
 * Authors:
 *      Colin Günther, coling@gmx.de
 */


/*! Implementation of the MAC Layer Management Entity (MLME).
    The term MLME refers to the concept as described in the IEEE 802.11
    documentation.
    It is the central access point (roster) for managing all MAC related tasks,
    needed for operating the wireless connections properly.

    \sa IEEE Std 802.11-2007, Section 10.3 "MLME SAP interface".
    \sa IEEE Std 802.11-2007, Section 11 "MLME".
 */


#include <mac_management/Roster.h>


using namespace Ieee80211::MacManagement;


/*! Handling over a new authenticate request to the MAC Layer Management Entity
    (MLME).
    Successful completion of this method does only mean that the MLME will try
    to fulfill the request, not that it was performed already.
    Note: It is expected that the authenticate request contains all necessary
    data, otherwise it will fail during processing later on .

 */
status_t
Roster::AddAuthenticateRequest(AuthenticateRequest *request)
{
    return fAuthenticateService.Process(request);
}


/*! Handling over a new join request to the MAC Layer Management Entity (MLME).
    Successful completion of this method does only mean that the MLME will try
    to fulfill the request, not that it was performed already.
    Note: It is expected that the join request contains all necessary data,
    otherwise it will fail during processing later on .

    \param request to be performed by the MLME.
    \return B_OK MLME will try to perform the request.
    \return B_ERROR MLME will not try to perform the request.
 */
```

```
52  status_t
53  Roster::AddJoinRequest(JoinRequest* request)
54  {
55      return B_OK;
56  }
```

C.3. mac_management/frames/FrameFactory.cpp

```
1  /*
2   * Copyright 2010 Haiku Inc. All rights reserved.
3   * Distributed under the terms of the MIT License.
4   *
5   * Authors:
6   *      Colin Günther, coling@gmx.de
7   */
8
9
10 /*! Central place for constructing IEEE 802.11 management frames.
11     \sa IEEE Std 802.11-2007, Section 7 "Frame formats".
12     \sa IEEE Std 802.11-2007, Section 7.2.3 "Management frames".
13     \sa IEEE Std 802.11-2007, Section 7.3 "Management frame body components".
14     \sa IEEE Std 802.11-2007, Section 7.4 "Action frame format details".
15  */
16
17
18 #include <mac_management/frames/FrameFactory.h>
19
20
21 using namespace Ieee80211::MacManagement;
22
23
24 /*! Constructs an authentication frame.
25     \sa IEEE Std 802.11-2007, Section 7.2.3.10 "Authentication frame
26         format"
27
28     \param request containing data needed to construct the frame.
29     \param frame will contain the constructed frame upon success.
30     \return B_OK The authentication frame was constructed successful.
31     \return Else Constructing the authentication frame failed.
32  */
33 status_t
34 FrameFactory::CreateAuthentication(AuthenticateRequest* request,
35     AuthenticationFrame** frame)
36 {
37     return B_OK;
38 }
```

C.4. mac_management/requests/AuthenticateRequest.cpp

```
1  /*
2   * Copyright 2010 Haiku Inc. All rights reserved.
3   * Distributed under the terms of the MIT License.
4   *
5   * Authors:
6   *      Colin Günther, coling@gmx.de
7   */
8
9
10 /*! Contains the data for the authentication request.
11
12     \sa IEEE Std 802.11-2007, Section 10.3.4.1 "MLME-AUTHENTICATE.request".
13  */
14
15
16 #include <mac_management/requests/AuthenticateRequest.h>
17
18
19 using namespace Ieee80211::MacManagement;
20
21
22 /*! Constructs a new object by using relevant data from the ConnectRequest.
23
24     \param request ConnectRequest object, containing data needed for
25         proper authentication.
26     \return NULL not enough memory.
27     \return AuthenticateRequest otherwise.
28  */
29 AuthenticateRequest::AuthenticateRequest(
30     StationManagement::ConnectRequest* request)
31 {
32     return;
33 }
```

C.5. mac_management/services/AuthenticateService.cpp

```
1  /*
2   * Copyright 2010 Haiku Inc. All rights reserved.
3   * Distributed under the terms of the MIT License.
4   *
5   * Authors:
6   *      Colin Günther, coling@gmx.de
7   */
8
9
10 /*! Implementation of the authenticate service.
11     Note: The 802.11 example implementation of this service processes
12     deauthentication requests, too. Our services handle one request type only,
13     which means that we use a dedicated deauthenticate service, instead.
14
15     \sa IEEE Std 802.11-2007, Annex C.3 "State machines for MAC STAs",
16         page 885 ff.
17  */
18
19
20 #include <mac_management/frames/AuthenticationFrame.h>
21 #include <mac_management/services/AuthenticateService.h>
22
23
24 using namespace Ieee80211::MacManagement;
25
26
27 // #pragma mark - service interface implementation
28
29
30 /*! Processes an authenticate request.
31     Successful return of this method does only mean, that the service is going
32     to process the request, not that it was processed already.
33
34     \param request to be processed.
35     \return B_OK Service is going to process the request.
36     \return Else Service can't process the request, due to an unrecoverable
37         error.
38  */
39 status_t
40 AuthenticateService::Process(AuthenticateRequest* request)
41 {
42     if (request == NULL)
43         return B_BAD_VALUE;
44
45     return fProcessQueue.Enqueue(request);
46 }
47
48
49 /*! Puts the service in its run state.
50
51     \param dummy never used. This is only required to allow different _Run()
```

```
52          methods per service in a common way.
53      \return B_OK The service terminated gracefully.
54      \return Else Unrecoverable error during service execution.
55   */
56  status_t
57  AuthenticateService::_Run(AuthenticateRequest* dummy)
58  {
59      AuthenticationFrame* frame = NULL;
60      AuthenticateRequest* request = NULL;
61      status_t status = B_OK;
62
63      do {
64          status = fProcessQueue.Dequeue(&request);
65          if (status != B_OK)
66              break;
67
68          status_t status = fFrameFactory.CreateAuthentication(request, &frame);
69          if (status != B_OK)
70              break;
71
72          status = fDistributeService.Process(frame);
73          if (status != B_OK)
74              break;
75
76          status = _WaitForEventAndConfirmRequest(request);
77      } while (status == B_OK);
78
79      return status;
80  }
81
82
83  // #pragma mark - observer interface implementation
84
85
86  /*! Called by the observed object (the observable) to notify the
87      AuthenticateService about an event.
88      Note: Normally only events the AuthenticateService registered for should be
89      notified here, but this should and will be checked nonetheless.
90
91      \param event which raises execution of this method.
92      \param observable source of the event.
93      \result B_OK The event will be processed by the AuthenticateService.
94      \result Else The event will not be processed by the AuthenticateService.
95          It shall be safe for the observable to ignore such errors.
96   */
97  status_t
98  AuthenticateService::NotifyEventFromObservable(Event* event,
99      Observable* observable)
100 {
101     if (event == NULL || observable == NULL)
102         return B_BAD_VALUE;
103
104     return _IfSupportedEventAndObservableStartProcessing(event, observable);
105 }
```

```
106
107
108 /*! Checks, whether the event is one of Class2Error or AuthenticateEven.
109  */
110 bool
111 AuthenticateService::_IsEventSupported(Event* event)
112 {
113     return true;
114 }
115
116
117 /*! Checks, whether the observable is the DistributeService.
118  */
119 bool
120 AuthenticateService::_IsObservableSupported(Observable* observable)
121 {
122     return observable == &fDistributeService;
123 }
124
125
126 // #pragma mark - private helper functions
127
128
129 /*! Creates an authenticate request result, by using data from the event.
130
131     \param event used to create the authenticate request result.
132     \param result used to return the constructed AuthenticateRequestResult.
133     \return B_OK AuthenticateRequestResult constructed successfully.
134     \return Else No result constructed.
135  */
136 status_t
137 AuthenticateService::_CreateRequestResult(Event* event,
138     AuthenticateRequestResult** result)
139 {
140     return B_OK;
141 }
142
143
144 /*! Waits for any event the AuthenticateService has registered for and confirms
145     the pending request.
146
147     \param request to be confirmed.
148     \return B_OK Event received and pending request confirmed.
149     \return Else request wasn't confirmed, due to an unrecoverable error.
150  */
151 status_t
152 AuthenticateService::_WaitForEventAndConfirmRequest(
153     AuthenticateRequest* request)
154 {
155     Event* event = NULL;
156     AuthenticateRequestResult* result = NULL;
157
158     status_t status = _WaitForEvent(&event);
159     if (status != B_OK)
```

```
160         return status;
161
162     status = _CreateRequestResult(event, &result);
163     if (status != B_OK)
164         return status;
165
166     return request->Confirm(result);
167 }
```

C.6. mac_management/services/DistributeService.cpp

```
1  /*
2   * Copyright 2010 Haiku Inc. All rights reserved.
3   * Distributed under the terms of the MIT License.
4   *
5   * Authors:
6   *      Colin Günther, coling@gmx.de
7   */
8
9
10 /*! Implementation of the distribute MAC Management Protocol Data Unit (MMPDU)
11     service.
12     \sa IEEE Std 802.11-2007, Annex C.3 "State machines for MAC STAs",
13         page 883 ff.
14  */
15
16
17 #include <mac_management/services/DistributeService.h>
18
19
20 using namespace Ieee80211::MacManagement;
21
22
23 // #pragma mark - service interface implementation
24
25
26 /*! Process a frame.
27     Successful return of this method does only mean, that the service is going
28     to process the request, not that it was processed already.
29
30     \param frame to be processed.
31     \return B_OK Service is going to process the frame.
32     \return Else Service can't process the frame, due to an unrecoverable error.
33  */
34 status_t
35 DistributeService::Process(Frame* frame)
36 {
37     return fProcessQueue.Enqueue(frame);
38 }
39
40
41 /*! Puts the service in its run state.
42
43     \param dummy never used. This is only required to allow different _Run()
44         methods per service in a common way.
45     \return B_OK The service terminated gracefully.
46     \return Else Error during service execution.
47  */
48 status_t
49 DistributeService::_Run(Frame* dummy)
50 {
51     Frame* frame = NULL;
```

```
52      status_t status = B_OK;
53
54      do {
55          status = fProcessQueue.Dequeue(&frame);
56          if (status != B_OK)
57              break;
58
59          status = fMpduRoster.AddManagementFrame(frame);
60      } while (status == B_OK);
61
62      return status;
63  }
64
65
66  // #pragma mark - observable interface implementation
67
68
69  /*! Register the observer with the event the observer is interested in.
70
71      \param observer to be registered.
72      \param event the observer is interested in.
73      \return B_OK The observer was successfully registered with the event.
74      \return Else The observer wasn't registered, due to an unrecoverable error.
75   */
76  status_t
77  DistributeService::AddObserverForEvent(Observer* observer, Event* event)
78  {
79      if (observer == NULL || event == NULL)
80          return B_BAD_VALUE;
81
82      return _IfSupportedObserverAndEventAddThem(observer, event);
83  }
84
85
86  /*! Checks, whether the observer is the AuthenticateServer.
87   */
88  bool
89  DistributeService::_IsObserverSupported(Observer* observer)
90  {
91      return true;
92  }
93
94
95  /*! Checks, whether the event is one of Class2Error or AuthenticateEven.
96   */
97  bool
98  DistributeService::_IsEventSupported(Event* event)
99  {
100     return true;
101 }
```

C.7. mpdu_coordination/Roster.cpp

```
1  /*
2   * Copyright 2010 Haiku Inc. All rights reserved.
3   * Distributed under the terms of the MIT License.
4   *
5   * Authors:
6   *      Colin Günther, coling@gmx.de
7   */
8
9
10 /*! Implementation of the MAC Protocol Data Unit (MPDU) Generation
11     state machine.
12     The IEEE 802.11 formal description of MPDU Generation puts fragmentation and
13     encryption services here, only. But the below implementation provides their
14     counterparts defragmentation and decryption, too. Thus it is more
15     appropriate of calling this entity MPDU Coordination.
16
17     \sa IEEE Std 802.11-2007, Annex C.3 "State machines for MAC STAs", page 837.
18  */
19
20
21 #include <mpdu_coordination/Roster.h>
22
23
24 using namespace Ieee80211::MpduCoordination;
25
26
27 /*! Handling over a new management frame to the MAC Protocol Data Unit (MPDU)
28     Coordination.
29     Successful completion of this method does only mean that the MPDU
30     coordinator will try to send the frame, not that it was sent already.
31
32     \param frame to be sent by the MPDU Coordinator.
33     \return B_OK MPDU coordinator will try to send the frame.
34     \return Else MPDU coordinator will not try to send the frame.
35  */
36 status_t
37 Roster::AddManagementFrame(MacManagement::Frame* frame)
38 {
39     return fFragmentService.Process(frame);
40 }
```

C.8. mpdu_coordination/algorithms/NullCipherAlgorithm.cpp

```
1  /*
2   * Copyright 2010 Haiku Inc. All rights reserved.
3   * Distributed under the terms of the MIT License.
4   *
5   * Authors:
6   *      Colin Günther, coling@gmx.de
7   */
8
9
10 /*! Implementation of a dummy cipher algorithm for circumstances, where neither
11     encryption nor decryption is needed.
12  */
13
14
15 #include <mpdu_coordination/algorithms/NullCipherAlgorithm.h>
16
17
18 using namespace Ieee80211::MpduCoordination;
19
20
21 // #pragma mark - CipherAlgorithm interface
22
23
24 /*! Doesn't decrypt anything.
25     But it checks whether the passed frame pointer is NULL.
26
27     \param frame isn't changed in anyway by this method.
28     \return B_OK It is safe to use the frame.
29     \return B_BAD_VALUE It isn't safe to use the frame, due to being a NULL
30         pointer.
31  */
32 status_t
33 NullCipherAlgorithm::Decrypt(Frame* frame)
34 {
35     if (frame == NULL)
36         return B_BAD_VALUE;
37     return B_OK;
38 }
39
40
41 /*! Doesn't encrypt anything.
42     But it checks whether the passed frame pointer is NULL.
43
44     \param frame isn't changed in anyway by this method.
45     \return B_OK It is safe to use the frame.
46     \return B_BAD_VALUE It isn't safe to use the frame, due to being a NULL
47         pointer.
48  */
49 status_t
50 NullCipherAlgorithm::Encrypt(Frame* frame)
51 {
```

```
52      if (frame == NULL)
53          return B_BAD_VALUE;
54      return B_OK;
55  }
```

C.9. mpdu_coordination/services/EncryptService.cpp

```
1  /*
2   * Copyright 2010 Haiku Inc. All rights reserved.
3   * Distributed under the terms of the MIT License.
4   *
5   * Authors:
6   *      Colin Günther, coling@gmx.de
7   */
8
9
10 /*! Implementation of the frame encryption service.
11     This service encrypts MAC Protocol Data Units, by using the cipher algorithm
12     currently in use by this service.
13  */
14
15
16 #include <mpdu_coordination/services/EncryptService.h>
17
18
19 using namespace Ieee80211::MpduCoordination;
20
21
22 // #pragma mark - service interface implementation
23
24
25 /*! Process a frame.
26     Successful return of this method does only mean, that the service is going
27     to process the frame, not that it was processed already.
28
29     \param frame to be processed.
30     \return B_OK Service is going to process the frame.
31     \return Else Service can't process the frame, due to an unrecoverable error.
32  */
33 status_t
34 EncryptService::Process(Frame* frame)
35 {
36     return fProcessQueue.Enqueue(frame);
37 }
38
39
40 /*! Puts the service in its run state.
41
42     \param dummy never used. This is only required to allow different _Run()
43         methods per service in a common way.
44     \return B_OK The service terminated gracefully.
45     \return Else Unrecoverable error during service execution.
46  */
47 status_t
48 EncryptService::_Run(Frame* dummy)
49 {
50     Frame* frame = NULL;
51     status_t status = B_OK;
```

```
52
53      do {
54          status = fProcessQueue.Dequeue(&frame);
55          if (status != B_OK)
56              break;
57
58          status = fCipherAlgorithm->Encrypt(frame);
59      } while (status == B_OK);
60
61      return status;
62  }
```

C.10. mpdu_coordination/services/FragmentService.cpp

```
/*
 * Copyright 2010 Haiku Inc. All rights reserved.
 * Distributed under the terms of the MIT License.
 *
 * Authors:
 *      Colin Günther, coling@gmx.de
 */


/*! Implementation of the frame fragmentation service.
    This service fragments MAC Service Data Units (MSDUs) and MAC Management
    Protocol Data Units (MMPDUs) into smaller units called MAC Protocol Data
    Units (MPDU).

    \sa IEEE Std 802.11-2007, Section 9.4 "Fragmentation".
    \sa IEEE Std 802.11-2007, Annex C.3 "State machines for MAC STAs", page 845.
 */


#include <mpdu_coordination/services/FragmentService.h>


using namespace Ieee80211::MpduCoordination;


/* #pragma mark - service interface implementation */


/*! Process a mac data frame.
    Successful return of this method does only mean, that the service is going
    to process the frame, not that it was processed already.

    \param frame to be processed.
    \return B_OK Service is going to process the frame.
    \return Else Service can't process the frame, due to an unrecoverable error.
 */
status_t
FragmentService::Process(MacData::Frame* frame)
{
    return fProcessDataFrameQueue.Enqueue(frame);
}


/*! Process a mac management frame.
    Successful return of this method does only mean, that the service is going
    to process the frame, not that it was processed already.

    \param frame to be processed.
    \return B_OK Service is going to process the frame.
    \return Else Service can't process the frame, due to an unrecoverable error.
 */
```

```
52  status_t
53  FragmentService::Process(MacManagement::Frame* frame)
54  {
55      return fProcessManagementFrameQueue.Enqueue(frame);
56  }
57
58
59  /*! Puts the service for processing mac data frames in its run state.
60
61      \param dummy never used. This is only required to allow different _Run()
62          methods per service in a common way.
63      \return B_OK The service terminated gracefully.
64      \return Else Unrecoverable error during service execution.
65   */
66  status_t
67  FragmentService::_Run(MacData::Frame* dummy)
68  {
69      MacData::Frame* frame = NULL;
70      status_t status = B_OK;
71
72      do {
73          status = fProcessDataFrameQueue.Dequeue(&frame);
74          if (status != B_OK)
75              break;
76
77          status = _FragmentAndEncryptFragments(frame);
78      } while (status == B_OK);
79
80      return status;
81  }
82
83
84  /*! Puts the service for processing mac management frames in its run state.
85
86      \param dummy never used. This is only required to allow different _Run()
87          methods per service in a common way.
88      \return B_OK The service terminated gracefully.
89      \return Else Unrecoverable error during service execution.
90   */
91  status_t
92  FragmentService::_Run(MacManagement::Frame* dummy)
93  {
94      return B_OK;
95  }
96
97
98  /* #pragma mark - private */
99
100
101 /*! If frame length is greater than dot11FragmentationThreshold divide the frame
102     in multiple frames of length dot11FragmentationThreshold.
103     Hand the fragments over to the distribution coordinate service afterwards.
104
105     \param frame to be fragmented eventually and distributed afterwards.
```

```
106     \return B_OK The frame was successfully handed over to the distribution
107         coordination stage.
108     \return Else The frame wasn't processed, due to an unrecoverable error.
109  */
110 status_t
111 FragmentService::_FragmentAndDistributeFragments(Frame* frame)
112 {
113     return B_OK;
114 }
115
116
117 /*! If frame length is greater than dot11FragmentationThreshold divide the frame
118     in multiple frames of length dot11FragmentationThreshold.
119     Encrypt the frame/frames afterwards.
120
121     \param frame to be fragmented eventually and encrypted afterwards.
122     \return B_OK The frame was successfully handed over to the encryption
123         stage.
124     \return Else The frame wasn't processed, due to an unrecoverable error.
125  */
126 status_t
127 FragmentService::_FragmentAndEncryptFragments(Frame* frame)
128 {
129     return fEncryptService.Process(frame);
130 }
```

D. Datenträgerinhalt

Datenträger

- **Literatur**
  - Cryptanalysis of IEEE 802.11i TKIP.pdf
  - Downloadzahlen für Haiku R1Alpha1.html
  - Haiku - A Brief Introduction to our Source Repository Layout.pdf
  - Haiku - Network Stack Architecture.pdf
  - Haiku - System calls.pdf
  - Linux - mac80211_hwsim.pdf
  - Request For Comments - 1155.txt
  - Sam Leffler - Wireless Networking in the Open Source Community - SANE2006.pdf
- **Sourcecode**
  - **FreeBSD 8**
    Kompletter Quellcode der Subversion Revision 199625
  - **Haiku**
    Kompletter Quellcode der Subversion Revision 35765
  - **Mac OS X 10.6**
    Headerdateien der WLAN-Stack-API
  - **Prototyp**
    - **ieee80211**
      Kompletter Quellcode der Subversion Revision 35579
    - Dokumentation.pdf
      Doxygen-Dokumentation des Prototypen
  - **Windows 7**
    Headerdateien der WLAN-Stack-API
- Masterarbeit.pdf

Literaturverzeichnis

[Style] **2009 IEEE Standards Style Manual.** *The Institute of Electrical and Electronics Engineers.* 2009. Online unter: `http://standards.ieee.org/guides/style/2009_Style_Manual.pdf`. Stand: 02.02.2010.

[Std80211] **IEEE Standard for Information technology.** Telecommunications and information exchange between systems. Local and metropolitan area networks. Specific requirements. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Std 802.11-2007. Revision of IEEE Std 802.11-1999. *The Institute of Electrical and Electronics Engineers.* New York. 12. Juni 2007. Online unter: `http://standards.ieee.org/getieee802/download/802.11-2007.pdf`. Stand: 13. März 2009.

[Std802] **802.** IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture. *The Institute of Electrical and Electronics Engineers.* New York. 7. Februar 2002. Online unter: `http://standards.ieee.org/getieee802/download/802-2001.pdf`. Stand: 13. März 2009.

[WiFi] **Wireless LANs.** 802.11-WLAN-Technologie und praktische Umsetzung im Detail. 802.11a/h. 802.11b. 802.11g. 802.11i. 80211n. 802.11d. 802.11e. 802.11f. 802.11s. 3., aktualisierte und erweiterte Auflage. *Jörg Rech.* Heise Zeitschriften Verlag. Hannover. 2008.

[Ether] **Ethernet.** Technologien und Protokolle für die Computervernetzung. Standard-Ethernet. Fast Ethernet. Gigabit-Ethernet. 10Gigabit-Ethernet. Power over Ethernet. 2., aktualisierte und überarbeitete Auflage. *Jörg Rech.* Heise Zeitschriften Verlag. 2008.

[Attacks] **Cryptanalysis of IEEE 802.11i TKIP.** *Finn Michael Halvorsen, Olav Haugen.* NTNU, Norwegian University of Science and Technology. Department of Telematics. Trondheim. 2009. Online unter: `http://download.aircrack-ng.org/wiki-files/doc/tkip_master.pdf`. Stand: 11. Februar 2010. Auch auf Datenträger.

[Std8023] **IEEE Standard for Information technology.** Telecommunications and information exchange between systems. Local and metropolitan area networks. Specific requirements Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. SECTION ONE. IEEE Std 802.3-2005.

The Institute of Electrical and Electronics Engineers. New York. 9. Dezember 2005. Online unter: http://standards.ieee.org/getieee802/download/802.3-2005_section1.pdf. Stand: 13. März 2009.

[Pragma] **Der Pragmatische Programmierer.** *Andrew Hunt, David Thomas.* Carl Hanser Verlag. München/Wien. 2003.

[Muster] **Entwurfsmuster.** Elemente wiederverwendbarer objektorientierter Software. 1. Auflage. 5., korrigierter Nachdruck. *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.* Addison Wesley Verlag. München, Bosten und andere. 2001.

[SoftArch] **Software-Architektur.** Grundlagen, Konzepte, Praxis. 2. Auflage. *O. Vogel et. al.* Spektrum Akademischer Verlag. Heidelberg. 2009.

[RFC1155] **RFC, Request For Comments.** Nummer 1155. Structure and Identification of Management Information for TCP/IP-based Internets. *M.T. Rose, K. McCloghrie.* IETF, Internet Engineering Task Force. 1990. Online unter: `http://www.ietf.org/rfc/rfc1155.txt`. Stand: 24 April 2010. Auch auf Datenträger.

[Clean] **Clean Code.** A Handbook of Agile Software Craftsmanship. *R. C. Martin et. al.* Prentice Hall. München, Bosten und andere. 2009.

[Analyse] **Objektorientierte Analyse und Design.** Mit praktischen Anwendungsbeispielen. 1., korrigierter Nachdruck. *Grady Booch.* Addison-Wesley. Bonn, Paris. 1995.

[DDD] **Domain-Driven Design.** Tackling Complexity in the Heart of Software. *Eric Evans.* Addison-Wesley. München, Bosten und andere. 2004.

[Balzert] **Lehrbuch der Software-Technik.** Software-Entwicklung. 2. Auflage. *Helmut Balzert.* Spektrum, Akademischer Verlag. Heidelberg, Berlin. 2000.

[SoftEng] **Software Engineering.** Fifth Edition. *Ian Sommerville.* Addison-Wesley Publishing Company Inc. Harlow, Reading, u.a. 1996.

[Modell] **Modellgetriebene Softwareentwicklung.** Techniken, Engineering, Management. 2., aktualisierte und erweiterte Auflage. *Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase.* dpunkt.verlag. 2007.

[Showstop] **Showstopper!** The Breakneck Race to Create Windows NT and the Next Generation at Microsoft. *G. Pacal Zachary.* Free Press. New York. 1994.

[Down] **Downloadzahlen für Haiku R1/Alpha1.** *Haiku Inc.* Online unter: `http://ryanleavengood.com/haiku/downloadresults.php`. Stand: 03. März 2010. Auch auf Datenträger.

[BeBook] **BeBook.** *ACCESS Co. Ltd.* 2001. Online unter: `http://www.haiku-os.org/legacy-docs/bebook/index.html`. Stand: 03. März 2010.

[HaiNet] **Haiku Network Stack Architecture.** *Axel Dörfler.* 2008. Online unter: `http://www.haiku-os.org/documents/dev/haiku_network_stack_architecture`. Stand: 24. April 2010. Auch auf Datenträger.

[HaiCode] **Haiku Quellcode.** SVN Revision 35765. *Haiku Inc. und andere.* 2010. Online unter: `http://dev.haiku-os.org/browser/haiku/trunk?rev=35765`. Stand: 04. März 2010. Auch auf Datenträger.

[BSDCode] **FreeBSD 8 Quellcode.** SVN Revision 199625. *The FreeBSD Project und andere.* 2009. Online unter: http://svn.freebsd.org/viewvc/base/release/8.0.0/. Stand: 24. April 2010. Auch auf Datenträger.

[RepoLay] **A Brief Introduction to our Source Repository Layout.** *Axel Dörfler.* Haiku Inc. 17. Mai 2007. Online unter: `http://www.haiku-os.org/documents/dev/a_brief_introduction_to_`↩`our_source_repository_layout`. Stand: 18 April 2010. Auch auf Datenträger.

[Syscalls] **System calls.** *romain.* Haiku Inc. 09. Dezember 2008. Online unter: `http://www.haiku-os.org/documents/dev/system_calls`. Stand: 18. April 2010. Auch auf Datenträger.

[Linux] **mac80211_hwsim.** Linux Wireless. 2010. Online unter: `http://wireless.kernel.org/en/users/Drivers/mac80211_hwsim`. Stand: 24. April 2010. Auch auf Datenträger.

[Win7] **wlanapi.h und andere.** Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 Service Pack 1. *Microsoft Corporation.* 2004. Stand: 24. April 2010. Auch auf Datenträger.

[MacOSX] **CoreWLAN.h und andere.** Xcode 3.2.1 Developer Tools. *Apple Inc.* 2009. Stand: 24. April 2010. Auch auf Datenträger.

[Leffler] **Wireless Networking in the Open Source Community.** *Sam Leffler.* SANE 2006. Online unter: `http://people.freebsd.org/~sam/SANE2006-Wireless.pdf`. Stand 18. April 2010. Auf auf Datenträger.

[Prototyp] **Prototyp-Quellcode zu objektorientierten WLAN-Stack-Entwurf.** SVN Revision 35579. *Haiku Inc., Colin Günther.* 2010. Online unter: `http://dev.haiku-os.org/log/haiku/branches/developer/colin/wire`↩`less/src/add-ons/kernel/network/devices/ieee80211/stack?rev=35579`. Auch auf Datenträger.