Implementierung und Evaluierung von Token- und FDMA-basierten MAC-Layer-Protokollen für 802.11

Martin Chauchet
Fachbereich Informatik
Hochschule Bonn-Rhein-Sieg
Grantham-Allee 20, Sankt Augustin
Email: martin.chauchet@smail.inf.h-brs.de

Erstprüfer: Prof. Dr. Karl Jonas
Email: karl.jonas@inf.h-brs.de

Zweitprüfer: M. Sc. Michael Rademacher
Email: michael.rademacher@inf.h-brs.de

24. April 2015

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Zusammenfassung

In der vorliegenden Arbeit wird der Entwurf und die Implementierung zweier alternativer Medienzugriffskontrollen, FDMA und Token Passing, für Punkt-zu-Punkt-Verbindungen auf der Basis von 802.11 beschrieben. Dabei wird zunächst die Notwendigkeit einer alternativen Medienzugriffskontrolle heraus- und bereits existierende Arbeiten zu diesem Thema vorgestellt. Um die Implementierung zu vereinfachen, wird anhand zuvor vorgestellter Kriterien ein Framework ausgewählt, dass den Entwicklungsprozess unterstützen soll. Hierzu werden zunächst verschiedene Frameworks vorgestellt und deren Eignung anhand der vorgenannten Kriterien überprüft. Es folgt eine Beschreibung der zur Implementierung eingesetzten Testumgebung sowie des ausgewählten Frameworks "Click Modular Router". Nachdem so die Arbeitsumgebung vollständig dokumentiert ist, erfolgt der Entwurf und die Implementierung der beiden Protokolle mithilfe des ausgewählten Frameworks. Im Anschluss dessen kann die Leistungsfähigkeit der beiden Protokolle im Bezug auf die zur Verfügung stehende Datenrate erhoben und ausgewertet werden. Dabei zeigt sich, dass sowohl die FDMA- als auch die Token Passing-Lösung hinter den Erwartungen zurückbleibt. Allerdings kann auch beobachtet werden, dass die Haltezeit des Tokens dazu genutzt werden kann, um eine feste (A-)Symmetrie von Up- und Downstream einzustellen. Die Ergebnisse lassen in jedem Fall Raum für weitere Forschungsarbeiten.

Kapitel 1

Einleitung

Wie die Verbreitung des Personal Computers, so ist auch die Erfolgsgeschichte des Internet beziehungsweise des World Wide Web als dessen Dienst für digitale Inhalte eng verknüpft weitgehenden Durchdringung des beruflichen ebenso wie des privaten Alltags. Nicht zuletzt auch aufgrund des gebotenen Unterhaltungs- und informationellen Mehrwerts: Soziale Netzwerke ebenso wie Video- und Multimediaplattformen ermöglichen ihren Nutzern den schnellen und unkomplizierten Austausch von Informationen, Erlebnissen und Erfahrungen über politische und geografische Grenzen hinweg. Darüber hinaus machen Cloud-Dienste persönliche Daten überall dort verfügbar, wo eine Internetverbindung vorhanden ist. Als Folge dessen ist die Erwartungshaltung der Nutzer an einen zeit- und ortsunabhängig verfügbaren Zugang zu Informations- und Kommunikationsdiensten in den letzten Jahren deutlich gestiegen; das so genannte "Web 2.0" wird als eine Selbstverständlichkeit erachtet.

Das Internet als technologische Stütze hinter eben diesen Informationsdiensten muss sich an diese neuen und sich in immer kürzeren Zyklen ändernden Anforderungen anpassen können. Zu diesen Anforderungen gehört beispielsweise die Verfügbarkeit von Zugängen mit hohen Datenraten sowohl im städtischen als auch insbesondere im bis dato nur unzureichend bedachten ländlichen Raum. Während in dicht besiedelten Gebieten - sogar in Schwellen- und Entwicklungsländern - die Versorgung üblicherweise ausreichend bis gut ist, gilt dies nicht für den vergleichsweise dünn besiedelten ländlichen Sektor. Der Breitbandatlas der Bundesregierung ([8]) zeigt deutlich noch weiße Flecken auf der digitalen Landkarte der eigentlich infrastrukturell starken Industrienation Deutschland. Ursächlich für diese digitale Kluft ist die gewinnorientierte Expansionsstrategie der als Zugangsanbieter auftretenden Internet Service Provider (ISP).

Im Allgemeinen können die Kosten bzw. Aufwände für Telekommunikationsausrüstung unterteilt werden in einmalige Aufwendungen für Ankauf und Inbetriebnahme, so genannte "*CAP*ital *EX*penditures" (CAPEX), und wiederkehrende oder permanente Kosten für die Aufrechterhaltung des Betriebes, gemeinhin als "*OP*erational *EX*penditures" (OPEX) bezeichnet. In Summe bilden CAPEX und OPEX dann die Gesamtaufwendungen (*TOT*al *EX*penditures, (TOTEX)). Eine wichtige Rolle bei der Bereitstellung von Zugängen mit hohen Datenraten nehmen kabelgebundene Anschlüsse auf der Basis von Digital Subscriber Line (DSL) ein, die spezialisierte und kostenintensive Ausrüstung auf Seiten der ISPs voraussetzen, zum Beispiel Outdoor-DSLAMs (DSL Access Multiplexer). Weiterhin werden qualitativ hochwertige Kabel auf Kupfer- oder Glasfaserbasis benötigt, die zumeist unterirdisch verlegt werden. In gut erschlossenen Gebieten ist dafür üblicherweise bereits Hilfsinfrastruktur in Form von Kabeltunnel vorhanden. Zumindest aber sind die zu überbrückenden Strecken zwischen Unterverteilung/Vermittlungsstelle und Übergabepunkt hinreichend kurz, um eine kosteneffiziente Verlegung von Kabeln in Tiefbautechnik zu ermöglichen. Außerdem machen Technologien wie Vectoring und G.fast bereits vorhandene Doppelader-Kabel auch für hohe Datenraten nutzbar.

In den vergleichsweise dünn besiedelten ländlichen Regionen sind diese Kupferkabel zwar auch vorhanden, allerdings spielen aufgrund der oft großen Distanzen deren physikalische Eigenschaften eine deutlich größere Rolle. Der Dämpfungsbelag des Kabels limitiert die maximale Kabellänge und das Tiefpassverhalten des Kupfers beschränkt die nutzbare Frequenzbandbreite nach oben hin. Infolgedessen eignen sie sich auf langen Strecken nur sehr eingeschränkt für hohe Datenraten. Alternative Technologien, die auf Glasfasermedien aufbauen (Fibre To The x (FTTx)), schaffen zumindest technologische Abhilfe. Die zur Verlegung dieser Kabel notwendigen Erdarbeiten über

mehrere Dutzend Kilometer verteuern allerdings die Initialkosten (CAPEX) nicht unerheblich und folglich scheuen ISPs den für sie unrentabel erscheinenden Ausbau oder sogar die Erstausrüstung potentieller Endkunden.

Eine etablierte Lösung für dieses Problem ist die Implementierung von drahtlosen Langstreckenfunkverbindungen (Wireless Long Distance (WiLD)) als Ersatz für in der Erde verlegte Kabel wo immer nötig, um auch an abgelegenen Standorten eine Internetanbindung mit hohen Datenraten zu ermöglichen. Die so entstehenden WiLD-Netzwerke agieren als "digitales Rückgrat" (Backhaul) zwischen Standorten, die über keine oder nur eine langsame Anbindung verfügen, und solchen mit schneller Internetverbindung. Als Basis dienen handelsübliche Netzwerkgeräte, die zu erschwinglichen Kosten erhältlich sind, und standardisierte Übertragungsprotokolle wie IEEE 802.16 (WiMAX) oder IEEE 802.11 (Wireless LAN, WiFi). In dieser Arbeit steht wegen des hohen Verfügbarkeits- und Verbreitungsgrades 802.11 im Fokus.

Auf Punkt-zu-Punkt-Verbindungen, die ein häufig anzutreffender Bestandteil von WiLD-Netzwerken sind, ist das Verhalten der Medienzugriffssteuerung (MAC Layer) von 802.11 allerdings nicht optimal, wie diverse Forschungsinitiativen in diesem Bereich gezeigt haben. Diese basiert auf dem Verfahren Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA), das eng verwandt ist mit dem aus IEEE 802.3 (Ethernet) bekannten CSMA/CD (CSMA mit Collision Detection), und wird in ihrer Gesamtheit als Distributed Coordination Function (DCF) bezeichnet. Durch die für die Vermeidung von Kollisionen nötigen Protokollabläufe wird ein Großteil der zur Verfügung stehenden Übertragungszeit auf Medienabtastung und Wartezeiten verwendet. Das ist wichtig bei vielen gleichzeitig das Medium benutzenden Stationen, genießt bei Punkt-zu-Punkt-Verbindungen mit genau zwei konkurrierenden Knoten allerdings nachgelagerte Priorität. Zusätzlich führt die mit der Entfernung skalierende Ausbreitungszeit (engl. propagation time) zu Überläufen protokollinterner Timeout-Mechanismen und somit zu unnötig vielen erneut übertragenen Paketen (Retransmissions).

Einer der ersten Schritte war bislang, diese Protokollinterna zur Laufzeit zu optimieren, beispielsweise die Anpassung von Timermechanismen in Abhängigkeit zur Distanz oder gar die komplette Abschaltung der Bestätigungslogik. Außerdem ließen sich die für Punkt-zu-Punkt-Verbindungen ungünstigen Standardvorgaben innerhalb der DCF justieren, wie in der Arbeit von Rademacher et al. [9] geschehen. Dies führte bereits zu einer signifikanten Steigerung der Leistungsfähigkeit der Übertragungsstrecke. Um aber eine noch höhere Saturierung der zur Verfügung stehenden Übertragungszeit zu erreichen, war und ist der Austausch der Standard-Medienzugriffskontrolle durch eine angepasste Eigenentwicklung unumgänglich. Hier boten sich zwei verschiedene Mechanismen an: Der häufig gewählte Zeitmultiplex-Ansatz (Time Division Multiple Access (TDMA)) teilt das Medium auf der Zeitachse zwischen den beteiligten Stationen auf, während im Frequenzmultiplex die zur Übertragung verfügbare Frequenzbandbreite in zueinander orthogonale Übertragungskanäle zerlegt wird (Frequency Division Multiple Access (FDMA)). Als Ableger von TDMA kann Token Passing betrachtet werden. Hierbei regelt die Weitergabe eines definierten Datenpaketes als eine Art Redestein den Zugriff auf das Medium. Die Station, die jeweils den Token hält, hat schreibenden Zugriff auf das Medium, alle anderen nur lesenden. Nach jeder erfolgreichen Übertragung wird der Token zur jeweils nächsten Station weitergereicht. Eine detaillierte Beschreibung zu TDMA, FDMA und Token Passing ist im Abschnitt 2.2 zu finden.

Zum aktuellen Zeitpunkt ist bereits eine Vielzahl von Protokollen vorgestellt worden, die jeweils für sich in Anspruch nehmen, eine potenzielle Lösung für die oben beschriebenen Probleme darzustellen. Wie bereits vom Autor dieser Arbeit in einem früheren Paper ([10]) gezeigt wurde, erfüllt allerdings keines dieser Protokolle alle vom Autor vorgestellten Kriterien und Anforderungen vollständig. Es liegt also nahe, ein Protokoll zu entwerfen, das zwecks Optimierung von Durchsatz und Delay alle genannten Kriterien erfüllt. In dieser Arbeit wird als Vorarbeit dazu ein Machbarkeitsnachweis erbracht, sowohl für eine Token- als auch für eine FDMA-basierte Medienzugriffsschicht für 802.11n. Dabei wird, sofern möglich, die Verwendung eines bereits existierenden Frameworks favorisiert. Dies verkürzt im Idealfall die Entwicklungsarbeit, weil viele benötigte Funktionen oder Subroutinen schon implementiert sind.

In Kapitel 2 werden zunächst die verwandten Arbeiten aus dem Bereich der drahtlosen Langstreckenverbindungen und alternativen Medienzugriffsprotokolle vorgestellt. Darauf folgt in Kapitel 3 die Vorstellung und Auswahl eines Entwicklungsframeworks. Dazu werden einige Kriterien vorgestellt und die Protokolle gegen diese abgeglichen. Das ausgewählte Framework wird dann noch einmal tiefer gehend erläutert. Die Testumgebung, auf der die Entwicklung und Evaluierung erfolgt, wird in Kapitel 4 vorgestellt. Das schließt insbesondere auch die einzelnen Arbeitsschritte bis zur

vollständigen Inbetriebnahme mit ein. Kapitel 5 beschreibt die Entwicklung eines Token- und eines FDMA-basierten Protokollansatzes. Dabei wird differenziert zwischen theoretischem Protokolldesign und praktischer Implementierung. In Kapitel 6 werden die zwei Protokolle gegenübergestellt und die Leistungsfähigkeit der beiden erhoben und bewertet. Um Vergleichbarkeit zu erreichen, wird zusätzlich die Leistungsfähigkeit der verwendete Funkstrecke mit dem Standard-Medienzugriff DCF gemessen. Die Ausarbeitung schließt mit Kapitel 7 und der dort enthaltenen Zusammenfassung. Hier fügt sich noch ein Ausblick auf künftige Arbeiten an.

Kapitel 2

Verwandte Arbeiten und Hintergrund

In diesem Kapitel soll zunächst ein kurzer Überblick über verwandte und bereits geleistete Arbeiten im Bereich der alternativen Medienzugriffs-Protokolle für 802.11 gegeben werden. Dazu werden zunächst unterschiedliche Verfahren des Medienzugriffs beschrieben. Es folgen diverse Protokolle, die nach eigenen Angaben bereits eine Optimierung der Übertragungsleistung auf Langstreckenfunkverbindungen erreicht haben sollen. Schließlich werden verschiedene Arbeiten über die Architektur des Treibersubsystems für drahtlose Netzwerkadapter unter Linux vorgestellt.

2.1 Leistung des 802.11 MAC

Wie bereits in der Einleitung angeführt, fügt sich die im Standard IEEE 802.11 geforderte Medienzugriffskontrolle entwurfsbedingt nicht hinreichend gut in die durch Langstrecken-Punkt-zu-Punkt-Verbindungen vorgegebenen Bedingungen ein. Dennoch wurde in [9] bereits gezeigt, dass durch Optimierungen im operativen Betrieb bereits eine signifikante Leistungssteigerung erreicht werden kann. Außerdem wurde die Verwendung kreuzpolarisierter Antennen zur Nutzbarmachung der im Zusatz 802.11n eingeführten Mehrwege-Technologie MIMO (Multiple Input Multiple Output) thematisiert. So wurden zwei unabhängige Datenströme zwischen zwei Stationen parallel übertragen, was erneut zur Steigerung der Übertragungsleistung beiträgt.

2.2 Verfahren zum Mehrfachzugriff

Um mehreren Teilnehmern einen geordneten Zugriff auf das selbe Medium zu ermöglichen, ohne dass sich diese gegenseitig stören, ist eine Medienzugriffskontrolle (Media Access Control (MAC)) obligatorisch. Hierfür kann das Medium auf unterschiedliche Arten ein- und den partizipierenden Stationen zugeteilt werden. Diese unterschiedlichen Zugriffsmethoden, wie sie unter anderem bei Kauffels [11], Orlamünder [12] oder Kurose und Ross [13] zu finden sind, werden im Folgenden beschrieben.

Wird das Medium in auf der Zeitachse eingeteilt, so spricht man von einem Zeitmultiplexverfahren (TDMA). Die zur Verfügung stehende Übertragungszeit wird dabei in Zeitschlitze fester oder variabler Länge unterteilt, die dann von den einzelnen Teilnehmern belegt werden. Jedem Teilnehmer wird der schreibende Medienzugriff dabei für einen dezidierten Zeitraum ermöglicht. In der restlichen Zeit erhält er lediglich lesenden Zugriff (vgl. hierzu auch [13], S. 47f., 491ff.). Die Zuweisung der Zeitschlitze an die teilnehmenden Stationen kann von einer zentralen Instanz kontrolliert und dann den Teilnehmern mitgeteilt werden. Alternativ können die Teilnehmer die Belegung unter sich aushandeln. In beiden Fällen ist eine strikte Zeitsynchronisation zwischen den Teilnehmern essentiell für eine effiziente Ausnutzung der verfügbaren Übertragungszeit. Wenn alle Teilnehmer Beginn und Ende jedes Zeitschlitzes genau identifizieren können, kann auf lange Schutzintervalle zwischen den Übertragungen zugunsten einer höheren Medienauslastung verzichtet werden. Problematisch ist allerdings die Art der Zeitsynchronisation, da diese über ein zuverlässiges Medium erfolgen muss, beispielsweise eine dedizierte Ethernetverbindung oder das Globale Positionierungs-

system (GPS). Die Zeitsynchronisation muss außerdem ständig erfolgen, um den Drift der in den Uhren der Teilnehmer verbauten Quarzkristalle zu kompensieren.

Als Alternative zu festen Zeitschlitzzuweisungen kann ein virtuelles Token, d.h. ein definiertes Datenpaket verwendet werden. Dieses wird zwischen den Stationen umher gereicht und garantiert ausschließlich dem Teilnehmer, der den Token über einen bestimmten Zeitraum hält, exklusiven schreibenden Zugriff auf das Medium innerhalb genau dieses Zeitraumes. Der Token wird nach Ablauf einer bestimmten Zeitspanne oder nach Erreichen eines Datenlimits an den nächsten Teilnehmer weitergereicht. Die Gesamtheit dieses Verfahrens wird auch als Token Passing bezeichnet (vgl. [13] S. 500f.).

Ein Beispiel für ein TDMA-basiertes Protokoll ist die Plesiochrone Digitale Hierarchie (PDH) und ihr Ableger, das Integrated Services Digital Network (ISDN), die auf Basis von Zeitschlitzen mehrere voneinander unabhängige paket- und leitungsvermittelte Datenströme voneinander trennen. Token Bus und Token Ring sind prominente Vertreter Token Passing-basierter Protokolle, auch wenn sie für den Stand der Technik im Bereich Netzwerkprotokolle keine Bedeutung mehr haben.

Wird stattdessen Frequenzmultiplex auf ein Medium angewandt, so wird dieses nicht auf der Zeit-, sondern auf der Frequenzachse geteilt. Das heißt, die zur Verfügung stehende Frequenzbandbreite wird in entsprechend viele Subkanäle aufgeteilt wie Kommunikationswegen zwischen Teilnehmern benötigt werden (vgl. hierzu [13] 47f., 491ff.). Dadurch wird jedem Teilnehmer ein eigener Subkanal zur Verfügung gestellt, den er exklusiv zur Datenübertragung nutzen kann. Daraus resultiert eine zeit-simultane Vollduplex-Kommunikation zwischen den Teilnehmern, jedoch - bei gleicher spektraler Effizienz - mit verminderter Bandbreite und infolgedessen in der Regel auch verminderter Datenrate. Da Funkmodule für 802.11 in der Regel im Halbduplex-Betrieb arbeiten, sind also je Vollduplexstrecke und Knoten zwei Funkmodule notwendig. Orthogonales Frequenzmultiplex (Orthogonal Frequency Division Multiplex (OFDM)) ist eine Variante des FDMA. Hier wird der verfügbare Kanal in viele schmale Subkanäle geteilt. Tritt während der Übertragung eine schmalbandige Störung auf, wird nur die Übertragung auf einem oder einigen wenigen Subkanälen gestört, nicht aber auf dem gesamten Kanal. OFDM wird beispielsweise bei 802.11 auf der Schicht 1 (PHY) und bei Digital Subscriber Line (DSL) eingesetzt.

Bei beiden beschriebenen Verfahren ist grundsätzlich die Möglichkeit gegeben, das Protokollverhalten an unterschiedliche Dienstgüteklassen und Verkehrsbedürfnisse anzupassen. Bei TDMA können Scheduling-Verfahren eingesetzt werden, die je Verkehrsklasse abgestimmte Zeitschlitze verwenden. Außerdem kann sowohl bei TDMA als auch bei FDMA das Verhältnis von Hin- zu Rückkanal durch Zuweisen von entsprechend mehr oder weniger Übertragungszeit beziehungsweise Subkanalbreite anhand der (A-)Symmetrie der Kommunikation eingestellt werden. Eine solche Asymmetrie weist beispielsweise der Verkehr zwischen Endkunde und Internet Service Provider auf, der in der Regel einen deutlich größeren Downstream- als Upstreamanteil enthält.

2.3 Existierende MAC-Protokolle für WiLD

In der Veröffentlichung von I. Hussain et al. [14] wurden bereits diverse TDMA-MAC-Protokolle für Langstreckenfunkverbindungen vorgestellt, einschließlich der Protokolle 2P [15], WiLDNet [16], JaldiMAC [17], JazzyMAC [18] und der Arbeit von Dhekne et al. [19].

2P [15] und dessen Abkömmling WiLDNet [16] basieren zur Vermeidung von Selbstinterferenz auf zweiphasigem Synchronbetrieb (Two phase synchronous operations). Das heißt, die Menge aller teilnehmenden Knoten im Netzwerk wird in zwei Untermengen aufgeteilt, wobei die eine Untermenge in der ersten Phase sendet und in der zweiten im Empfangsmodus ist und die zweite jeweils umgekehrt agiert. Folglich wird ein bipartiter Topologiegraph benötigt. Das schränkt die Flexibilität, Skalierbarkeit und Nutzbarkeit deutlich ein. Dieser Aspekt wird von JazzyMAC [18] addressiert. Dieses erlaubt überlappende Übertragungen und verbessert die Kontrolle über den Trade-Off zwischen Durchsatz und Verzögerung. Zeitschlitze mit variabler Länge zur Anpassung an den Verkehrsbedarf gehören ebenfalls zur Liste der Verbesserungen.

JaldiMAC [17] propagiert TDMA auf Punkt-zu-Mehrpunkt-Verbindungen für Wireless Internet Service Provider (WISP). Es bietet ebenfalls eine Anpassung an den Verkehrsbedarf, beispielsweise Asymmetrie oder Dienstgüteklassen mit Vorgaben an Verzögerung und Jitter. In [19] arbeiten Ashutosh Dhekne und seine Mitautoren mit einem Zeitmultiplex-Verfahren mit fester Zeitschlitzlänge und strikter enger Zeitsynchronisation. Dabei wird die Vergabe der Zeitschlitze an die teilnehmenden Stationen von einem zentralen Knoten geregelt. Die Übertragung von Verwaltungsinforma-

tionen zu beziehungsweise von diesem zentralen Knoten erfolgt dabei jeweils in dedizierten und getrennt vom Nutzdatenstrom. Sam Leffler von Errno Consulting hat in [20] aufgezeigt, inwiefern FreeBSD bereits "ab Werk" den Einsatz von TDMA auf drahtlosen Verbindungen unterstützt. Nach Angaben des Autors ist es möglich, ein TDMA-basiertes Netzwerk mit Betriebssystemmitteln aufzubauen, ohne dass Zusatzsoftware oder Treibermodifikationen nötig sind.

SoftToken ist, wie der Name suggeriert, ein Protokoll, das mit Token Passing arbeitet. Es wird von Lucas Eznarriaga et al. in [21] vorgestellt. Aufgrund seines Designs ist es primär für Punkt-zu-Mehrpunkt-Verbindungen ausgelegt, bei denen ein Master-Knoten die zentrale Koordination übernimmt. Das als US-Patent 6,795,418 ([22]) von Sunghyun Choi geschützte Verfahren kombiniert als Hybrid herkömmliches TDMA mit einer Token Passing- und einer Polling-Funktionalität. Im Regelfall arbeitet das Protokoll mit festen, zugewiesenen Zeitschlitzen. Beendet ein Knoten seine Übertragung vor dem Ende seines Zeitschlitzes, kann dieser den nächsten Teilnehmer durch einen Token darüber benachrichtigen. Dadurch wird wertvolle Übertragungszeit nicht durch Warten verschwendet. Die Polling-Funktion kommt zum Einsatz, wenn ein Hidden-Terminal-Szenario vorliegt. Das bedeutet, dass der Token die nächste Station beispielsweise wegen zu großer Entfernung oder Funkschatten nicht erreichen kann. In diesem Fall empfängt der zentrale Knoten diesen Token und fragt seinerseits bei der als nächstes sende-berechtigten Station an (Polling).

Um Interferenzen zwischen sendenden und empfangenden Funkmodulen auf benachbarten Verbindungen zu vermeiden, ist eine korrekte Kanalzuweisung unumgänglich, sofern man nicht auf Behelfskonstruktionen wie SynOP zurückgreifen möchte. In [23] wird von Partha Dutta et al. ein Algorithmus zur Kanalzuteilung vorgestellt, der die Netzwerktopologie auf einen gerichteten Graphen abbildet. Dabei wird jeder Duplex-Link innerhalb des WiLD-Netzwerks als zwei gerichtete Kanten dargestellt, denen jeweils ein unterschiedlicher Kanal zugewiesen wird. Dies ermöglicht einen Vollduplex-Betrieb auf jedem logischen Link. Voraussetzung sind allerdings zwei Funkmodule je logischem Link, je eins zum Senden und zum Empfangen. Insgesamt wird das Problem der Kanalzuteilung auf das Knotenfärbeproblem reduziert, das nach den Ausführungen von Richard Karp [24] als $\mathcal{NP}$-vollständig bekannt ist.

Sowohl Duarte et al. [25] als auch Choi und seine Mitautoren [26] verwenden verschiedene Varianten der Signalauslöschung, um einen Vollduplex-Betrieb bei nur einem belegtem Kanal pro logischem Link zu ermöglichen. Das kann über das gesamte Netzwerk betrachtet zu einer Erhöhung der spektralen Effizienz führen, wenn ein effizienter Algorithmus zur Kanalzuteilung angegeben werden kann. Die Kanalzuteilung lässt sich auf das Problem der Kantenfärbung im ungerichteten Graphen reduzieren, dieses ist bekanntermaßen $\mathcal{NP}$-vollständig [24]. Erstmals wird in [26] das Prinzip der Antennenauslöschung vorgestellt. Dies setzt eine exakt phasenverschobene Platzierung der Antennen voraus, um schon auf dieser Ebene der Übertragung möglichst viel Interferenzauslöschung zu ermöglichen.

Das aus dem Mobilfunkbereich (GSM, UMTS) bekannte Spreizcodeverfahren wird durch Orfanos, Habetha und Liu in [27] auf 802.11 übertragen. Dabei werden in einer Punkt-zu-Mehrpunkt-Kommunikation die Datenströme der teilnehmenden Stationen mit paarweise orthogonalen Codes aufgespreizt. Dadurch verringert sich zwar die jeder Station zur Verfügung stehende Sendekapazität. Da dem zentralen Access Point jedoch alle Spreizcodes bekannt sind, kann dieser die simultan empfangenen Ströme wieder auftrennen und einzeln auswerten.

In [10] wurde bereits gezeigt, dass nach Ansicht des Autors keines der in diesem Abschnitt vorgestellten Protokolle alle Anforderungen an drahtlose Langstrecken-Punkt-zu-Punkt-Verbindungen vollständig erfüllt. Infolgedessen scheint es ratsam, einen eigenen Ansatz auf Token Passing-Basis zu verfolgen, der in dieser Ausarbeitung diskutiert werden soll.

2.4 Linux und 802.11

Wie andere Netzwerkschnittstellen beziehungsweise Hardwarekomponenten im Allgemeinen benötigen auch Adapter für 802.11 eine Treibersoftware, über die sie vom Betriebssystem angesprochen werden können. Das Engagement der Open-Source-Szene ist in diesem Fall sehr stark (beispielsweise der Community-getriebene MadWifi-Treiber für Atheros-Karten [28]) und beeinflusst auch die Lizenzpolitik der Unternehmen. So haben bereits viele Treiber als quelloffene Software Einzug gehalten in den Linux-Kernel bzw. dessen Treiberzweig.

Für 802.11 steht im Linux-Kernel die sogenannte Soft-MAC-Architektur bereit. Diese lagert bestimmte Funktionen aus der Sicherungsschicht (OSI-Schicht 2, MAC) der Treiber aus und zentralisiert diese in einem eigenen Sublayer. Dieser abstrahiert die hardwareabhängigen Treiber ge-

genüber dem restlichen darüber liegenden Netzwerkstack und führt so als eine Art Mapperklasse zu einer Standardisierung der Funktionsaufrufe gegenüber den darunterliegenden Treibern. Das entsprechende Modul im Linux-Kernel heißt "mac80211" und kann vom Userspace aus durch die Schnittstellen "cfg80211" und "nl80211" angesprochen werden.

Einen Überblick über die Architektur des Wireless-Subsystems für quelloffene Treiber innerhalb des Linux-Kernel geben M. Vipin und S. Srikanth in [29]. Dabei werden insbesondere die Treiber der Hersteller Atheros und Intel berücksichtigt und in diese Architektur eingeordnet.

Ähnlich geht F. Gringoli in [30] vor. Auch hier wird das Linux-Wireless-Subsystem samt des Soft-MAC in seine Bestandteile zerlegt und der Kommunikationsfluss unter diesen dargestellt. Dabei liegt der Fokus hierbei erneut auf Atheros, ergänzt durch den Broadcom B43-Treiber.

In [1] unternimmt Daniel Mur einen Deepdive in die Struktur der beiden Open-Source-Treiber ath5k und ath9k, die beide vom Hersteller Atheros stammen. Der Paketfluss vom Dienst-übergabepunkt zwischen Schicht 3 und 2 bis hin zur Übergabe an die physikalische Schnittstelle wird anhand der Klassen und Funktionen innerhalb des Quellcodes aufgeschlüsselt. Das schließt auch die Soft-MAC-Architektur mit ein, die ebenfalls Teil dieses Paketflusses ist. Interessant ist auch, wie der Sende- und Empfangsvorgang getrennt voneinander betrachtet werden.

Das Schlagwort "Raw Frame Injection" beschreibt die Möglichkeit, auf einer Netzwerkschnittstelle beliebige Pakete zu versenden. Dabei wird üblicherweise der Protokollstack nahezu vollständig umgangen; auch das Frameformat muss nicht mehr dem Standard der Schnittstelle entsprechen. Gunther et al. haben sich die Eignung verschiedener Open-Source-Treiber für Raw Frame Injection angesehen und diese beurteilt [31]. Unter anderem wurde auch getestet, ob tatsächlich alle Teile des Protokollstacks außer Kraft gesetzt werden oder ob einzelne Funktionen aktiv bleiben. Beispielsweise ließ sich beobachten, dass beim Treiber ath9k die Distributed Coordination Function weiterhin aktiv bleibt.

Kapitel 3

Frameworks für die Protokollentwicklung

Um die Entwicklungszeit für eine Softwarekomponente zu verkürzen und den Implementierungsaufwand zu verringern, stellt die Verwendung von Frameworks ein probates Mittel dar. In diesem Kapitel werden diverse Frameworks verglichen und bewertet, die auf die Entwicklung eigener Medienzugriffskontrollen ausgerichtet sind. Zunächst erfolgt die Festlegung von Kriterien für die Bewertung, anhand derer die Eignung der Frameworks bemessen wird, gefolgt von einer Beschreibung der Charakteristika der einzelnen Frameworks und einem tabellarischen Vergleich.

3.1 Kriterien

Zum Zeitpunkt des Projektstarts waren verschiedene Frameworks verfügbar, die sich auf die Entwicklung von alternativen Medienzugriffsprotokollen für 802.11 spezialisiert haben. Diese Frameworks werden im Verlauf des Kapitels nacheinander kurz vorgestellt. Zuvor werden einige Kriterien ausgearbeitet, die der Bewertung der Frameworks dienen sollen. Die Erläuterung dieser Kriterien befindet sich im kommenden Abschnitt.

3.1.1 Treiber- und Hardwareunterstützung

Eine Liste aller vom Linux-Kernel unterstützten Open-Source-Wireless-Treiber findet sich auf der Seite des Linux-Wireless-Projektes ([2]). Setzt man als Filterkriterium die Unterstützung von 802.11n und die Verfügbarkeit eines Monitor Modes an, so bleiben folgende Treiber übrig:

Driver Name	Vendor	Monitor Mode	Raw Frame Injection
ath9k	Atheros	•	•
brcmsmac	Broadcom	•	?
iwlwifi	Intel	•	•
mwl8k	Marvell	•	?
rt2800	Ralink	•	•

Tabelle 3.1: Vergleich von Open-Source-Wireless-Treibern (in Anlehnung an [2, 3, 4, 5, 6, 7])

Wenn als zusätzliches Kriterium noch die Unterstützung von Packet Injection gefordert wird, verbleiben die Treiber der Hersteller Atheros (ath9k), Intel (iwlwifi) und Ralink (rt2800).

Aufgrund des hohen Marktanteils, der breiten Community-Unterstützung und der Verfügbarkeit entsprechender Hochleistungs-WLAN-Adapterkarten (auch Referenzdesigns) wird eine Implementierung auf Basis von Atheros (beziehungsweise ath9k) bevorzugt.

Zur Atheros-Treiberfamilie gehören neben ath9k auch noch MadWifi, ath5k und ath10k. Dabei stellt MadWifi historisch betrachtet den Ursprung der ath*-Treiber. Der Herstellertreiber für die ersten Atheros-Chipsätze für 802.11 stand zunächst nicht quelloffen zur Verfügung. Infolgedessen begann Sam Leffler (Urheber des OpenBSD-Projektes) mit der Entwicklung eines quelloffenen

Linux-Treibers für eben diese WLAN-Karten, der zunächst auf dem weiterhin geschlossen Hardware Abstraction Layer (HAL) von Atheros aufsetzte. Später wurde auch ein quelloffener HAL als "OpenHAL" entwickelt.

Durch den Erfolg des MadWifi-Treibers, der bis heute noch in Forschungs- und Experimentalaufbauten verwendet wird, war Atheros im Zugzwang und veröffentlichte schließlich im Jahr 2008 den Treiber ath5k für 802.11a/b/g unter einer Open Source-Lizenz. Die Entwicklung des Treibers fand und findet im weiteren Verlauf hauptsächlich durch das Linux-Wireless-Team statt. Nachfolger des ath5k sind der bereits vorgestellte ath9k für 802.11n und ath10k für 802.11ac (vgl. hierzu auch [3]).

3.1.2 Performance

Obwohl sich die Leistungsfähigkeit eines Frameworks ohne tiefere Kentnisse über die genaue Funktionsweise nicht exakt vorhersagen lassen kann, gibt es dennoch einige Anhaltspunkte. Einer dieser Anhaltspunkte lautet "Kontextwechsel" und beschreibt den Sprung von Prozeduraufrufen zwischen den verschiedenen Schichten der x86-Systemarchitektur. Die letztere besteht symbolisch gesehen aus vier Ringen, die jeweils klar gegeneinander abgegrenzt sind. Die äußeren drei Ringe werden als User-Mode bezeichnet, der innerste als Kernel-Mode. Prozesse können immer ausschließlich innerhalb eines Ringes laufen, nie über Ringgrenzen hinweg. Dabei werden den Prozessen immer höhere Berechtigungen zugeteilt, je näher sie dem Kernel-Mode, dem so genannten "Ring 0", kommen. Dort befinden sich üblicherweise der Betriebssystemkern und die Hardwaretreiber. Zur Kommunikation mit Prozessen in den anderen Ringen müssen Prozeduraufrufe bemüht werden. Linux (ebenso wie Windows) verwendet lediglich zwei der vier Ringe, Ring 0 für Kernel und Treiber und Ring 3 für Anwendungsprogramme und Dienste (vgl. hierzu auch Andrew S. Tanenbaum "Moderne Betriebssysteme" [32] S. 30).

Pro und Kontra User-Mode

Wenn nun ein Prozess aus dem Userspace auf ein Hardware-Gerät zugreifen möchte, muss er den Weg durch die Ringe bis zum Kernel-Space wählen und dort die Funktionsaufrufe der Treiber verwenden. Jeder Wechsel zwischen zwei benachbarten Ringen kostet aufgrund dieser mit der Kapselung verbundenen Funktionsaufrufe wertvolle Rechenzeit. Dies führt zwangsläufig, wenn auch nur in geringem Maße, zu einer Verschlechterung der Gesamtperformance gegenüber einem Funktionsaufruf innerhalb des Kernel- oder Userspace, da hier kein Wechsel zwischen verschiedenen Berechtigungsstufen auf CPU-Ebene nötig ist. Wenn folglich ein Framework für die Verwendung im User-Mode vorgesehen ist, könnte dies auf eine schlechtere Leistungsfähigkeit hindeuten, da eben mehr Prozeduraufrufe über Ringgrenzen hinweg anfallen. Details hierzu werden in den Veröffentlichungen von Jochen Liedtke [33] und John Ousterhout [34] diskutiert.

Auf der anderen Seite ermöglicht ein im User-Mode operierendes Framework eine besser Interaktion mit dem Benutzer. Konfigurationsänderungen können direkter vorgenommen werden, beispielsweise über Kommandozeilenbefehle, und das Starten und Stoppen des Frameworks ist ebenso einfach möglich, indem lediglich der betroffene Prozess beendet wird.

Pro und Kontra Kernel-Mode

Wenn ein Prozess oder ein Dienst im Kernel-Mode läuft, bedeutet das in der Regel, dass er dem Kernel als zusätzliches Modul zur Verfügung gestellt wird. Dieses kann bei Bedarf geladen werden, wenn die Funktionalität benötigt wird, die dieser Dienst zur Verfügung stellt. Ebenso kann das Modul wieder entladen werden, falls es nicht mehr gebraucht wird, vorausgesetzt es existieren keine Abhängigkeiten seitens anderer Module oder Dienste. Da das Modul im gleichen Prozessor-Ring läuft wie Kernel und Treiber, ist hier eine direkte Kommunikation möglich. Daraus könnte eine gegenüber dem User-Mode bessere Leistungsfähigkeit abgeleitet werden.

Die Konfiguration eines Kernelmoduls ist ohne weitere Hilfsmittel jedoch ungleich komplizierter als die eines User-Mode-Dienstes, weil keine direkte Interaktion mit dem Benutzer vorgesehen ist. Daher verfügen Kernelmodule häufig über zusätzliche Schnittstellen, über die sie vom User-Mode aus via Prozeduraufrufen gesteuert werden können. Ein Beispiel ist der Linux-Wireless-Stack, der (wie in Abschnitt 4.1.3 beschrieben wird) aus einem Kernelmodul und mehreren Helferklassen besteht, die eine Schnittstelle in den User-Mode bereitstellen. Über diese Schnittstelle kön-

nen Drahtlosnetzwerkadapter beispielsweise mit dem Kommandozeilenwerkzeug ”iw” und anderen Tools durch den Benutzer konfiguriert werden.

3.1.3 Programmierung

Die Programmiersprache beziehungsweise Art der Programmierung des Frameworks kann ebenfalls als Kriterium herangezogen werden. Dabei sollte unterschieden werden zwischen der Sprache, in der das Framework entwickelt wurde, und der Konfigurationssprache, die verwendet wird, um eigene Protokollentwürfe umzusetzen. Die Sprache kann allerdings auch bei beiden identisch sein. Klassische Programmiersprachen wie C oder C++, bei denen aus dem Quelltext im Compiler eine Binärdatei erstellt wird, versprechen durch nativen Maschinencode eine höhere Leistungsfähigkeit. Scriptsprachen wie Python oder Perl können hingegen mit Flexibilität bei Änderungen im Quellcode punkten, da dieser nicht nach jeder Anpassung neu compiliert, sondern durch einen Interpreter zur Laufzeit in Maschinenbefehle umgesetzt wird.

3.1.4 Relevanz

Um die praktische und wissenschaftliche Relevanz der einzelnen Frameworks bewerten zu können, werden außerdem die Anzahl der Zitationen in wissenschaftlichen Arbeiten sowie das Datum des letzten Beitrags zum Projekt selber (Commit) erhoben. Viele Zitationen weisen auf eine große theoretische und/oder praktische Bedeutung bzw. Verbreitung hin, während ein aktueller Commit Aktivität und Weiterentwicklung innerhalb des Frameworks verspricht. Als Referenzplattform für die Anzahl der Zitationen wird aufgrund der guten Zugänglichkeit Google Scholar verwendet. Der Internetauftritt des hinter dem jeweiligen Framework stehenden Projektteams dient als Quelle für den letzten Projektbeitrag.

3.2 Frameworks

3.2.1 SoftMAC

SoftMAC wurde von Michael Neufeld und seinen Mitautoren im Jahre 2005 in [35] vorgestellt. Erklärtes Ziel von SoftMAC ist es, im Sinne des Rapid Prototyping die Protokollentwicklung praxisnäher zu gestalten und von Simulationen hin zu realen Test- und Entwicklungsumgebungen zu schwenken. Im Idealfall unterstütze eine Entwicklungsplattform auf Basis von Software Defined Radio die Protokollentwickler, so die Autoren. Da SDR aber zum Zeitpunkt der Veröffentlichung für die Forscher noch unerschwinglich war, fiel die Wahl auf Standardhardware nach 802.11.

Bei der Entwicklung von SoftMAC berücksichtigten die Entwickler primär WLAN-Karten mit Chipsätzen des Herstellers Atheros, für die bis dato aber noch keine quelloffenen Herstellertreiber unter Linux zur Verfügung standen. MadWifi als Community-Treiber und der im Reverse Engineering des Atheros-HAL entschlüsselte OpenHAL (Hardware Abstraction Layer) bildeten folglich die quelloffene Basis für die Entwicklung von SoftMAC. SoftMAC verwendet den Monitor-Mode der Atheros-Karten zur Übertragung von Datenframes beliebigen Formates. Zusätzlich werden die Funktionen Virtual Carrier Sensing, Clear Channel Assessment und der Backoff Timer angegangen, um eine möglichst flexible Nutzung der WLAN-Schnittstellen zu ermöglichen. Die Autoren versuchen außerdem, die Leistungsfähigkeit und Flexibilität von SoftMAC zu demonstrieren, indem sie testweise zwei MAC-Protokolle auf SoftMAC implementieren. Bei diesen handelt es sich um einen TDMA- und einen adaptiven Reed-Solomon-Ansatz. Außerdem stellen die Autoren MultiMAC als Wrapper-Funktion vor. Sie erlaubt, zu Experimentalzwecken mehrere MAC-Layer parallel zu betreiben und zwischen diesen zu wechseln.

3.2.2 MadMAC

MadMAC ([36]), das in 2006 veröffentlich wurde, also etwa ein Jahr nach SoftMAC, setzt ebenfalls auf MadWifi auf. Entwickelt wurde es von Ashish Sharma, der mittlerweile für das Google-Android-Team arbeitet, mit dem gleichen Ziel wie SoftMAC, nämlich der Bereitstellung eines flexiblen Testbeds für Forschungen an alternativen MAC-Protokollen. Die Verwendung von gut verfügbaren standardisierten Hardwarekomponenten sichert die Kosteneffizienz des Testbeds.

Die Verwendung von MadMAC soll das Versenden von Paketen zu beliebigen Zeitpunkten und in beliebigen Formaten ermöglichen. Hierzu wurde der MadWifi-Treiber durch die Autoren

entsprechend erweitert. Wie bereits erwähnt, soll das Framework unabhängig sein von der Art der implementierten Protokolle, also Zeit-, Frequenzmultiplex oder anderen. Dennoch lag der Fokus bei der Entwicklung auf der Implementiertung eines Zeitmultiplexprotokolls. Dies ist auch in der Architektur des Frameworks in Ansätzen zu sehen (siehe Sharma et al. [37], Seite 5).

3.2.3 FreeMAC

Ebenfalls von Ashish Sharma entwickelt wurde das Framework FreeMAC, das in [38] im Jahre 2008 vorgestellt wurde. Es ist eine Weiterentwicklung von MadMAC und basiert wie dieses ebenfalls auf dem MadWifi-Treiber. Neue Funktionen beziehungsweise Merkmale von FreeMAC gegenüber seinem Vorgänger sind die Unterstützung von "regelmäßigen Kanalwechseln und genauere Kontrolle des Timing bei zu sendenden Paketen" ([38], S. 1). Zu Demonstrationszwecken und als Argumentationsbasis für Leistungsanalysen wurde ein Mehrkanal-TDMA-Medienzugriff implementiert.

Ashish Sharma arbeitet mittlerweile bei Google im Android-Entwicklungsteam. Seine Arbeiten im MOMENT-Labor (http://moment.cs.ucsb.edu/) an der Universität von Santa Barbara, Kalifornien, MadMAC und FreeMAC, sind dort leider nicht herunterladbar. Eine Kontaktaufnahme war nicht erfolgreich.

3.2.4 OverlayMAC

Mit dem Overlay MAC Layer (OML), vorgestellt in [39], wurde versucht, über der existierenden Medienzugriffsschicht von 802.11 eine weitere Zugriffskontrolle zu implementieren. Das bedeutet zum einen, dass keine oder nur wenige Abhängigkeiten zu hardwarenahen Funktionen in den Schichten 1 und 2 existieren. Außerdem sind keine Eingriffe in zentrale Linux-Kernelmodule nötig. Das wiederum erhöht die Kompatibilität und Portierbarkeit für künftige Kernelaktualisierungen und -änderungen.

Die Basis für den OML bildet der Click Modular Router von Eddie Kohler ([40]), der auch später in diesem Kapitel noch beschrieben wird. In einer Beispielimplementierung wurden außerdem WLAN-Schnittstellen mit Atheros-Chipsätzen in Kombination mit dem MadWifi-Treiber eingesetzt.

3.2.5 Click Modular Router

Der Click Modular Router wurde im Jahr 2000 von Eddie Kohler im Rahmen dessen Promotion zum Ph.D. vorgestellt. Es handelt sich bei diesem Software-Produkt unter offener, BSD-ähnlicher Lizenz um einen Paketverarbeitungsdienst, der unter Linux im User- oder Kernellevel ablaufen kann. Ein- und ausgehende Pakete ab OSI-Schicht drei abwärts können nahezu beliebig ausgewertet, verändert, generiert oder verworfen werden. Dadurch lässt sich die Funktionalität beliebiger Netzwerkgeräte imitieren. Auch wird der Anwender bei der schnelle Bereitstellung von Prototypen zum Testen neuer Funktionen oder Protokollmechanismen (Rapid Prototyping) unterstützt und entlastet. Dabei ist Click unabhängig von den Hardware-Spezifikationen der Netzwerkschnittstellen, auf die zugegriffen werden soll. Eine detailliertere Einführung in Click wird in Abschnitt 4.3 gegeben.

3.3 Vergleich der Frameworks

In der Tabelle 3.2 werden die zuvor einzeln vorgestellten Frameworks anhand der ebenfalls vorgestellten Kriterien einander gegenüber gestellt und bewertet. Dabei wurden die Daten - soweit möglich - den jeweils zugehörigen wissenschaftlichen Artikeln und Ausarbeitungen entnommen.

Wie der Tabelle 3.2 zu entnehmen ist, wird - mit Ausnahme des Click Modular Router - offenbar keines der Frameworks aktuell noch weiterentwickelt. Daher ist fraglich, ob eine Kompatibilität zu aktuellen Versionen des Linux-Kernels und der zugehörigen Hilfs- und Dienstprogramme noch gegeben ist. Außerdem fällt die Hardware- beziehungsweise Treiberunterstützung sehr einseitig zugunsten von MadWifi aus. Dieser wird allerdings, wie weiter oben schon beschrieben, seit 2008 nicht weiterentwickelt. Eine Unterstützung von 802.11n respektive 802.11ac ist ebenfalls nicht vorgesehen. OverlayMAC basiert zwar grundsätzlich auf dem Click Modular Router, wäre in der Folge nahezu unabhängig von den verwendeten Netzwerkschnittstellen. Da die Beispielimplementierung ebenfalls auf MadWifi aufsetzt, ist der notwendige Änderungsaufwand bei einer Umstellung auf

Kriterien	**Soft-MAC**	**Mad-MAC**	**Free-MAC**	**Overlay-MAC**	**Click Modular Router**
Treiber- & Hardwareunterstützung					
MadWifi	•	•	•	•	•
ath5k	–	–	–	–	•
ath9k	–	–	–	–	•
ath10k	–	–	–	–	•
andere	–	–	–	–	•
Implementierung					
User Mode	–	–	–	•	•
Kernel Mode	•	•	•	•	•
OSI Layer	2	2	2	btw. 2 & 3	–
Programmierung					
Sprache	C++ (?)	C++ (?)	C++ (?)	C++/ Click (?)	C++/ Click
Verfügbarkeit	–	–	–	–	•
Relevanz					
Zitationen*	139	51	69	147	2069
Letzter Commit	?	?	?	?	8/13/2014 (GitHub)
(* Stand 01.02.2015)					

Tabelle 3.2: Vergleich der Frameworks

ath9k oder ath10k nicht abschätzbar und unter Umständen ähnlich hoch wie eine vollständige Neuimplementierung direkt auf Click.

Die genannten Gründe - Aktualität und aktive Weiterentwicklung des Projekts, Hardware- und Treiberunterstützung, wissenschaftliche Relevanz - führen dazu, dass eine Umsetzung auf der Basis des Click Modular Routers vorgeschlagen und durchgeführt wird.

Kapitel 4

Testumgebung und Vorbereitungen

Dieses Kapitel dient der Beschreibung der für die Durchführung dieses Projekts eingesetzten Testumgebung. Dabei werden zuerst die einzelnen Hardware- und Software-Komponenten (Testboards, WLAN-Karten, Betriebssystem, Treiber etc.) vorgestellt. Es folgt ein Exkurs in die Themenbereiche CRDA-Datenbank und Raw Packet Injection. Das Kapitel schließt mit einer Einführung in die Funktionsweise und Eigenschaften des Click Modular Router sowie dessen Installation und Feineinstellung.

4.1 Testumgebung

4.1.1 Hardware

Da in der vorliegenden Arbeit die praktische Umsetzbarkeit eines alternativen Medienzugriffs untersucht werden soll, wird für dessen Implementierung und die Durchführung der Versuche eine Testumgebung bereitgestellt. Diese Testumgebung besteht aus zwei identischen Knoten auf der Basis des Einplatinen-Computers (engl. Single Board Computer, SBC) APU1C des Herstellers PC Engines (siehe [41]). Das APU1C ist bestückt mit einem AMD-Prozessor vom Typ T40E (Bobcat), der 64 Bit-fähig ist und auf 4 GiByte Hauptspeicher zurückgreifen kann. Als Schnittstellen für Massenspeicher stehen je ein SD-Karten-, SATA- und mSATA-Steckplatz zur Verfügung. Netzwerkseitig kann das System über drei Gigabit-LAN-Ports angebunden werden. WLAN- respektive WWAN-Verbindungen sind nachrüstbar über zwei miniPCI Express-Steckplätze. USB 2.0- und RS232-Anschlüsse sowie Anschlussmöglichkeiten für GPIO, I2C und LPC bilden den Abschluss. Für die Installation des Betriebssytems soll im Rahmen dieses Projektes eine SD-Karte verwendet werden, die jedoch nicht im Lieferumfang enthalten ist. Eine Größe von mindestens 16 Gigabyte ist empfehlenswert.

Als WLAN-Schnittstellen werden im Testbed Karten des Typs R11e-5HnD (siehe [42]) aus der Serie RouterBoard des lettischen Herstelles MikroTik eingesetzt. Diese sind ausschließlich für den Betrieb im 5GHz Band ausgelegt. Um das thermische Rauschen bei hohen Sendeleistungen und Umgebungstemperaturen zu minimieren, sind auf den Karten ausreichend dimensionierte Kühlkörper aufgebracht. Zwecks Anbindung von Antennen hat jede Karte zwei MMCX-Anschlüsse, unterstützt damit also 2x2-MIMO mit einer maximalen Bruttodatenrate von 300Mbit/s bei optimalen Bedingungen. Zunächst wurde im Mischbetrieb getestet, d.h. jeder der beiden Knoten war mit je einer 802.11n und -ac Karte ausgestattet. Um Fehlerquellen auszuschließen, wurden die 802.11ac im späteren Verlauf des Projektes noch gegen 802.11n-Karten des oben genannten Typs getauscht.

Die beiden Knoten werden jeweils durch ein Outdoor-Gehäuse "StationBox ALU" des Herstellers RF Elements gegen widrige Witterungsverhältnisse geschützt (siehe [43]). Im Lieferumfang befindet sich eine Montageplatte aus Kunststoff. Um eine verbesserte Wärmeableitung über das Gehäuse zu ermöglichen, wird diese gegen ein Äquivalent aus Aluminium ausgetauscht. Die Durchführungen auf der Unterseite des Gehäuses werden belegt durch N-Stecker für die Antennen sowie Zuführungen für Betriebsspannung und Netzwerk.

Zu Testzwecken und zum Einarbeiten in den Click Modular Router wird zudem eine virtuelle Testumgebung unter VMware Workstation mit zwei Knoten betrieben. Auf diesen ist ebenfalls Ubuntu 14.04 LTS installiert. Da die Einrichtung einer Drahtlosverbindung in VMware Workstation nicht vorgesehen ist, werden zwei dedizierte Netzsegmente eingerichtet, die als Ersatz für die drahtlosen Verbindungen fungieren sollen. Auf diese Netzsegemente haben lediglich die beiden Knoten mit jeweils einer virtuellen Ethernetschnittstelle Zugriff. VMware Workstation bietet außerdem die Möglichkeit, an das Hostsystem angeschlossene USB-Geräte an eine virtuelle Maschine durchzureichen. Wenn also eine Drahtlosverbindung zwingend benötigt würde, wäre die naheliegende Option, eine oder mehrere USB-WLAN-Adapter zu verwenden. Für weitere Informationen siehe [44].

4.1.2 Betriebssystem

Betriebssystemseitig wurde zunächst die Linux-Distribution Voyage Linux ausgewählt. Diese basiert auf Debian und wird bereits in anderen Projekten im Wireless-Umfeld erfolgreich eingesetzt. Im Hintergrund verrichtete ein Kernel mit der Versionsnummer 3.14 seinen Dienst. Wie sich allerdings im Laufe des Projekts herausgestellt hat, existieren kleinere Inkompatibilitäten zwischen der aktuellen Version des Click Modular Router und der Kernel-Version 3.14. Dies äußert sich durch einen Fehler, der beim Kompilieren des Click-Quellcodes erzeugt wird. Ein Patch, der diese Fehlermeldung behandelt, löst zwar das Kompilierungsproblem. Allerdings lässt sich beobachten, dass bei jedem Entladen der Konfiguration des Click-Kernelmoduls der gesamte Linux-Netzwerkstack derart in Mitleidenschaft gezogen wird, dass nur noch ein hartes Zurücksetzen der Knoten durch temporäres Abschalten der Stromversorgung diese wieder in einen betriebsbereiten Zustand versetzen kann. Folglich kann in dieser Konfiguration kein stabiler Betrieb gewährleistet werden.

Als Alternative zu Voyage Linux bietet sich der Einsatz von Debian Wheezy und Backports an. Die hier eingesetzten Kernelversionen 3.2 respektive 3.16 weisen allerdings ebenfalls die oben beschriebenen bekannten Probleme in Zusammenhang mit Click auf, also einen instabilen Netzwerkstack nach dem Entladen der Konfiguration.

Einen stabilen Unterbau für die Testumgebung bildet schließlich die ebenfalls auf Debian basierende Distribution Ubuntu in der Version 14.04 LTS Server. LTS steht für "Long Term Support", diese Distribution wird vom Distributor Canonical über fünf Jahre mit Aktualisierungen und Sicherheitspatches versorgt (siehe [45]). So ist ein stabiler und abgesicherter Systembetrieb über eine lange Laufzeit hinweg gegeben. In der Server-Version ist weder eine grafische Oberfläche noch viele aus dem Desktop-Umfeld bekannte Systemfunktionen und Zusatzsoftware wie der Network Manager enthalten. Dadurch bleibt das System schlank und wichtige Systemressourcen werden freigehalten. Im Lieferumfang von Ubuntu 14.04 LTS befindet sich der Linux-Kernel in Version 3.13. Diese ist vollständig kompatibel zu der eingesetzten Click-Version 2.0.1.

Für die Installation des Betriebssystems wird zunächst ein bootfähiger USB-Stick erstellt, auf den die Installationsdateien entpackt werden. Für den ersten Schritt steht unter Linux das systemeigene Werkzeug "Syslinux" zu Verfügung, unter Windows muss auf Drittanbietersoftware zurückgegriffen werden. Der Stick muss in beider Fällen vorher mit dem Dateisystem FAT16 oder FAT32 formatiert werden.

Folgende Drittanbietersoftware wurde während des Projektes erfolgreich getestet:

- UNetbootin ([46])
- Fedora LiveUSB Creator ([47])
- Universal USB Installer ([48])

Diese Liste ist nur ein Auszug und erhebt keinen Anspruch auf Vollständigkeit. Sowohl UNetbootin als auch der Fedora LiveUSB Creator stehen für Windows, Linux und Mac OS X zur Verfügung. Alle drei angeführten Werkzeuge integrieren sowohl die Vorbereitung des USB-Mediums als auch das Entpacken des Installationsabbildes in eine Benutzeroberfläche. Für Ubuntu am besten geeignet ist UNetbootin.

Linux unterstützt, wie im Übrigen Windows auch, zwei unterschiedliche Herangehensweisen bei der Installation des Betriebssystems. Im interaktiven Modus führt ein Installationsassistent durch den Einrichtungsprozess, der die wichtigsten Parameter abfragt und die Installationsroutine entsprechend konfiguriert. Hier ist allerdings eine Schnittstelle erforderlich, über die eine Interaktion mit dem Anwender erfolgen kann. Üblicherweise ist dies die Kombination aus Grafikeinheit

inklusive Bildschirm für die Darstellungsebene und Maus respektive Tastatur für die Eingaben des Benutzers. Alternativ kann dem Installationsassistenten auch eine vorher angepasste Antwortdatei übergeben werden. Die Parameter für die Installation werden dann aus dieser Antwortdatei übernommen und eine Interaktion mit dem Anwender ist nicht erforderlich. Daher wird dieser Installationsmodus auch als "unbeaufsichtigt" (engl. "unattended") bezeichnet.

Das APU1C besitzt keine Grafikeinheit, an die ein Bildschirm angeschlossen werden könnte, allerdings steht eine serielle Schnittstelle (RS232) zur Verfügung. Diese kann ebenfalls als Ein- und Ausgabe verwendet werden. Das setzt allerdings eine Anpassung des Bootmanagers und einen zweiten PC mit serieller Schnittstelle voraus. Dem Bootloader muss lediglich der serielle Port mitgeteilt werden, auf die er die Ausgabe umleiten soll. Das geschieht durch Änderungen in den Dateien isolinux.cfg, txt.cfg und syslinux.cfg. Eine ausführliche Anleitung findet sich im PC Engines-Forum unter [49] und im Anhang. Nachdem das Installationsmedium erfolgreich vorbereitet ist, kann das APU1C von diesem USB-Stick gestartet und die Installation begonnen werden. Die serielle Schnittstelle arbeitet mit einer Baudrate von 115200 bei einer Abfolge von 8 Datenbits, keinem Paritätsbit und einem Stoppbit (8N1). Besondere Einstellungen brauchen während der Installation nicht vorgenommen werden. Bei der Installation des Bootloaders GRUB muss darauf geachtet werden, dass dieser in den Bootsektor der SD-Karte geschrieben wird. Außerdem sollte bei der Auswahl der Softwarepakete die Option "SSH-Server" angewählt werden, damit später ein administrativer Zugriff über das Netzwerk möglich ist.

Nach der Installation ist es ratsam, noch einige Einstellungen innerhalb des Betriebssystem vorzunehmen. Dazu gehören:

- Hostname zur eindeutigen Identifizerung (/etc/hostname))
- Eine feste IP-Adresse (/etc/network/interfaces)
- Message of the day zur Identifizierung des Knotens (/etc/motd)
- Namensgebung der WLAN-Schnittstellen persistent und konsistent auch nach einem Neustart des Knotens (/etc/udev/rules.d/70-persistent-rules)
- SSH Public Key Authentifizierung

Auszüge aus den entsprechenden Dateien finden sich im Anhang.

Wenn das Betriebssystem vollständig angepasst ist, kann auf einem dritten System hiervon ein Abbild angefertigt werden. Dadurch kann der zweite Knoten mit exakt der gleichen Konfiguration bestückt werden. Außerdem ist eine Sicherungskopie sinnvoll, wenn ein Knoten nach einer fehlerhaften Konfiguration wiederhergestellt werden muss.

Mit dem Befehl

Listing 4.1: Sicherung der SD-Karte

```
~# dd if=/dev/sdc of=apu-6b7b.img bs=4M
```

wird ein Abbild der SD-Karte im aktuellen Verzeichnis abgelegt. Ein anschließendes

Listing 4.2: Rücksicherung auf eine leere SD-Karte

```
~# dd if=apu-6b7b.img of=/dev/sdc bs=4M
```

schreibt dieses Abbild dann auf eine leere SD-Karte. Von dieser kann dann der zweite Knoten gestartet werden. Auch hier müssen respektive sollten wieder die oben angegebenen Anpassungen (Hostname, IP-Adresse, MOTD, ...) geleistet werden, bevor der Knoten betriebsbereit ist. Außerdem empfiehlt sich, auch die öffentlichen SSH-Schlüssel der root-Benutzer der beiden Knoten wechselseitig zur Liste der vertrauenswürdigen Schlüssel hinzuzufügen. Dies erledigt der folgende Befehl:

Listing 4.3: Kopieren der öffentlichen Schlüssel

```
~# ssh-copy-id root@10.20.111.101
~# ssh 10.20.111.101
~# ssh-copy-id root@10.20.111.100 && exit
```

Während des Projektes hat sich die folgende Arbeitsumgebung innerhalb des Dateisystems bewährt (siehe Abbildung 4.1). Dabei werden die einzelnen Verzeichnisse folgendermaßen genutzt:

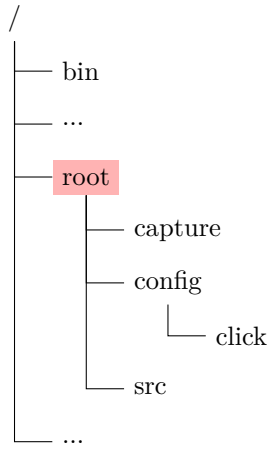


Abbildung 4.1: Arbeitsumgebung (nach eigener Darstellung)

- capture: Zielverzeichnis für Kommunikationsmitschnitte, beispielsweise mit *tshark* oder *tcpdump*.
- config: Enthält Konfigurationsdateien und -scripte. Direkt unterhalb dieses Verzeichnisses liegen automatische Konfigurationsscript für die einzelnen Szenarien respektive zu testenden Protokollvarianten (siehe Listings A.19 bis A.28 im Anhang).
- config/click: Verzeichnis für die Click-Konfigurationsdateien (siehe Listings A.16 bis A.18 im Anhang)
- src: Verzeichnis für die Tarballs und den Sourcecode der verschiedenen verwendeten Software-Pakete (Click, CRDA, Wireless RegDB, iw, ...)

nach der Einrichtung der arbeitsumgebung fehlen noch einige Software-Pakete, die aus den Paketquellen nachinstalliert werden sollten. Darunter befinden sich unter anderem git, gcc und autoconf, die für die Kompilierung von Software-Quellcode benötigt werden. Genauere Anweisungen finde sich im Anhang im Listing A.11.

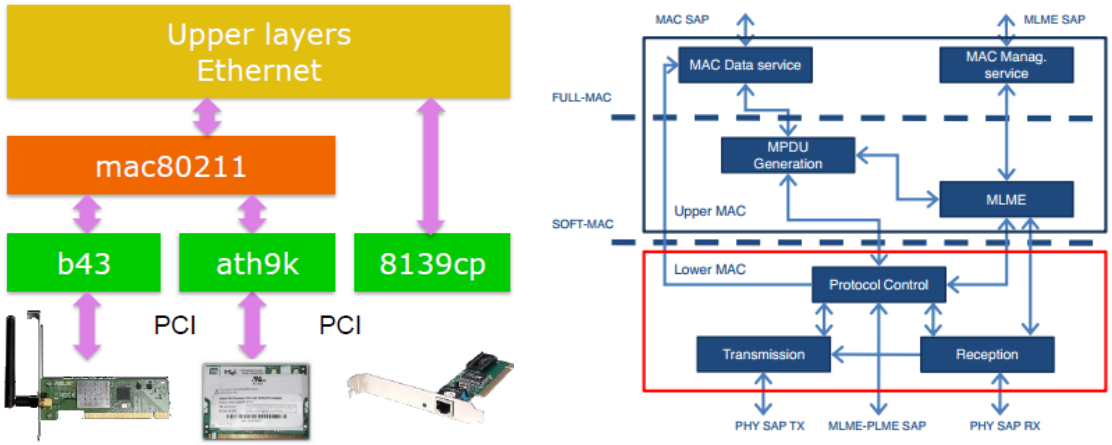
Mit Beendigung dieses Kapitels ist die Einrichtung der Testumgebung abgeschlossen. Diese ist jetzt bereit zur Installation des Click Modular Router.

4.1.3 Treiber

Wie bereits im Unterkapitel 4.1.1 beschrieben wurde, basieren die verwendeten WLAN-Karten auf Chipsätzen des Herstellers Atheros. Dieser stellt dafür die Treiber ath9k (802.11n) und ath10k (802.11ac) bereit. Diese bilden die Nachfolge von ath5k, der 2008 von Atheros erstmalig unter freier Lizenz veröffentlicht wurde ([50]), und sind ebenfalls quelloffen. ath10k erfordert allerdings zusätzlich noch eine Firmware für die WLAN-Schnittstellenkarte, die separat heruntergeladen werden muss.

Beide Treiber fügen sich in die Soft-MAC-Architektur innerhalb des Linux-Wireless-Stacks ein. Das bedeutet, dass der Großteil der OSI-Schicht 2, also die Medienzugriffskontrolle, in dem als Wrapperklasse agierenden Kernelmodul ”mac80211” stattfindet. Die gerätespezifischen Treiber klinken sich in dieses Modul ein und werden von diesem als Schnittstelle zum eigentlichen Gerät verwendet.

Auf eine vollständige Beschreibung der Struktur von ath9k und ath10k wird an dieser Stelle verzichtet. Für ath9k und dessen Vorgänger ath5k findet sich eine solche in [1]. Dort werden Sende- und Empfangspfad von Kernel zur physikalischen Schnittstelle dargestellt, herunter gebrochen auf einzelne Funktionsaufrufe.



(a) Die Schichten innerhalb des Linux Wireless Stacks

(b) Full- und Soft-MAC im Vergleich

Abbildung 4.2: Der Linux-Wireless-Stack (nach [30, 51])

4.2 Schnittstellen-Tuning

4.2.1 Anpassung der CDRA-Datenbank

Die IEEE als Standardisierungsorgan und treibende Kraft hinter vielen Standards im Telekommunikationssektor ist ebenso international tätig wie die Mitgliedsunternehmen, die sie unter ihrem Dach vereinigt. Üblicherweise sind jedoch die regulatorischen Bestimmungen für den Betrieb von Telekommunikationsausrüstung, zu denen Funkverbindungen auf der Basis von 802.11 in jedem Fall gehören, im Gegensatz dazu an nationales Recht gebunden. Diese nationalen Regulationsvorgaben sind jedoch keineswegs einheitlich, vielmehr existieren von Land zu Land durchaus unterschiedliche Vorgaben bezüglich Frequenznutzung und Sendeleistung.

Um das Verhalten von Funkkomponenten nach 802.11 beeinflussen und so an den jeweiligen Betriebsort anpassen zu können, existiert eine Datenbank mit den regulatorischen Vorgaben aller beteiligten Nationen. Unter Linux ist für die Zuweisung und Einhaltung dieser Vorgaben der so genannte ”Central Regulatory Domain Agent” (CRDA) vorgesehen, der als Userland-Applikation über die Schnittstelle NL80211 mit den Treibern der WLAN-Karten kommuniziert und auf die gerade beschriebene Datenbank ”Wireless RegDB” zurückgreift ([52]).

Zu Forschungszwecken ist es allerdings bisweilen nötig, die nationalen Regulationsvorschriften zeitlich und örtlich begrenzt zu umgehen, um beispielsweise zu simulieren, wie sich ein System unter bestimmten Bedingungen verhält. Dazu kann die CRDA-Datenbank an die eigenen Vorhaben angepasst werden (wie in [53] zu sehen).

Softwareseitig werden einige Kryptografie-Pakete und eine aktuelle Version der User-/Kernelspace-Schnittstelle NL80211 vorausgesetzt:

- python-m2crypto
- libgcrypt11
- libgcrypt11-dev
- libnl-dev
- openssl
- pkg-config

Anschließend werden die aktuellen Quellcode-Tarballs des CRDA und der Wireless RegDB von den Webservern der Linux Foundation heruntergeladen und entpackt. Im entpackten Ordner der Wireless RegDB findet sich anschließend eine Datei ”db.txt”. Diese enthält die Regulationsbestimmungen aller beteiligten Länder in folgender Form:

Listing 4.4: Eintrag in der Wireless RegDB (Bsp.)

```
country XY:
(Frequenzband @ Kanalbreite), (Signalstaerke), weitere Optionen
```

Beispiel Deutschland:

Listing 4.5: Eintrag ”Deutschland” in der Wireless RegDB

```
country DE: DFS-ETSI
# entries 279004 and 280006
(2400 - 2483.5 @ 40), (100 mW)
# entry 303005
(5150 - 5250 @ 80), (100 mW), NO-OUTDOOR, AUTO-BW
# entries 304002 and 305002
(5250 - 5350 @ 80), (100 mW), NO-OUTDOOR, DFS, AUTO-BW
# entries 308002, 309001 and 310003
(5470 - 5725 @ 160), (500 mW), DFS
# 60 gHz band channels 1-4, ref: Etsi En 302 567
(57000 - 66000 @ 2160), (40)
```

Zusätzlich existiert noch eine für alle Länder geltende globale Limitierung (”world”), die der Datenbank als Eintrag ”00” vorangestellt ist. Diese und die der jeweils aktiven Länderkennung zugeordneten Bestimmungen werden verglichen und die restriktivsten Vorgaben dann angewendet.

Für eine Freischaltung aller Kanäle sowohl im 2,4GHz- als auch im 5GHz-Band muss zunächst die globale Vorgabe auf folgendes abgeändert werden:

Listing 4.6: Neue Regulationsdomäne in der Wireless RegDB

```
country 00:
(2402 - 2482 @ 40), (30)
(5170 - 5835 @ 40), (30)
```

Anschließend wird eine neue Regulationsdomäne mit einem beliebigen freien Länderkürzel (beispielsweise WB) an das Ende der Datenbank angefügt. Die Regulationsvorgaben sind die gleichen wie im vorangehenden Listing. Nachdem alle Änderungen eingespielt sind, wird die Datenbank nun in das vom CDRA benötigte Binärformat konvertiert und installiert. Dies erledigt das unter Linux übliche Werkzeug *make* mit anschließendem *make install*. Eine passende Konfigurationsdatei befindet sich bereits im Umfang des Tarballs.

Bei der Erstellung der binären Datenbank wird diese mit ebenfalls in diesem Schritt erstellten selbst-signierten Zertifikaten digital signiert. Damit der CRDA im nächsten Schritt diese selbstsignierte Datenbank akzeptiert, müssen zunächst die Zertifikate in das Quellcodeverzeichnis des CRDA kopiert werden. Danach wird CRDA wie im vorigen Schritt die Wireless RegDB kompiliert und installiert. Zum Abschluss ist noch ein Neustart erforderlich, damit die nötigen Dateien sauber entladen und neu geladen werden können.

Mittels des Werkzeugs *iw* lässt sich nun die korrekte Funktion des CRDA testen:

Listing 4.7: Setzen der neuen Regulationsdomäne

```
# iw reg set WB && iw reg get
```

Als Ausgabe sollten die zuvor in der Datenbank gespeicherten selbst definierten Vorgaben auf dem Bildschirm erscheinen. Damit ist die Anpassung der CRDA-Datenbank abgeschlossen.

Wie jedoch eine Ausgabe von ”iw phy0 info” zeigt, scheint noch eine zusätzliche Sicherungsmaßnahme in den Linux-Wireless-Stack eingebaut zu sein (siehe Listing 4.8).

Listing 4.8: Ausgabe der aktiven Regulationsbestimmungen (iw phy0 info)

```
...
HT TX/RX MCS rate indexes supported: 0-15
Frequencies:
* 5180 MHz [36] (17.0 dBm)
* 5200 MHz [40] (17.0 dBm)
* 5220 MHz [44] (17.0 dBm)
* 5240 MHz [48] (17.0 dBm)
* 5260 MHz [52] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5280 MHz [56] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5300 MHz [60] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5320 MHz [64] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5500 MHz [100] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5520 MHz [104] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5540 MHz [108] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5560 MHz [112] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5580 MHz [116] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5600 MHz [120] (disabled)
```

```
* 5620 MHz [124] (disabled)
* 5640 MHz [128] (disabled)
* 5660 MHz [132] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5680 MHz [136] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5700 MHz [140] (20.0 dBm) (passive scanning, no IBSS, radar detection)
* 5745 MHz [149] (30.0 dBm)
* 5765 MHz [153] (30.0 dBm)
* 5785 MHz [157] (30.0 dBm)
* 5805 MHz [161] (30.0 dBm)
* 5825 MHz [165] (30.0 dBm)
...
```

Um diese Sicherungsmaßnahme umgehen zu können, werden zunächst die Quelldateien für den Linux-Kernel benötigt. Diese können unter Ubuntu folgendermaßen installiert werden:

Listing 4.9: Installation der Linux-Source-Dateien

```
~# apt-get update
~# apt-get install linux-source-3.13.0
```

Nach der Installation befindet sich die Quelldateien unterhalb von /usr/src/linux-source-3.13.0. Die relevanten Dateien sind net/wireless/reg.c und drivers/net/wireless/ath/regd.c (siehe [54]). Die erste der beiden ist Bestandteil des Kernelmoduls "cfg80211" aus der Linux-Soft-MAC-Architektur. In ihr wird eine neue Datenstruktur namens *ieee80211_reg_rule* mit zwei Substrukturen (*ieee80211_freq_range* und *ieee80211_power_rule*) definiert:

Listing 4.10: Auszug aus net/wireless/reg.c

```
...
struct ieee80211_freq_range {
        u32 start_freq_khz;
        u32 end_freq_khz;
        u32 max_bandwidth_khz;
};

struct ieee80211_power_rule {
        u32 max_antenna_gain;
        u32 max_eirp;
};

struct ieee80211_reg_rule {
        struct ieee80211_freq_range freq_range;
        struct ieee80211_power_rule power_rule;
        u32 flags;
        u32 dfs_cac_ms;
};
...
```

Die so geschaffene Struktur für Regulations-Regeln kann dann mit Inhalt gefüllt werden. Hier ein Beispiel, ebenfalls aus der Datei net/wireless/reg.c:

Listing 4.11: Auszug aus der Datei net/wireless/reg.c (Forts.)

```
...
/* IEEE 802.11a, channel 36..48 */
REG_RULE(5180-10, 5240+10, 160, 6, 20,
                NL80211_RRF_PASSIVE_SCAN |
                NL80211_RRF_NO_IBSS),

/* IEEE 802.11a, channel 52..64 - DFS required */
REG_RULE(5260-10, 5320+10, 160, 6, 20,
                NL80211_RRF_PASSIVE_SCAN |
                NL80211_RRF_NO_IBSS |
                NL80211_RRF_DFS),
...
```

Wie in den Listings 4.10 und 4.11 zu sehen ist, besteht eine solche Regel aus der Angabe eines Frequenzbandes samt Kanalbreite, Antennengewinn und maximaler Sendeleistung sowie den so genannten Flags. Diese Flags geben die Einschränkungen an, die innerhalb des in der Regel angegebenen Frequenzbandes gelten, also beispielsweise keine Adhoc-Netzwerke (NO_IBSS) oder passive Radar-Scans (PASSIVE_SCAN).

Eigentlich dienen diese hart implementierten Regeln nur als Rückfallsicherung, falls auf dem betroffenen System der CRDA-Dienst nicht installiert ist. Scheinbar wirken die Regeln aber auch

bei vorhandenem CRDA-Daemon, so dass eine freie Kanalwahl zu Testzwecken nur eingeschränkt möglich ist. Um hier Abhilfe zu schaffen, werden die Einträge wie folgt angepasst (siehe hierzu auch Listing 4.12): Die Zeilen, die für jede Regel die Flags enthalten, werden auskommentiert. Die Regeln selbst erhalten dann den Wert "0" als Flag übergeben.

Listing 4.12: Auszug aus der Datei net/wireless/reg.c nach Änderung (Forts.)

```
...
/* IEEE 802.11a, channel 36..48 */
REG_RULE(5180-10, 5240+10, 160, 6, 20, 0),
        //NL80211_RRF_PASSIVE_SCAN |
        //NL80211_RRF_NO_IBSS),

/* IEEE 802.11a, channel 52..64 - DFS required */
REG_RULE(5260-10, 5320+10, 160, 6, 20, 0),
        //NL80211_RRF_PASSIVE_SCAN |
        //NL80211_RRF_NO_IBSS |
        //NL80211_RRF_DFS),
...
```

Die Treiberfamilie *ath* besitzt ebenfalls eine solche Rückfallsicherung. Diese befindet sich in der Datei "drivers/net/wireless/ath/regd.c". Auch hier werden wieder die Flags auskommentiert und auf Null gesetzt (siehe Listing 4.13)

Listing 4.13: Auszug aus der Datei drivers/net/wireless/ath/regd.c

```
* We allow IBSS on these on a case by case basis by regulatory domain */
#define ATH9K_5GHZ_5150_5350    REG_RULE(5150-10, 5350+10, 80, 0, 30, 0)
//                              NL80211_RRF_PASSIVE_SCAN | NL80211_RRF_NO_IBSS)
#define ATH9K_5GHZ_5470_5850    REG_RULE(5470-10, 5850+10, 80, 0, 30, 0)
//                              NL80211_RRF_PASSIVE_SCAN | NL80211_RRF_NO_IBSS)
#define ATH9K_5GHZ_5725_5850    REG_RULE(5725-10, 5850+10, 80, 0, 30, 0)
//                              NL80211_RRF_PASSIVE_SCAN | NL80211_RRF_NO_IBSS)
```

Anschließend können die entsprechenden Kernelmodule neu kompiliert und nach Entladen der alten Module ausgetauscht werden, wie in Listing 4.14 gezeigt wird. Eine Anleitung zum Kompilieren von Linux-Kernelmodulen des Benutzers "Nixcraft" ist unter cybercity.com [55] zu finden.

Listing 4.14: Kompilieren der Kernelmodule

```
~# rmmod ath9k ath9k_common ath9k_hw ath mac80211 cfg80211
~# cd /lib/modules/3.13.0-44-generic/kernel/net/wireless/
~# mv cp cfg80211.ko cfg80211.ko.bkp && cp lib80211_crypt_ccmp.ko
    lib80211_crypt_ccmp.ko.old && cp lib80211_crypt_tkip.ko lib80211_crypt_tkip.ko.
    old && cp lib80211_crypt_wep.ko lib80211_crypt_wep.ko.old && cp lib80211.ko
    lib80211.ko.old
~# cd /usr/src/linux-source-3.13.0/net/wireless/
~# make -C /lib/modules/`uname -r`/build M=`pwd` modules
~# cp *.ko /lib/modules/3.13.0-44-generic/kernel/net/wireless/
~# cd /lib/modules/3.13.0-44-generic/kernel/drivers/net/wireless/ath/
~# cp ath.ko ath.ko.old
~# cd /usr/src/linux-source-3.13.0/drivers/net/wireless/ath
~# make -C /lib/modules/`uname -r`/build M=`pwd` modules
~# cp ath.ko /lib/modules/3.13.0-44-generic/kernel/drivers/net/wireless/ath/
~# reboof -f
```

Auch diese Maßnahmen zeigten allerdings bislang keine Wirkung, wie sich in abschließenden Versuchen gezeigt hat. Es scheint folglich noch eine dritte Sicherungsmaßnahme vorgesehen zu sein, die bis jetzt im Rahmen des Projektes noch nicht gefunden wurde.

4.2.2 Raw Packet Injection

Ziel dieses Projektes ist, wie bereits beschrieben, die Implementierung einer alternativen Medienzugriffskontrolle für 802.11. Im Zuge dieser Implementierung soll daher die bestehende Medienzugriffskontrolle so weit wie möglich abgeschaltet oder umgangen werden, damit der tatsächliche Zugriff auf das Medium möglichst frei und uneingeschränkt erfolgen kann.

Eine solche Möglichkeit, die MAC-Schicht zu umgehen, kann durch die so genannte Raw Packet Injection realisiert werden. Dabei werden Datenpakete beliebigen Formats (also Rohdaten) an der MAC-Schicht vorbei in den WLAN-Treiber eingeschleust. Dafür muss sich die WLAN-Schnittstelle

im Monitor-Modus befinden. Dieser wird üblicherweise dafür genutzt, rohe Datenpakete zu Debugzwecken aus dem Datenstrom abzugreifen, ähnlich dem Promiscous Mode einer Ethernet-Schnittstelle. Im Monitor-Modus werden auch Datenpakete angezeigt, die normalerweise durch den Treiber ausgefiltert werden, beispielsweise Verwaltungsinformationen, Beacons und ähnliche.

Mit Raw Packet Injection ist es nun möglich, auch beliebige Datenpakete zu senden. Dafür müssen diese Datenpakete mit einem speziellen Header versehen werden, der entsprechende Informationen für den eigentlichen Sendevorgang enthält. Auf diese Weise können beispielsweise die Datenrate, der Sendekanal, die Sendeleistung, RTS/CTS und andere vorgegeben werden. Die Datenpakete können allerdings nur versendet werden, wenn im Radiotap-Header alle benötigten Felder korrekt ausgefüllt sind.

Die folgenden Kapselungselemente für die unterschiedlichen Header-Typen sind bereits in Click implementiert (siehe [56]):

- AthdescEncap/AthdescDecap
- ExtraEncap/ExtraDecap
- Prism2Encap/Prism2Decap
- RadiotapEncap/RadiotapDecap
- WepEncap/WepDecap
- WifiEncap/WifiDecap

Außerdem können den Paketen die entsprechenden Sendeinformationen wie Datenrate oder Sendeleistung mithilfe der folgenden Elemente mitgegeben werden:

- MadwifiRate: Madwifi wireless bit-rate selection algorithm
- SetRTS: Enable/disable RTS/CTS for a packet.
- SetTXPower: Sets the transmit power for a packet.
- SetTXRate: Sets the bit-rate for a packet.
- SetWifiExtraFlag: Sets the Wifi flags on a packet.

Bis zum jetzigen Zeitpunkt war keine Beispielimplementierung zu finden, anhand der das passende Element für eine Raw Packet Injection auf dem verwendeten Atheros-Treiber ath9k zu erkennen gewesen wäre. Versuche mit "RadiotapEncap/RadiotapDecap" und "ExtraEncap/ExtraDecap" in Kombination mit "SetTXPower" und "SetTXRate" haben keinen Erfolg gebracht. Für MadWifi sind entsprechende Beispiele zu finden. Allerdings funktionieren diese nicht mit ath9k, wie sich gezeigt hat. Wie in [31] zu finden ist, erlaubt ath9k allerdings scheinbar keine Raw Packet Injection im 5GHz-Band. Daher ist nicht auszuschließen, dass eine solche Konfiguration mit Click dennoch funktionsfähig gewesen wäre.

4.3 Der Click Modular Router: Einführung

4.3.1 Einführung in Click

Wie bereits in Kapitel 3.2 beschrieben, entstand der Click Modular Router im Jahr 2000 als Ph.D.-Thesis des Amerikaners Eddie Kohler. Seitdem wird er konsequent gepflegt und weiterentwickelt, um zu aktuellen Linux-Distributionen kompatibel zu bleiben.

Mit Click können nahezu beliebige Datenpakete generiert, verworfen, verändert, weitergeleitet, dupliziert, gefiltert oder anderweitig ausgewertet werden. Im folgenden Kapitel wird die Funktionsweise von Click umrissen und die zugehörigen Fachtermini erläutert und eingeordnet. Dabei sind die Informationen, soweit nicht anders gekennzeichnet, allesamt der offiziellen Click-Benutzerdokumentation ([40]) entnommen.

4.3.2 Elemente und Konfigurationen

Das Verhalten des Click Modular Router wird über eine zur Laufzeit veränderbare Konfiguration vorgegeben. Diese, bestehend aus einer Verkettung von so genannten Elementen, wird in einer C++-ähnlichen Scriptsprache verfasst und kann in Dateiform oder als Zeichenkette übergeben werden.

Element

Ein Element ist eine atomare Funktionseinheit innerhalb des Click Modular Router. Sie erfüllt einen spezifischen Zweck, d.h. die Veränderung respektive Verarbeitung von Datenpaketen auf eine bestimmte Art und Weise. Typische Click-Elementgattungen sind beispielsweise Warteschlangen (Queues), Priorisierungen, Zufallselemente, Klassifizierungsmechanismen (Classifier) oder Schnittstellen (Devices).

Jedes Element verfügt über jeweils einen oder mehrere Ein- und Ausgänge, über die Datenpakete dem Element zugeführt beziehungsweise entnommen werden können. Über diese Ein- und Ausgänge werden die Verknüpfungen zu anderen Elementen hergestellt. Dabei wird zwischen so genannten Push- und Pull-Schnittstellen unterschieden. Elemente mit Pull-Ausgang erwarten vom Folgeelement, dass sich dieses eigenständig Pakete abholt (pull), wenn es dazu bereit ist. Ein Beispiel dafür sind physikalische Netzwerkschnittstellen (ToDevice) oder Scheduling-Elemente. Ein Element mit Push-Ausgang entscheidet dagegen selbstständig, wann es dem Folgeelement ein Datenpaket übergibt (push). Elemente mit so genannten "agnostischen" Schnittstellen können entweder als Push- oder als Pullschnittstelle initialisiert werden, abhängig davon, welcher Schnittstellentyp am Vorgänger- und Folgeelement vorhanden ist.

Wenn zwei Elemente verkettet werden, muss bezüglich der zu verbindenden Schnittstellen Homogenität herrschen. Das bedeutet, es darf immer nur ein Push-Ausgang mit einem Push-Eingang beziehungsweise Pull-Ausgang mit Pull-Eingang verbunden werden. Eine Verbindung zwischen Pull- und Push-Schnittstellen ist nicht gestattet. Eine Ausnahme bilden Konverter-Elemente mit unterschiedlich konfigurierten Ein- und Ausgängen. Diese können zwischen Pull- und Pushschnittstellen vermitteln. Das Warteschlangenelement "Queue" beispielsweise verfügt über einen Push-Eingang und einen Pull-Ausgang, kann also Datenpakete entgegennehmen und speichern, bis das Folgeelement sie abholt. "Unqueue" hingegen arbeitet umgekehrt. Es entnimmt einem Pull-Ausgang ein Datenpaket und reicht es direkt an das Folgeelement mit Push-Eingang weiter.

Konfiguration

Eine gültige Click-Konfiguration ist eine funktionsfähige Verkettung von Elementen zu einem bestimmten Zweck. Die Sprache der Konfiguration ist, wie bereits beschrieben, C++-ähnlich. Elemente können implizit oder explizit deklariert werden. Der Paketfluss von Element zu Element wird anschließend mit Pfeilen ("->") symbolisiert. Ein Paketfluss innerhalb der Konfiguration führt dabei immer von einer oder mehreren Quelle zu einer oder mehreren Senken. Auch die Ein- und Ausgänge können explizit oder implizit angesprochen werden.

Das folgende Listing zeigt eine gültige Minimalkonfiguration mit expliziter Deklaration und teilweise expliziter Schnittstellenzuweisung.

```
tsrc :: TimedSource(0.2);
dev :: ToDevice (eth0);
tsrc[0] -> dev;
```

Dabei werden zunächst zwei Elemente deklariert und initialisiert. Das erste, "tsrc" ist eine zeitgesteuerte Quelle, die im Abstand von 0,2 Sekunden Pakete generiert und an das nächste Element weiterreicht. Das zweite Element "dev" wird mit einer Netzwerkschnittstelle des Betriebssystems (eth0) initialisiert. Es schaltet sich in den Protokollstack dieser Netzwerkschnittstelle ein und kann über diese dann exklusiv verfügen. In der dritten Zeile wird dann die Verknüpfung zwischen den zuvor initialisieren Elementen hergestellt. Dabei wird beim Element "tsrc" explizit der erste Ausgang "[0]" gewählt. Findet keine explizite Aus- oder Eingangswahl statt, dann versucht Click beim Laden der Konfiguration eine sinnvolle Belegung der Ein- und Ausgänge vorzunehmen.

4.3.3 User- und Kernelmodus

Der Click Modular Router bietet zwei Möglichkeiten, die im oben stehenden Abschnitt beschriebenen Konfigurationen zu laden. Zu Test- und Debugzwecken empfiehlt sich das Usermodus-

Programm ”click”. Fehler in der Konfiguration lassen sich durch die direkte Interaktion mit dem Benutzer leichter entdecken.

Außerdem steht der Click-Daemon im Kernelmodus zur Verfügung. Im Zusammenspiel mit dem Betriebssystemkern, insbesondere im Bezug auf Treiber und andere hardwarenahe Operationen, entfällt so der Kontextwechseln von User- nach Kernelmodus. Eine Konfiguration lässt sich mit ”click-install $FILENAME$” und ”click-uninstall” laden respektive entladen. Der Click-Daemon integriert sich zur Interaktion mit dem Benutzer in das Proc-Dateisystem (unterhalb von ”/proc/click” beziehungsweise auch direkt unterhalb von ”/click”). Dort kann der Zustand und die Eigenschaften der einzelnen Elemente über so genannte Handler abgefragt und verändert werden. Außerdem lässt sich auch die gesamte Konfiguration ausgeben. Der Zugriff erfolgt entsprechend des Unix/Linux-Codex ”Everything is a file”, beispielsweise mittels der Kommandozeilen-Werkzeuge ”cat” zum Auslesen oder ”echo” zum Verändern von Konfigurationsparametern.

Sofern möglich, kann auch versucht werden, eine funktionsfähige Konfiguration im laufenden Betrieb gegen eine andere auszutauschen. Dieser ”Hot Swapping” genannte Vorgang versucht, die neue Konfiguration zu initialisieren. Schlägt dies fehl, bleibt die alte Konfiguration in Kraft. Bei Erfolg hingegen übernimmt die neue Konfiguration auch alle Betriebszustände der alten. So ist in beiden Fällen eine unterbrechungsfreie Paketverarbeitung gewährleistet.

An dieser Stelle sei darauf hingewiesen, dass nicht alle Click-Elemente sowohl für User- als auch Kernelmodus zur Verfügung stehen.

4.4 Der Click Modular Router: Inbetriebnahme

Der vorige Abschnitt stellte, wie bereits erwähnt, lediglich eine Übersetzung und Zusammenfassung der Benutzerdokumentation von Click dar, um dem Leser einen Einblick in dessen Struktur und Funktionalität zu gewähren. Im Gegensatz dazu bezieht sich dieses Unterkapitel auf Erfahrungen und Arbeitsleistungen während der Projektphase, stützt sich also auf praktische Erkenntnisse des Autors bei der Inbetriebnahme des Click Modular Router auf der zu Beginn dieses Kapitels beschriebenen Testumgebung.

4.4.1 Installation

Die Installation des Click Modular Router ist weitestgehend unkompliziert. Da die Software nicht in den üblichen Paketquellen vorhanden, sondern nur als Quellcode verfügbar ist, muss dieser zunächst heruntergeladen werden. Das geschieht entweder als Tarball oder per ”git clone” vom Github-Repository. Nach dem Entpacken wird mit dem configure-Script ein Makefile erstellt, auf dessen Basis die Software kompiliert werden kann. Der genaue Ablauf ist im Listing 4.15 aufgeführt. Die entsprechenden Abhängigkeiten wie Compiler und *git* selber wurden bereits während der Einrichtung der Testumgebung aufgelöst.

Listing 4.15: Check-Out und Installation des Click Modular Router

```
~# cd ~/src && git clone git://github.com/kohler/click
~# cd click
~# ./configure --enable-linuxmodule --enable-all-elements --enable-local
~# make && make install
```

Bei der Erstellung des Makefiles durch ”./configure” werden drei Optionen mitgegeben. ”–enable-linuxmodule” bewirkt, dass das Click-Kernelmodul erstellt wird. Die Protokolle, die in diesem Projekt erarbeitet werden, sollen aus Performancegründen im Kernelspace ablaufen. Daher wird auf die Kompilierung der Userspace-Funktionen verzichtet. Mit ”–enable-all-elements” werden alle in Click enthaltenen Elemente beim Kompilierungsvorgang berücksichtigt, ”–enable-local” schließt darüber hinaus explizit das Verzeichnis mit ein, in dem Eigenentwicklungen abgelegt werden sollten (elements/local). Wenn beim Kompilieren und installieren keine Fehler aufgetreten sind, sollte nun auf der Kommandozeile die Funktion ”click-install” zur Verfügung stehen. Diese erlaubt das Laden von Konfigurationsdateien in das Click-Kernelmodul.

4.4.2 Click und ath9k

Wie bei ersten Performance-Tests auffiel, klappt das Zusammenspiel zwischen dem Click Modular Router und dem Treiber für die Atheros-WLAN-Schnittstellen, namentlich ath9k, nicht unbedingt

reibungslos. Vielmehr zeigte sich bei Durchsatzmessungen mit dem Werkzeug ”iperf” eine deutlich schlechtere Leistungsfähigkeit als ohne aktive Click-Konfiguration. Ein Indiz war zunächst eine hohe CPU-Last, die durch den Prozess ”syslogd” hervorgerufen wird. Außerdem lag die durchschnittliche Round Trip Time von Datenpaketen auf der drahtlosen Verbindung um ein Vielfaches höher und der Durchsatz um ein Vielfaches niedriger.

Ein Blick in die Datei /var/log/syslog verriet den Grund für diese Hohe Auslastung des Prozessors. Zu erkennen ist bei jedem ausgehenden und ankommenden Paket die Auslösung eines so genannten Call Traces, also eine Auflistung, welche Funktionen in welcher Reihenfolge aufgerufen wurden und an welcher Stelle es zu einem Fehler kam. Wie sich im Listing zeigt, ist der Verlauf der Funktionsaufrufe in etwa wie folgt: Click → mac80211 → ath9k.

Listing 4.16: Call-Trace-Auszug aus dem Syslog

```
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631704] ------------[ cut here
    ]------------
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631751] WARNING: CPU: 0 PID: 1225 at /build
    /buildd/linux-3.13.0/drivers/net/wireless/ath/ath9k/xmit.c:2218 ath_tx_start+0
    x304/0x310 [ath9k]()
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631756] Modules linked in: click(OX)
    proclikefs(OX) arc4 ath9k ath9k_common ath9k_hw ath10k_pci kvm_amd ath10k_core
    ath kvm ath3k mac80211 btusb bluetooth k10temp sp5100_tco cfg80211 i2c_piix4
    mac_hid lp parport usb_storage ahci libahci r8169 mii
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631813] CPU: 0 PID: 1225 Comm: kclick
    Tainted: G           OX 3.13.0-44-generic #73-Ubuntu
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631818] Hardware name: PC Engines APU/APU,
    BIOS 4.0 09/08/2014
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631823]  0000000000000009 ffff8800d005fa98
    ffffffff81720d86 0000000000000000
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631835]  ffff8800d005fad0 ffffffff810677cd
    ffff8800d051a230 ffff8800d05197c0
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631843]  0000000000000000 ffff8800d212e700
    ffff8800d005fb60 ffff8800d005fae0
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631852] Call Trace:
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631869]  [<ffffffff81720d86>] dump_stack+0
    x45/0x56
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631881]  [<ffffffff810677cd>]
    warn_slowpath_common+0x7d/0xa0
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631890]  [<ffffffff810678aa>]
    warn_slowpath_null+0x1a/0x20
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631905]  [<ffffffffa0379984>] ath_tx_start
    +0x304/0x310 [ath9k]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.631919]  [<ffffffffa0371973>] ath9k_tx+0xa3
    /0x1c0 [ath9k]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.632021]  [<ffffffffa012dea9>]
    __ieee80211_tx+0x249/0x350 [mac80211]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.632065]  [<ffffffffa01303cf>] ieee80211_tx
    +0xbf/0xf0 [mac80211]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.632110]  [<ffffffffa01305ed>]
    ieee80211_xmit+0x9d/0x100 [mac80211]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.632153]  [<ffffffffa0130ffa>]
    ieee80211_subif_start_xmit+0x68a/0xd70 [mac80211]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.632168]  [<ffffffff81010000>] ?
    xen_smp_intr_free+0xc0/0x1d0
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.632177]  [<ffffffff8109d4f5>] ?
    sched_clock_cpu+0xb5/0x100
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.633168]  [<ffffffffa0434f9b>]
    _ZN8ToDevice12queue_packetEP6PacketP12netdev_queue+0x11b/0x240 [click]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.634138]  [<ffffffffa0442ee7>] ?
    _ZN13FullNoteQueue4pushEiP6Packet+0xe7/0x300 [click]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.635092]  [<ffffffffa0435162>]
    _ZN8ToDevice8run_taskEP4Task+0xa2/0x390 [click]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.636096]  [<ffffffffa03c38f5>]
    _ZN12RouterThread6driverEv+0x385/0x570 [click]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.637115]  [<ffffffffa0472eec>]
    _ZL11click_schedPv+0x12c/0x280 [click]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.638081]  [<ffffffffa0472dc0>] ?
    _Z19click_cleanup_schedv+0x180/0x180 [click]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.639031]  [<ffffffffa0472dc0>] ?
    _Z19click_cleanup_schedv+0x180/0x180 [click]
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.639068]  [<ffffffff8108b572>] kthread+0xd2
    /0xf0
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.639091]  [<ffffffff8108b4a0>] ?
```

```
    kthread_create_on_node+0x1c0/0x1c0
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.639101]  [<ffffffff817317bc>] ret_from_fork
    +0x7c/0xb0
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.639109]  [<ffffffff8108b4a0>] ?
    kthread_create_on_node+0x1c0/0x1c0
Feb  3 11:09:29 apu-6b7b kernel: [ 4283.639117] ---[ end trace 4f6209ef51490c8b
    ]---
```

Dieses Verhalten trat zum ersten mal auf, als eine FDMA-Konfiguration in Click geladen war. Daher lag die Vermutung nahe, dass es an vertauschten MAC-Adressen läge (mehr hierzu in Abschnitt 5.1.2). Um dies entweder verifizieren oder ausschließen zu können, wurden daher zwei Strategien verfolgt. Zum ersten wurde versucht, mittels ARP-Spoofing auf beiden WLAN-Karten eines Knotens die gleiche MAC-Adresse zu konfigurieren. Dies ist unter Linux möglich mit folgendem Befehl (Listing 4.17):

Listing 4.17: ARP-Spoofing unter Linux

```
~# ip link dev device set address aa:bb:cc:dd:ee:ff
```

Außerdem wurde eine Click-Konfiguration getestet, die keine Änderungen am Verhalten des darüber und darunterliegenden Protokollstapel hervorrufen sollte. Diese Passthrough-Konfiguration findet sich im Anhang im Listing A.16. Bei beiden Verfahren wurden mit *tshark* Mitschnitte der Pakete auf den zugehörigen Monitorschnittstellen erstellt. Anhand dieser Mitschnitte wurden die versendeten Pakete auf Gleichheit überprüft. Bei diesem Vergleich war kein signifikanter Unterschied feststellbar. Die Leistungsfähigkeit war jedoch in allen Fällen (außer wenn keine Konfiguration geladen war) ähnlich eingeschränkt.

Weitere Recherchen im Internet ergaben als Fehlerquelle eine fehlerhafte Sendewarteschlangenzuweisung im Modul "mac80211". Durch die Art, wie sich der Click Modular Router in den Linux-Netzwerk-Stack einklinkt, wird allen durch Click behandelten Paketen durch den Soft-MAC-Layer keine Sendewarteschlange zugewiesen. Dies führt bei der finalen Übergabe an die physikalische WLAN-Karte zu Verwirrung im Treiber und infolgedessen zu Fehlermeldungen im Systemlog. Die Abbildung 4.3 versucht diesen Vorgang noch einmal grafisch darzustellen.

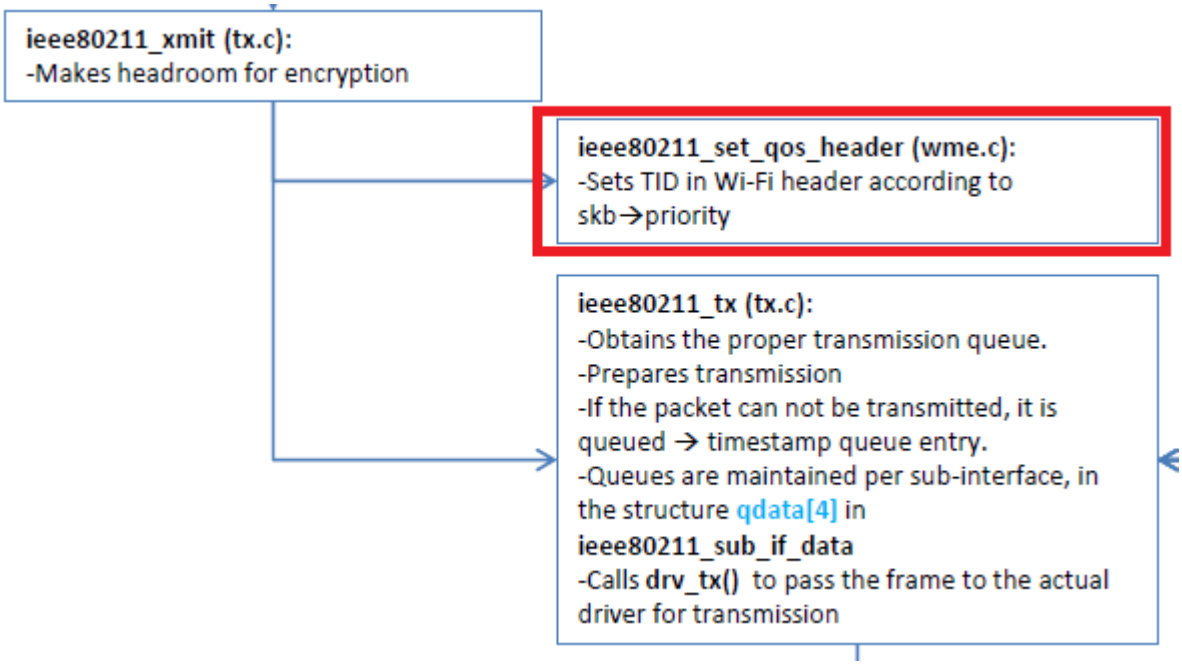


Abbildung 4.3: Setzen der Sendewarteschlange in ath9k (nach [1])

Ein Patch, der von Ben Greear in [57] vorgestellt wird, beschreibt ein ähnliches Problem und schlägt eine Änderung in der Datei net/mac80211/tx.c vor. Diese Änderung legt das TX Queue Mapping an eine andere Stelle innerhalb des Moduls und ermöglicht so auch dem Click Modular Router, ein korrektes Mapping durchzuführen.

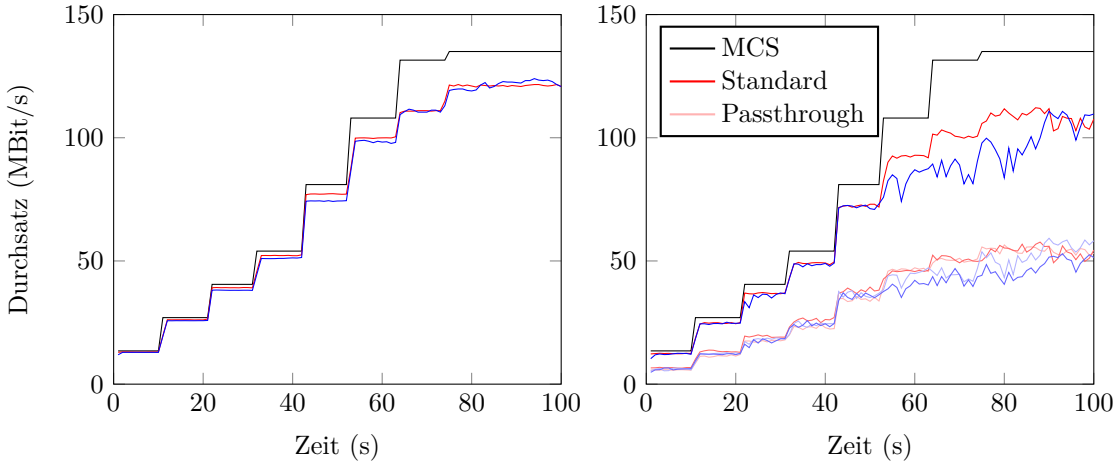
Listing 4.18: Auszug aus der Datei net/mac80211/tx.c

```
...
        memcpy(skb_push(skb, hdrlen), &hdr, hdrlen);

// Patch!!!!
skb_set_queue_mapping(skb, ieee80211_select_queue(sdata, skb));
// Patch Ende!!!!!

nh_pos += hdrlen;
h_pos += hdrlen;
...
```

Nach dem Einspielen dieses Patches folgt das erneute Kompilieren und Austauschen des Kernelmoduls wie in Abschnitt 4.2.1 beschrieben. Wie sich nach dem obligatorischen Neustart offenbart, zeigt der Patch Wirkung. Die Leistungsfähigkeit ist nun durchaus vergleichbar mit der ohne aktive Click-Konfiguration. Siehe hierzu auch Abbildung 4.4.



(a) Unidirektionaler Datenfluss

(b) Bidirektionaler Datenfluss

Abbildung 4.4: Vergleich Durchsatz ohne und mit Click-Konfiguration (Passthrough) (nach eigener Darstellung)

Der rote Kurvenverlauf in Abbildung 4.4a (Unidirektional) stellt den Durchsatz in Megabit pro Sekunde (MBit/s) ohne aktive Click-Konfiguration dar, der blaue mit Passthrough-Konfiguration. In Abbildung 4.4b (Bidirektional) sind mit den gleichen Farben die kumulierten Werte aus beiden Verkehrsrichtungen zu sehen. Zusätzlich sind hier in hellblau beziehungsweise hellrot noch die einzelnen Datenströme je Richtung aufgetragen. Als Referenz ist in beiden Teilabbildungen in schwarz die maximale Nominaldatenrate nach Spezifikation aufgetragen. Wie sich aus beiden (Teil-)Abbildungen ablesen lässt, verschlechtert die aktive minimale Click-Konfiguration den Durchsatz - wenn überhaupt - nur marginal. In der linken Abbildung liegt der Durchsatz Click-Konfiguration bei MCS 7 sogar leicht über dem Durchsatz ohne Click. Die Erfassung der Messdaten erfolgte wie in Abschnitt **??** beschrieben mit iperf stufenweise von MCS 0 bis 7, jeweils in 5 Durchläufen für uni- und bidirektionalen Datenfluss.

4.4.3 ARP-Auflösung

Mit der Verwendung von Click funktioniert die ARP-Auflösung des Linux-Netzwerkstacks nicht mehr zuverlässig. Vermutlich greift Click die Pakete ab und leitet diese nicht ordnungsgemäß weiter. In der Folge ist eine Kommunikation auf höheren Protokollschichten (IP, TCP, UDP, ...) nicht mehr möglich. Zur Lösung dieses Problems können zwei Strategien vorgeschlagen werden: Zum einen erlaubt Linux das manuelle Hinzufügen von Einträgen in der ARP-Tabelle über das Kommando "arp -s *IP-Adresse MAC-Adresse*". Automatisiert ließe sich das beispielsweise über einen Eintrag im Linux-Autostart "rc.local" realisieren. Diese Option ist allerdings mit dem Nachteil verbunden, dass die Konfiguration an zwei Stellen gepflegt werden muss und die eigentliche Protokollfunktionalität der Adressauflösung aus dem (fingierten) Protokollstack ausgelagert wird. Als

zweite Möglichkeit bietet Click ein Element namens *ARP-Resolver* an. ARP-Anfragen vom Betriebssystem werden über einen Klassifizierungsmechanismus abgefangen und an diesen Resolver weitergeleitet. Dieser antwortet dem Betriebssystem mit der entsprechend konfigurierten Adresse. Die letztere Vorgehensweise ist auch die in der Click-Dokumentation vorgeschlagene (siehe [56]).

Kapitel 5

Implementierung

In diesem Kapitel wird der Entwurf und die Implementierung der beiden Protokollvarianten FDMA und Token Passing beschrieben. Das bedeutet, dass jeweils zuerst die Spezifikationen und Eigenschaften des jeweiligen Protokolls beschrieben werden. Anschließend wird die Implementierung in Click umgesetzt. Für das Token Passing-Protokoll wird außerdem die Implementierung des Click-Elements "Tokencontroller" beschrieben.

5.1 FDMA

5.1.1 Entwurf

In diesem Kapitel soll die Konzeption und Umsetzung einer Full-Duplex-Kommunikation auf Basis von Frequenzmultiplex (FDMA) beschrieben werden. Im Gegensatz zum Token-Passing ist hier keine sonderlich komplexe Protokolllogik vonnöten. Den Ausgangspunkt bilden zwei IBSS-Funkzellen, von denen je Station ausschließlich eine zum Senden und die andere zum Empfangen von Paketen verwendet werden soll. Die beiden Funkzellen liegen zwecks Interferenzvermeidung auf unterschiedlichen Kanälen beziehungsweise Frequenzen. Die Abbildung 5.1 zeigt das noch einmal grafisch.

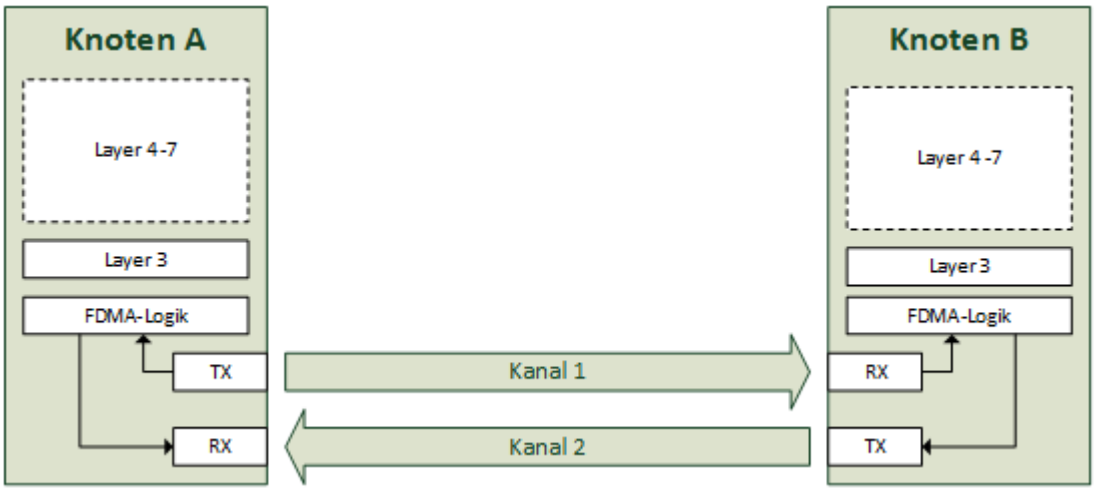


Abbildung 5.1: FDMA-Übertragungsschema (nach eigener Darstellung)

Benötigt wird folglich eine Protokolllogik, die als Verteiler mit Diodenfunktion agiert, d.h. von einem logischen, d.h. einem virtuellen Schnittstellen ein- und ausgehende Datenströme auf unterschiedliche physikalische Schnittstellen abbildet.

5.1.2 Implementierung

Die Umsetzung im Click Modular Router ist entsprechend simpel. Es werden, wie Abbildung 5.2 zeigt, je ein aus- und ein eingehender Datenkanal modelliert. Ausgehende Daten gehen von der

virtuellen Hostschnittstelle an Schnittstelle A (ath0) und werden dort gesendet. Auf Schnittstelle B (ath1) empfangene Daten werden dann an das virtuelle Hostschnittstelle zurück geleitet und können vom darüber liegenden Protokollstack entgegengenommen und verarbeitet werden. Der ARP-Resolver ist aus den in Abschnitt 4.4.3 genannten Gründen notwendig.

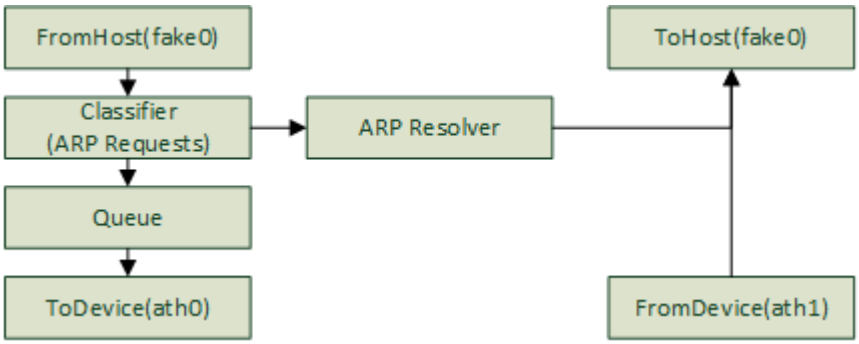


Abbildung 5.2: FDMA - Click-Datenfluss (nach eigener Darstellung)

Die zugehörige Click-Konfiguration setzt sich wie folgt zusammen: Zunächst wird die virtuelle Hostschnittstelle initialisiert und mit einer IP-Adresse konfiguriert. Die MAC-Adresse erhält es von der ausgehenden Schnittstelle, in diesem Fall "*ath1*".

```
dev_fake_in :: FromHost(fake0, PREFIX 172.16.0.1/24, ETHER ath1);
dev_fake_out :: ToHost(fake0);
```

Dann werden die Schnittstelle für die Sende- und Empfangsrichtung initialisiert. Ab jetzt stehen diese dem Linux-Kernel nicht mehr direkt, sondern nur noch über den Click Modular Router zur Verfügung.

```
dev_in :: FromDevice(ath0);
dev_out :: ToDevice (ath1);
```

Es folgt die Initialisierung der Queue "q" und der Kombination aus Klassifizierung und ARP-Resolver. Die Queue ist notwendig, um die Pull-Schnittstelle der Sendeschnittstelle (ToDevice) mit der Push-Schnittstelle der virtuellen Hostschnittstelle (bzw. des Classifiers) bedienen zu können. Erläuterungen zu den unterschiedlichen Schnittstellenarten der Click-Elemente finden sich in Abschnitt 4.3.

```
q :: Queue;
c :: Classifier(12/0806 20/0001, -);
ar :: ARPResponder(172.16.0.2/24 4c:5e:0c:10:fa:ef);
```

Der Classifier wird so eingestellt, dass er ARP-Pakete (Ethernet-Typ 0x0806) vom Typ "Request" (ARP-Typ 0x0001) ausfiltert und an Ausgang 0 weiterleitet. Dazu wird im ersten Schritt das Offset auf 12 Bytes (Ethernet-Typenfeld) gesetzt und der zu prüfende Wert auf "0806" (ARP). Die zweite Stufe der Filterung erfordert ein Offset von 20 Bytes (ARP-Header-Typenfeld). Das Filterkriterium in diesem Fall ist "0001" (Request). Alle anderen Pakete werden an Ausgang 1 weitergereicht. Der ARP-Responder wiederum erhält als Konfiguration die Ziel-IP-Adresse des Partnerknoten, genauer: Die der virtuellen Hostschnittstelle. Als ARP-Antwort hierzu soll er die MAC-Adresse der **empfangenden** Schnittstelle des Partnerknoten zurückgeben.

Der Paketfluss in Senderichtung wird nun wie folgt konfiguriert: Die virtuelle Hostschnittstelle nimmt vom darüber liegenden Protokollstack alle Pakete entgegen und leitet sie an den Klassifizierungsmechanimus weiter. An dessen Ausgang 0 hängt der ARP-Resolver, der seine Antworten zurück an die virtuelle Hostschnittstelle reicht. Von dort finden sie ihren Weg in die ARP-Tabelle des Betriebssystems. Von Ausgang 1 des Klassifizierers geht es weiter durch die Queue zur Sendeschnittstelle.

```
dev_fake_in -> c;
c[0] -> ar -> dev_fake_out;
c[1] -> q -> dev_out;
```

Die Empfangsrichtung ist deutlich simpler. Hier ist auch keine Queue notwendig, da die Elemente DromDevice und ToHost jeweils über Push-Schnittstellen verfügen. Alle von der empfangenden Schnittstelle entgegengenommenen Pakete werden direkt zur virtuellen Hostschnittstelle durchgereicht. Eine Kontrolle auf fehlerhafte oder fälschlich empfangene Pakete muss nicht erfolgen. Das geschieht bereits im darunterliegenden Treiber.

```
dev_in -> dev_fake_out;
```

Damit ist die Click-Konfiguration für FDMA vollständig beschrieben und kann implementiert werden. Falls künftig das Senden über den Monitormodus noch nachgereicht wird, kann dieses nachträglich noch in die Konfiguration übernommen werden. Die Einkapselung der Pakete erfolgt dann entsprechend nach der Queue und das Auspacken der Pakete aus dem Radiotap-Header zwischen Empfangs- und Hostschittstelle.

5.2 Token Passing

5.2.1 Entwurf

Der Aufwand beim Entwurf einer Token Passing-basierten Lösung beschränkt sich nicht nur auf das im Vorkapitel gezeigte Load Balancing. Wie bereits erläutert wurde, teilen sich beim Token Passing alle beteiligten Stationen das Medium und lediglich die Station, die den Token hält, ist zum Senden berechtigt (siehe Abbildung 5.3). Alle anderen Stationen empfangen. Nach dem Senden gibt

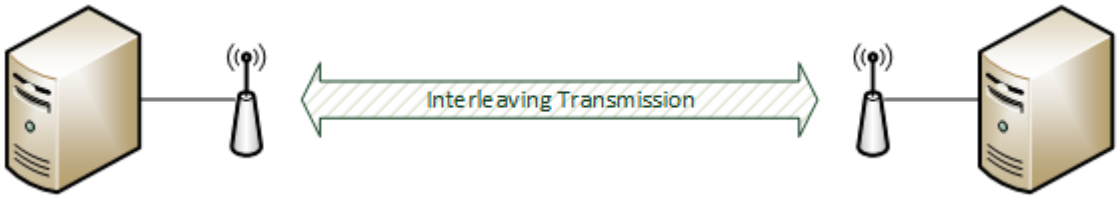


Abbildung 5.3: Token Passing Übertragungsschema (nach eigener Darstellung)

die Station den Token an die nächste Station weiter. Bei einer reinen Punkt-zu-Punkt-Verbindung wird der Token also immer hin- und hergereicht.

Es muss also ein Mechanismus gefunden werden, der diesen Wechsel zwischen den Zuständen ”Senden” (TX) und ”Empfangen” (RX) abbildet. Außerdem sollte wenn möglich noch der Verlust einzelner Token-Pakete abgefangen werden, so dass in diesem Fall der Datenfluss nicht langfristig zum Erliegen kommt. Aus diesen Anforderungen wurde der in Abbildung 5.4 gezeigte Zustandsautomat entwickelt.

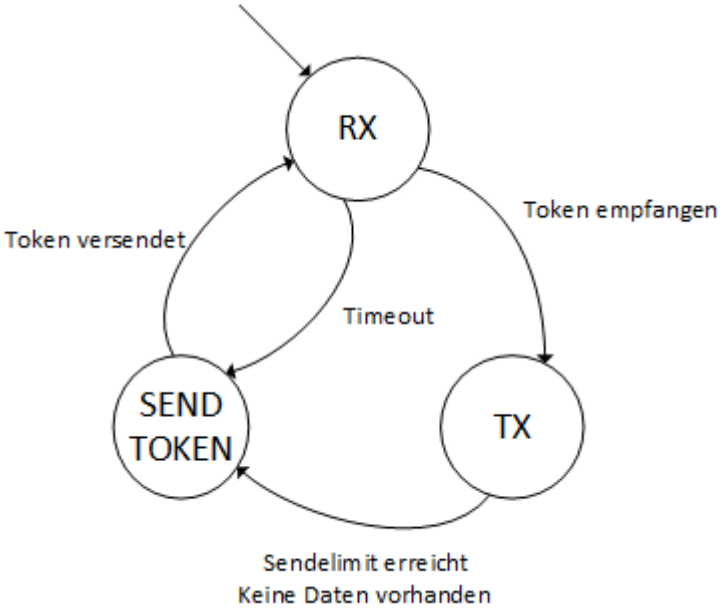


Abbildung 5.4: Token Passing - Zustandsautomat (nach eigener Darstellung)

Zu Beginn befindet sich der Knoten im Empfangszustand (RX). In diesem Zustand wartet er auf den Empfang eines Token. Sobald dieses erhalten wurde, findet ein Wechseln in den Sendezustand statt (TX). In diesem Zustand sendet der Knoten die im Sendepuffer befindlichen Daten Paket für Paket, bis er ein definiertes, konfigurierbares Sendelimit erreicht hat oder keine Daten mehr vorhanden sind. Dann erfolgt ein Wechsel in den Zustand ”SENDTOKEN”. Der Name dieses Zustandes spricht für sich, denn in diesem Zustand wird lediglich der Token an den anderen Teilnehmen weitergereicht. Außerdem dient dieser Zustand der Wiederherstellung der Betriebsbereitschaft nach dem Verlust eines Token-Paketes: Sobald der Knoten in den Zustand ”RX” wechselt, wird ein Zeitgeber gestartet. Wenn vor Ablauf des Zeitgebers kein Token empfangen wurde, also kein Wechsel in den Sendezustand eingeleitet werden kann, dann erfolgt ein Wechsel in den Zustand ”SENDTOKEN”, um dem Gegenüber den Ball erneut zuzuspielen. Dieser Vorgang wiederholt sich solange, bis der Knoten vom Gegenüber entweder einen Token oder Nutzdaten zurückerhält. Damit ist der reguläre Betrieb wiederhergestellt und ein Nutzdatenfluss zwischen den Stationen kann wieder stattfinden.

5.2.2 Click-Konfiguration

Die Implementierung in Click erfolgt analog zu der im vorhergehenden Kapitel gezeigten FDMA-Implementierung, außer, dass es sich bei der physikalischen Sende- und Empfangsschnittstelle um das gleiche Gerät handelt und nicht zwei unterschiedliche. Eine Blockbild der Click-Konfiguration zeigt Abbildung 5.5.

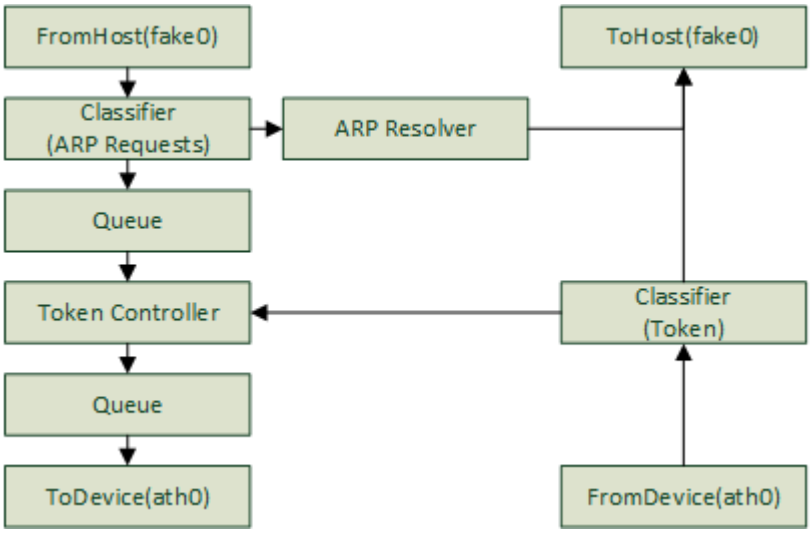


Abbildung 5.5: Token Passing - Click-Datenfluss (nach eigener Darstellung)

Zu Beginn werden alle benötigten Elemente initialisiert und konfiguriert (siehe folgendes Listing). Dabei wird dem Tokencontroller-Element die MAC-Adresse seines Kommunikationspartners übergeben.

```
dev_fake_in :: FromHost(fake0, PREFIX 172.16.0.1/24, ETHER ath0);
dev_fake_out :: ToHost(fake0)
dev_in :: FromDevice(ath0)
dev_out :: ToDevice (ath0)

q_pre :: Queue
q_post :: Queue

c :: Classifier(12/0806 20/0001, -);
ar :: ARPResponder(172.16.0.2/24 4c:5e:0c:10:fa:ef);

tctl :: TokenController(PEER 4c-5e-0c-10-fa-82, MAXDATA 20);
tcl :: Classifier(12/08FF, -);
...
```

Die Senderseite besteht wie bei FDMA zunächst aus einem Klassifizierungsmechanismus (Classifier). Dieser filtert ARP-Requests aus dem Datenstrom und leitet diese an einen ARP-Resolver weiter. Die Antworten werden durch die virtuelle Hostschnittstelle zurück an den darüber liegenden Protokollstack übergeben.

```
...
dev_fake_in -> c;
        c[0] -> ar -> dev_fake_out;
...
```

Alle Pakete, die keine ARP-Requests sind, werden zunächst in eine Queue gegeben. Aus dieser Queue kann sich das Element ”Tokencontroller” bedienen, solange es Sendeberechtigung hat. Es entnimmt die Pakete der Queue und übergibt sie über eine zweite Queue an die sendende Schnittstelle (in diesem Fall ”ath0”). Die Notwendigkeit der zweiten Queue zur Vermittlung zwischen Pull- und Push-Schnittstellen wurde bereits im vorherigen Unterkapitel 5.1 erläutert.

```
...
        c[1] -> q_pre -> [0]tctl -> q_post -> dev_out;
...
```

Auf Empfängerseite werden zunächst mittels eines zweiten Classifiers die Token-Pakete anhand ihres Ethernet-Typs aus dem Datenstrom ausgegliedert und an Eingang 1 des Tokencontrollers weitergeleitet. Die virtuelle Hostschnittstelle nimmt alle anderen Pakete vom Ausgang 1 des Classifiers entgegen und reicht sie an die darüber liegenden Protokollschichten weiter:

```
...

dev_in -> tcl[1] -> dev_fake_out;
tcl[0] -> [1]tctl;
```

Damit ist die Konfiguration des Click Modular Router für Token Passing abgeschlossen. Allerdings wurde in diesem Kapitel das Element "Tokencontroller" verwendet, das nicht in den Standard-Elementen von Click vorhanden, sondern eine Eigenentwicklung ist. Das nächste Unterkapitel widmet sich der Erläuterung dieses Elements.

5.2.3 Click-Element "Tokencontroller"

Auch die Token Passing-Logik muss in Click realisiert werden. Ein vorgefertigtes Element gibt es dafür allerdings nicht. Daher wurde für diese Anforderung ein eigenes Click-Element programmiert, dass den in Abbildung 5.4 gezeigte Automaten umsetzt. Das Grundgerüst für den "Tokencontroller" stammt von den beiden Click-Standard-Elementen "Unqueue" für das Paketmanagement und "TimedSource" für die Timerfunktionalität. Der komplette Quellcode des Elements befindet sich im Anhang.

State Machine

Startzustand:

```
int TokenController::initialize(ErrorHandler *errh)
{
        ...
        _state = RX;
        ...
}
```

Der folgende Code-Teil implementiert den Hauptteil der State Machine:

```
...
//Implement State machine here
switch(_state) {
        case RX:
                //WAIT FOR TOKEN
                if(_tokenPresent)
                        _state = TX;
                break;
        case TX:
                while (worked_packets < _maxData) {
                        if (Packet *p = input(0).pull()) {
                                worked_packets++;
                                worked_bytes+=p->length();
                                output(0).push(p);
                        }
                        else if (!_signal) {
                                ////NoData
                                break;
                        }
                }
                //LIMIT reached
                _state = SENDTOKEN;
                break;
        case SENDTOKEN:
                //Generate & pass token
                token = Packet::make(_headroom, _data.data(), _data.length(), 0);
                if (q = token->push_mac_header(14)) {
                        memcpy(q->data(), &ethh, 14);
                }
                output(0).push(q);
                _tokenPassed++;

                // switch state
                _tokenPresent = false;
```

```
                _state = RX;
                _timer.reschedule_after_ms(_interval);
                break;
        default:
                return false;
}
...
```

Deutlich zu sehen sind die den einzelnen Zuständen zugeordnetem Aktivitäten: Befindet sich der Knoten im empfangenden Zustand, wird nichts unternommen. Wenn dann ein Token übergeben wird, folgt der Übergang in den Sendezustand. In diesem holt eine WHILE-Schleife solange Pakete von Eingang 0 und sendet sie an den Ausgang, bis das Datenlimit erreicht ist oder keine Pakete mehr vorhanden sind. Es folgt der Zustand "SENDTOKEN". In diesem Zustand wird zunächst ein Tokenpaket mit dem zuvor definierten eindeutigen Inhalt generiert. Anschließend wird dem Paket ein Ethernet-Header übergestülpt. Dieser enthält zur Kennzeichnung des Tokenpakets die MAC-Adresse des Gegenübers sowie den nicht belegten Ethernet-Typ $0x08FF$. Das Tokenpaket wird anschließend durch den Ausgang an das Folgeelement übergeben. Abschließend werden noch einige Variablen gesetzt und der Timer zurückgesetzt. Dann folgt der Wechsel in den Empfangszustand

Der Übergang von RX zu SENDTOKEN ist in oben stehendem Auszug nicht zu sehen, da er von der Timerlogik abhängt und in der entsprechenden Subroutine ausgeführt wird:

```
void TokenController::run_timer(Timer *)
{
        if(!_active)
                return;
        if(_state == RX && !_tokenPresent)
        {
                // switch STATE
                _state = SENDTOKEN;
                _timer.reschedule_after_ms(_interval);
        }
}
```

Wenn also der Timer abgelaufen ist, der Knoten sich im Emfangszustand befindet und kein Token vorhanden ist, wird in den Zustand "SENDTOKEN" gewechselt.

Der Tokencontroller erwartet den Token an einem dedizierten Eingang (Eingang 1). Dieser Eingang ist sinnvollerweise ein Push-Eingang, da der Token ja dem Controller aktiv überreicht und nicht passiv zur Abholung bereitgestellt wird. Entsprechend wird die zuständige Push-Subroutine aufgerufen:

```
void TokenController::push(int, Packet *p)
{
        if(_state == RX) {
                //*p = NULL;
                _tokenPresent = true;
        }
}
```

Wenn also der Token überreicht wird, wird die entsprechende globale Boolean-Variable "_tokenPresent" gesetzt. Dadurch wird im nächsten Durchlauf des Zustandsautomaten der entsprechende Zustandsübergang aufgerufen.

Initialisierungswerte und Handler

Wie im Vorabschnitt beschrieben, ist eine Bedingung für den Übergang vom Sende- in den Empfangszustand das Erreichen eines vorgegebenen Zeit- oder Datenlimits. In der aktuellen Version des Tokencontrollers ist dieses Limit beschränkt auf eine maximale Anzahl an Paketen pro Sendedurchgang. Die Erweiterung um ein zeitliches Limit sowie um eine maximale Datenmenge in Byte ist allerdings bereits vorgesehen. Nur die entsprechenden Prüfungen innerhalb des Zustandsautomaten sind aus Zeitgründen noch nicht implementiert. Das folgende Listing zeigt die Initialisierungswerte für die einzelnen Limits und die Auswertung der aus der Click-Konfiguration heraus übergebenen Parameter.

```
int TokenController::configure(Vector<String> &conf, ErrorHandler *errh)
{
        _maxTime = 100;
        _maxData = 100;
        _maxByte = 10000;
```

```
        _interval = 1000;
        _active = true;
        return Args(conf, this, errh)
                .read_p("MAXTIME", _maxTime)
                .read_p("MAXDATA", _maxData)
                .read_p("MAXBYTE",_maxByte)
                .read("PEER",_peer)
                .read("ACTIVE", _active).complete();
}
```

Im Listing auch zu sehen ist der Timeout-Wert für den Fall eines Token-Verlusts. Dieser ist vorerst auf eine Sekunde voreingestellt. Ein Handler für diesen Wert ist allerdings noch nicht implementiert, folgt aber noch.

Für die wichtigsten Parameter, beispielsweise Datenlimit, Status, Anzahl Tokenübergänge, sind zudem Handler für das Click-Prozess-Dateisystem implementiert. Diese Handler werden in der im folgenden Listing abgebildeten Subroutine angelegt.

```
void TokenController::add_handlers()
{
        add_data_handlers("active", Handler::OP_READ | Handler::CHECKBOX, &_active)
            ;
        add_data_handlers("maxtime", Handler::OP_READ, &_maxTime);
        add_data_handlers("maxdata", Handler::OP_READ, &_maxData);
        add_data_handlers("maxbyte", Handler::OP_READ, &_maxByte);
        add_data_handlers("tokenpassed", Handler::OP_READ, &_tokenPassed);
        add_data_handlers("tokenpresent", Handler::OP_READ, &_tokenPresent);
        add_data_handlers("peer", Handler::OP_READ, &_peer);
        //add_data_handlers("state", Handler::OP_READ, &_state);
        add_write_handler("active", write_param, h_active);
        add_write_handler("reset", write_param, h_reset, Handler::BUTTON);
        add_write_handler("maxtime", write_param, h_maxTime);
        add_write_handler("maxdata", write_param, h_maxData);
        add_write_handler("maxbyte", write_param, h_maxByte);
        add_write_handler("tokenpresent", write_param, h_tokenPresent);
        add_task_handlers(&_task, &_signal);
}
```

Auf einen Teil der Parameter wird sowohl Lese- (”add_data_handlers(..)”) als auch Schreibzugriff gewährt (”add_write_handler(..)”). Der Status (”active”) wird dabei beispielsweise explizit als Bool’scher Wert (”CHECKBOX”) angelegt. Ein Handler für den Partnerknoten (”PEER”) ist derzeit noch nicht implementiert. Dieser wird in Kürze noch folgen.

Token-Paket

Wie bereits in Kapitel 4.2.2 beschrieben, kann bis zum jetzigen Zeitpunkt keine Raw Packet Injection genutzt werden, um Pakete an der Medienzugriffskontrolle vorbei zu schleusen. Daher wird die Token Passing-Logik auf den Standard-MAC-Layer aufgesetzt. Eine Teilfunktionalität dieses MAC-Layers ist das Auspacken der zu senden Pakete aus ihrem Ethernet-Header und das Einpacken in einen 802.11-Header.

Das bedeutet, dass in Eigenregie generierte Pakete zunächst mit einem gültigen Ethernet-Header versehen werden müssen, bevor sie an die Medienzugriffskontrolle übergeben werden. Das gilt auch für den im Token Passing verwendeten Token.

Zunächst wird die MAC-Adresse des Gegenüber übergeben. Das geschieht während des Initialisierungsvorganges des Elements bei der Parameterübergabe aus der Click-Konfiguration:

```
int
TokenController::configure(Vector<String> &conf, ErrorHandler *errh)
{
...
.read("PEER", _peer)
...
}
```

Anschließend folgt die Initialisierung der Zählervariable ”_tokenpassed”, die mitzählt, wie oft der Token übergeben wurde. Der State Machine wird mitgeteilt, dass der Token derzeit nicht vorhanden ist (”_tokenPresent = false;”) und die Payload des Tokens wird festgelegt.

```
int TokenController::initialize(ErrorHandler *errh)
{
        ...
```

```
        _tokenPassed = 0;
        _tokenPresent = false;
        _data = "TOKENTOKENTOKEN";
        ...
}
```

Zu Beginn jedes Aufrufs der State Machine wird zunächst der Ethernet-Header für das Tokenpaket generiert (siehe folgendes Listing). Dabei wird als Quell-MAC-Adresse ein Feld mit sechs Nullbytes verwendet, da dieses für die Funktionalität nicht ausschlaggebend ist. Als Zieladresse wird die im Zuge der Initialisierung des Tokencontrollers übergebene Adresse des Partnerknotens verwendet. Das Ethernet-Typenfeld wird auf den bis dato nicht vergebenen Wert "0x08FF" gesetzt, um das Tokenpaket eindeutig zu identifizieren.

```
bool TokenController::run_task(Task *)
{
        Packet *token;
        click_ether ethh;
        uint8_t ethaddr [6] = {0,0,0,0,0,0};
        memcpy(ethh.ether_shost, ethaddr,6);;
        memcpy(ethh.ether_dhost,_peer.data(),6);
        ethh.ether_type=htons(0x08FF);
        WritablePacket *q;
        ...
}
```

Damit ist die Erstellung des Tokenpakets abgeschlossen. In dieser Implementierung wurde der Verwendung eines dedizierten Pakets für die Übertragung des Token der Vorzug gegeben gegenüber einer Piggyback-Lösung. Ersteres fügt sich besser in den Paketfluss von Click ein. Außerdem werden die Nutzdatenpakete nicht verändert. Die für die Übertragung des Tokens notwendige Sendezeit auf dem Medium dürfte zu vernachlässigen sein.

5.2.4 Token-Übertragung

In diesem Abschnitt soll die korrekte Funktion der vorgestellten Token Passing-Implementierung gezeigt werden. Dazu wird mit der Kommandozeilenvariante "tshark" ein Mitschnitt von der Kommunikation erzeugt und anschließend in der grafischen Oberfläche von Wireshark aufbereitet und exportiert. Als Datenquelle für die Mitschnitte wurden jeweils Monitor-Schnittstellen erzeugt und angezapft.

No.	Time	Source	Destination	Protocol	Length	Info
36	46.4836060	00:00:00_00:00:00	4c:5e:0c:10:fa:82	LLC	93	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF
50	48.4836900	00:00:00_00:00:00	4c:5e:0c:10:fa:82	LLC	93	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF
55	49.6572300	00:00:00_00:00:00	4c:5e:0c:10:fa:b9	LLC	110	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF
59	50.4836810	00:00:00_00:00:00	4c:5e:0c:10:fa:82	LLC	93	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF
66	50.4865040	00:00:00_00:00:00	4c:5e:0c:10:fa:b9	LLC	110	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF
76	50.4889170	00:00:00_00:00:00	4c:5e:0c:10:fa:82	LLC	93	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF
77	50.4897110	00:00:00_00:00:00	4c:5e:0c:10:fa:b9	LLC	110	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF
79	50.4900710	00:00:00_00:00:00	4c:5e:0c:10:fa:82	LLC	93	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF
80	50.4912350	00:00:00_00:00:00	4c:5e:0c:10:fa:b9	LLC	110	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF
82	50.4915680	00:00:00_00:00:00	4c:5e:0c:10:fa:82	LLC	93	U, func=UI; SNAP, OUI 0x000000 (Encapsulated Ethernet), PID 0x08FF

Abbildung 5.6: Token Passing: Token-Fluss (Wireshark) (nach eigener Darstellung)

Abbildung 5.6 zeigt den Fluss der Token-Pakete zwischen den Knoten. Die Quell-MAC-Adresse ist dabei zu vernachlässigen, da diese im Tokencontroller derzeit noch nicht gesetzt wird. Aus den Ziel-MAC-Adressen lässt sich entnehmen, dass die kKommunikation - da es sich um eine Punkt-zu-Punkt-Verbindung handelt - zwischen den Knoten mit den MAC-Adressen *4c:5e:0c:10:fa:82* und *4c:5e:0c:10:fa:b9* stattfindet.

Wie zu sehen ist, wird der erste Token (hervorgehobenes Paket) zum Zeitpunkt $t_0 = 46,483$ an den Knoten *4c:5e:0c:10:fa:82* gesendet. Da innerhalb der vorgegebenen Zeit keine Antwort zurück kommt, folgte das nächste Tokenpaket zum Zeitpunkt $t_1 = 48,483$. Mit dem zugehörigen Antwortpaket zum Zeitpunkt $t_2 = 49,457$ wird die Kommunikationsbeziehung etabliert und Nutzdaten können versendet werden.

Die Abbildung 5.7 zeigt den Inhalt eines Token-Pakets entsprechend der Aufzeichnung durch Wireshark. Deutlich zu sehen ist der Text "TOKENTOKENTOKEN", der den Token als solchen identifiziert (rot markiert). Auch der Ethernet-Typ "0x08FF" findet sich hier wieder (blau markiert). Somit ist gezeigt, dass das Token Passing-Verfahren grundsätzlich funktioniert. Die Leistungsfähigkeit wird im folgenden Kapitel ermittelt.

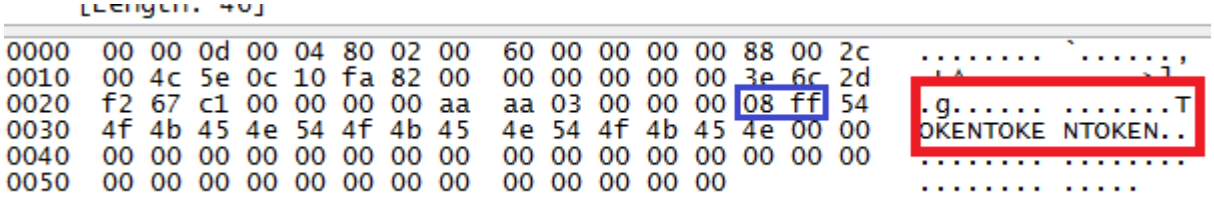


Abbildung 5.7: Token Passing: Inhalt des Token-Pakets (nach eigener Darstellung)

Kapitel 6

Evaluation

Nach der erfolgreichen Implementierung im Vorkapitel erfolgt nun in diesem Kapitel die Leistungserfassung der beiden Protokollvarianten FDMA und Token Passing. Dazu werden zunächst die verwendeten Werkzeuge und die Parameter der Erfassung, das heißt Anzahl der Wiederholungen, Betriebsmodi, Konfigurationen etc. beschrieben. Anschließend erfolgt die Darstellung und Interpretation der Ergebnisse, aus denen sich gegebenenfalls weitere Handlungsschritte und Folgemaßnahmen ableiten lassen.

6.1 Prämissen und Durchführung

Zur Leistungserfassung der beiden implementierten Protokollvarianten sowie aller anderen Versuchsaufbauten wurde das Programm iperf verwendet. Dieses bietet in zwei Betriebsmodi die Möglichkeit, Datendurchsatz und Jitter für UDP- und TCP-Verbindungen zu erfassen. Es handelt sich um eine Client-Server-Applikation. Für weitere Informationen kann die offizielle iperf-Dokumentation unter [58] eingesehen werden.

Damit die Durchsatzergebnisse vergleichbar sind, soll die verwendete verwendete Bandbreite und damit die spektrale Effizienz in allen Versuchsaufbauten jeweils gleich gehalten werden. Das bedeutet: Da im FDMA-Versuchsaufbau mit zwei 20 MHZ breiten Kanälen gearbeitet wird, darf für Token Passing ein 40 MHZ breiter Kanal verwendet werden, ebenso für die Referenzmessungen ohne Click beziehungsweise mit Passthrough-Konfiguration.

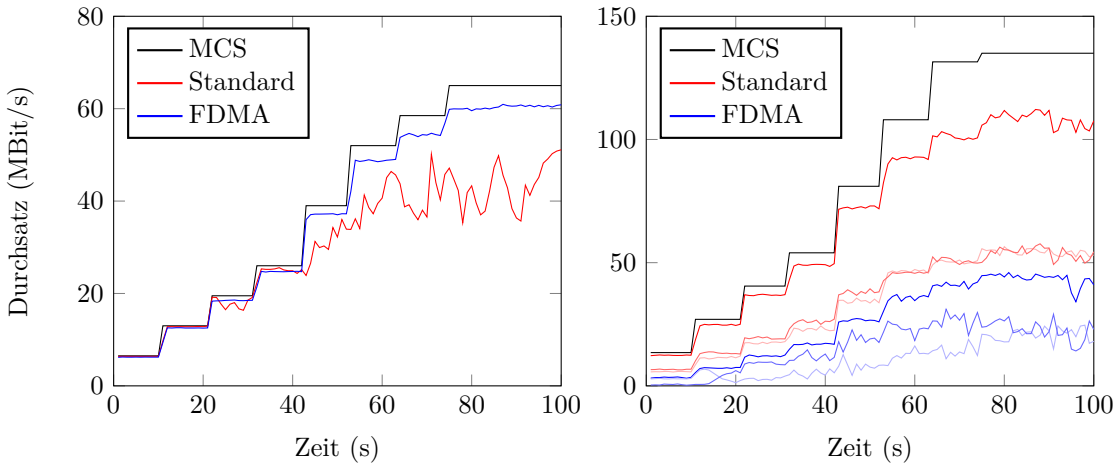
Pro verwendetem Medienzugriffsverfahren werden je fünf Messungen für uni- und bidirektionalen Datenfluss durchgeführt, über die hinterher jeweils der Durchschnitt gebildet wird. Die Dauer jeder Messung ist auf 100 Sekunden begrenzt, und im Verlauf wird die Datenrate schrittweise angehoben. Für Messungen im 802.11n-Betriebsmodus (HT20/HT40) erfolgt dies über die schrittweise Erhöhung des Modulations- und Kodierungsschemas (Modulation and Coding Scheme, MCS) von MCS0 bis MCS7 in 10-Sekunden-Intervallen. Das Token-Verfahren erwies sich in ersten Versuchen leider als instabil im 802.11n-Betrieb, daher werden die Messungen hier im 802.11a/g-Modus durchgeführt. Hier wird die Sendedatenrate ebenfalls in 10-Sekunden-Intervallen schrittweise von 6 auf 54 Megabit pro Sekunde gesteigert, entsprechend der im Standard festgehaltenen Datenraten.

Alle Messungen erfolgen im UDP-Modus von iperf, dabei wird für bidirektionalen Datenfluss der entsprechende Betriebsmodus von iperf gewählt (”iperf -d”).

6.2 Auswertung und Vergleich

6.2.1 FDMA

Die unten stehende Abbildung 6.1 zeigt in blau den Durchsatz in Megabit pro Sekunde der in Abschnitt 5.1 beschriebenen FDMA-Konfiguration sowohl mit uni- als auch mit bidirektionalem Datenfluss. Als Vergleich sind das Resultat ohne aktive Click-Konfiguration in rot und der nach Standard mit dem jeweiligen MCS mögliche Maximalwert in schwarz aufgetragen. In Teilabbildung 6.1b wurden die Datenraten beider Senderichtungen jeweils addiert, um eine Gesamtdatenrate zu erhalten. Die einzelnen Datenströme sind aber in hellrot (Default) bzw. hellblau (FDMA) ebenfalls aufgetragen.



(a) Unidirektionaler Datenfluss

(b) Bidirektionaler Datenfluss

Abbildung 6.1: Vergleich Durchsatz ohne und mit Click-Konfiguration (FDMA) (nach eigener Darstellung)

Wie anhand der Grafik zu sehen ist, ist der unidirektionale Durchsatz der FDMA-MAC-Layer deutlich besser und stabiler als der des Standard-MAC. Er orientiert sich an der maximalen MCS-Rate, bei hohen Datenraten wächst jedoch die Differenz zwischen maximaler und realer Datenrate. Der Durchsatz des Standard-MAC skaliert zu Beginn (d.h. bei niedrigen Datenraten) ebenfalls noch mit der maximalen Datenrate nach MCS, fluktuiert aber hingegen später sehr stark.

Im bidirektionalen Betrieb (Teilabbildung 6.1b) zeigt sich hingegen ein anderes Bild. Hier liegt der Standard-MAC nach kumuliertem Durchsatz vorne, auch wenn dieser ebenfalls bei hohen Datenraten wieder zu fluktuieren beginnt. Die Datenraten des FDMA-MAC brechen im bidirektionalen Betrieb völlig ein und liegen deutlich unter dem jeweils erreichbaren Maximalwert. Der Grund hierfür ist auf den ersten Blick nicht ersichtlich. Hier müssen weitere Versuche zeigen, warum die Datenraten im Verhältnis zum unidirektionalen Betrieb so niedrig ausfallen.

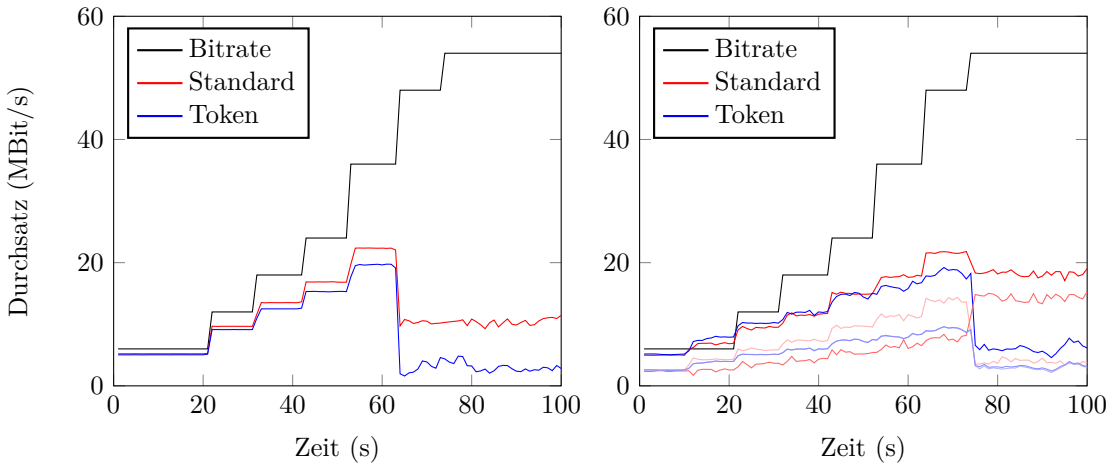
In jedem Fall bleibt die Medienzugriffskontrolle auf Frequenzmultiplexbasis deutlich hinter den an sie gestellten Erwartungen an den Durchsatz zurück. Rein rechnerisch hätte die nutzbare Datenrate um ein Vielfaches höher ausfallen müssen.

6.2.2 Token Passing

Auch für das in Abschnitt 5.2 beschriebene Token Passing-Verfahren wurden Messungen mit uni- und bidirektionalem Datenfluss durchgeführt. Dabei hat sich gezeigt, dass es aus nicht nachvollziehbaren Gründen zu einem Stillstand des gesamten Netzwerkstacks der beiden Knoten kommen kann. Dieser Stillstand war allerdings nur bedingt reproduzierbar, da er scheinbar zufällig auftritt. Dennoch ist das Verhalten des Tokencontroller diesbezüglich als instabil zu bewerten.

Das Ergebnis der Messungen ist in Abbildung 6.2 aufgetragen. Analog zum Vorkapitel findet sich in Teilabbildung 6.2a der unidirektionale Teil wieder mit dem Standard-MAC-Layer in rot und Token Passing in blau. In Teilabbildung 6.2b ist dann der bidirektionale Teil dargestellt, ebenso wie im Vorkapitel auch wieder mit kumuliertem Durchsatz in rot (Standard) und blau (Token Passing) sowie den Teildatenströmen in hellblau (Token Passing) respektive hellrot (Standard).

Bei unidirektionalem Datenfluss ist zu beobachten, dass der Durchsatz des Token Passing-MAC-Layers ähnlich verläuft wie der des Standard-MAC. Beide skalieren zunächst mit der maximal erreichbaren Bitrate, auch sinkt bei beiden mit steigender Bitrate der Wirkungsgrad der Übertragung. Nachdem die maximale Datenrate auf 48 Megabit pro Sekunde angehoben wurde, bricht bei beiden der Durchsatz ein, beim Standard-MAC-Layer allerdings weniger stark als beim Token Passing-MAC. Ein ähnliches Bild zeigt sich auch mit bidirektionalem Datenfluss, hier bricht allerdings die kumulierte Übertragungsrate des Standard-MAC kaum ein. Was sich allerdings zeigen lässt, ist eine Asymmetrie der beiden zugehörigen Datenströme. Diese liegen etwa um den Faktor vier auseinander. Beim Token Passing-Verfahren hingegen liegt diese Asymmetrie nicht vor. Vielmehr sind beide Datenströme so symmetrisch, dass ihre zugehörigen Kurven im Plot exakt aufeinander liegen.

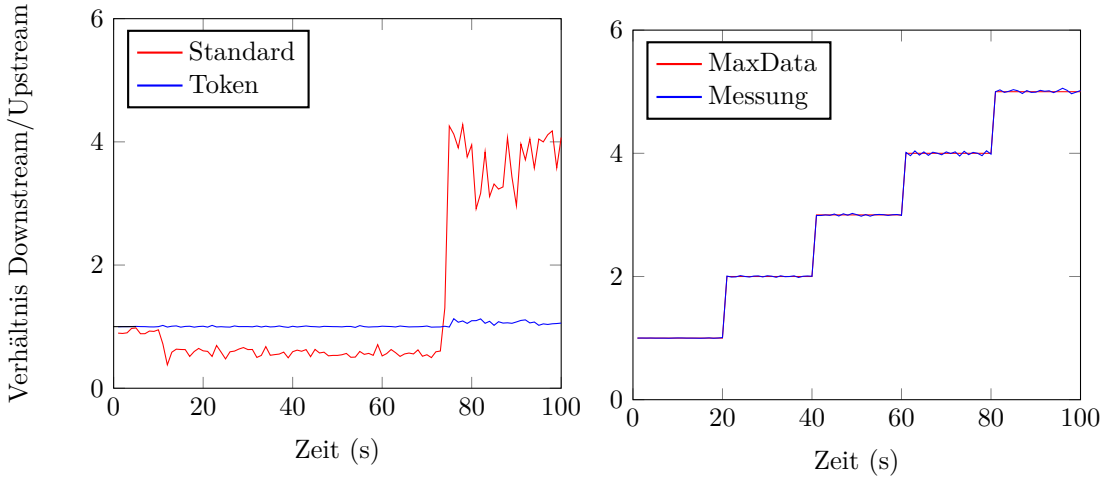


(a) Unidirektionaler Datenfluss

(b) Bidirektionaler Datenfluss

Abbildung 6.2: Vergleich Durchsatz ohne und mit Click-Konfiguration (Token Passing) (nach eigener Darstellung)

Abbildung 6.3a zeigt dieses Verhalten noch deutlicher. Hier sind die jeweiligen Datenströme mit Standard- respektive Token Passing-MAC-Layer im Verhältnis zueinander aufgetragen worden. Was sich zeigen lässt ist, dass das Verhältnis der Datenströme - also welcher Datenstrom den höheren Anteil an der zur Verfügung stehenden Übertragungsrate erhält - im Standard-MAC nicht oder nur schwer vorherzusagen geschweige denn einzustellen ist. Das passende Schlagwort an dieser Stelle wäre ”Fairness”. Es zeigt sich allerdings, dass die Verwendung des Token-Mechanismus, obwohl dieser im Bezug auf auf die Übertragungsrate und Verträglichkeit mit den Übertragungsmodi von 802.11 noch Verbesserungspotential vorhält, diese Fairness bereits garantieren kann.



(a) Verhältnis der Datenströme ohne und mit Click-Konfiguration

(b) Verhältnis der Schwellwertes und Datenraten

Abbildung 6.3: Verhältnis Datenströme Downstream/Upstream (Token Passing) (nach eigener Darstellung)

Anhand dieser Erkenntnis wurde eine weitere Messreihe gestartet. Im Zuge dieser wurde der Datenschwellwert (”maxdata”) des Tokencontrollers auf beiden Seiten auf 10 Pakete festgelegt und anschließend auf einem der beiden Knoten (genauer: der Clientseite) in Zehnerschritten erhöht. Das Zeitintervall betrug dabei 20 Sekunden, die Messung lief wie zuvor über 100 Sekunden. Der Höchstwert betrug also zum Schluss 50 Pakete. Wie zuvor auch wurden 5 Messungen durchgeführt, deren Ergebnisse dann gemittelt wurden. Die Messungen fanden ausschließlich im bidirektionalen Betrieb statt.

In Abbildung 6.3b ist das Ergebnis dieser Messungen aufgetragen. Dabei stellt die rote Kurve

das Verhältnis der jeweils auf den Tokencontrollern eingestellten Datenlimits (”maxdata”) dar, die blaue Kurve das Verhältnis der tatsächlich gemessenen Datenraten. Wie leicht zu sehen ist, liegen beiden Kurven sehr nah beieinander. Lediglich leichte Ausreißer sind auf der blauen Kurve zu erkennen, welche auf Schwankungen während der Messungen zurückzuführen sind.

Festgehalten werden kann also, dass von einem Produktiveinsatz des Token Passing-Verfahrens aufgrund der mangelnden Leistungsfähigkeit und Instabilität derzeit noch abgesehen werden sollte. Allerdings haben die Versuche auch die Möglichkeiten gezeigt, die diese Form der Medienzugriffskontrolle im Bezug auf die Regulierung der für die jeweilige Senderichtung zur Verfügung stehenden Datenrate bietet.

Kapitel 7

Zusammenfassung

Gegenstand dieser Arbeit war die Erarbeitung und Untersuchung von alternativen Mechanismen zur Medienzugriffskontrolle für drahtlose Langstreckenfunknetze nach 802.11. Dafür wurde zunächst ein Einblick in den grundsätzlichen Aufbau dieser Funknetze gegeben, ebenso wie eine Begründung für deren sinnvollen Einsatz bei der Bereitstellung von Internetzugängen mit hohen Datenraten in bislang schlecht oder gar nicht erschlossenen Gegenden. Auch wurde darauf hingewiesen, warum sich die im Standard 802.11 spezifizierte Medienzugriffskontrolle auf der Basis von CSMA/CA nur bedingt für Langstrecken- respektive Punkt-zu-Punkt-Verbindungen eignet.

In Kapitel 2 folgte ein Überblick über verwandte Arbeiten, unter anderem auch zur Optimierung des Standard-MAC-Layers durch Justierung diverser Parameter. Ebenfalls in diesem Kapitel zu finden ist ein Überblick über die unterschiedlichen Arten, eine Medienzugriffskontrolle zu etablieren. Hier wurde erläutert, auf welche Arten ein Medium unterteilt werden kann, und wo diese praktische Anwendung finden, gefolgt von der Vorstellung bereits existierender alternativer MAC-Layer-Protokolle für 802.11.

Im Anschluss wurden in Kapitel 3 verschiedene Frameworks zur Entwicklung von MAC-Protokollen für Funkverbindungen evaluiert. Hierfür wurden zunächst Kriterien festgelegt und erläutert, anhand derer die Evaluierung stattfand. Die verschiedenen Frameworks wurden umreißend vorgestellt und anschließend anhand der Kriterien bewertet. Wie sich zeigte, war der Click Modular Router ([40]) das am besten geeignete Werkzeug für die Implementierung eigener MAC-Protokolle, unter anderem auch wegen seiner Hardware-Unabhängigkeit.

Kapitel 4 beinhaltet eine ausführliche Vorstellung der Versuchsanordnung beziehungsweise der Testumgebung, unterteilt in die Beschreibung der beteiligten Rechnerknoten, Grundlagen zum Thema "Raw Packet Injection" und abschließend eine Einführung in Funktionsweise und Eigenschaften des Click Modular Router. Insbesondere wurde Wert gelegt auf Reproduzierbarkeit, daher enthält der erste Teil zwar auch eine Auflistung der Hardware-Spezifikationen, aber auch eine Anleitung zur Installation und Konfiguration von Ubuntu Linux auf den eingesetzten APU1C-Boards. Außerdem werden Anpassungen an der CRDA-Datenbank beschrieben, die eine freie Wahl der Sendefrequenz und -leistung unabhängig von regionalen Regulationsvorgaben ermöglichen sollten, jedoch ohne Wirkung blieben. Im Zusammenspiel zwischen dem Click Modular Router und dem Treiber ath9k ergaben sich im Laufe des Projektes Schwierigkeiten, deren Lösung ebenfalls in diesem Kapitel diskutiert wird. Stichworte sind hier Sendewarteschlangenzuweisung und ARP-Auflösung.

Die eigenliche Hauptarbeit dieser Ausarbeitung, Entwurf und Implementierung je eines FDMA- und Token Passing-Protokols, wird in Kapitel 5 beschrieben. Die Komplexität des FDMA-Teils ist verhältnismäßig gering, dementsprechend einfach ist die Umsetzung in eine Konfiguration für den Click Modular Router. Die Im Vorkapitel festgestellte ARP-Problematik wurde ebenfalls berücksichtigt. Für das Token Passing-Protokoll wurde zunächst anhand eines Zustandsautomaten das Verhalten des Protokolls festgelegt. Anschließend wurde eine Click-Konfiguration konstruiert und ein neues Element entworfen, das das Verhalten des Zustandsautomaten umsetzen soll. Die Implementierung dieses neuen Elements "Tokencontroller" erfolgte in C++ unter Einbindung und Berücksichtigung der benötigten Click-Bibliotheken und Subroutinen. In der Implementierungsbeschreibung wurden insbesondere die Teilbereiche Zustandsautomat, Initialisierungswerte und Handler und Token-Paket bedacht. Die grundsätzliche Funktionsbereitschaft wurde anhand eines Paket-Mitschnitts (Wireshark-Trace) nachgewiesen.

Die zuvor implementierten Protokolle müssen sich in Kapitel 6 einer Bewertung ihrer Leistungs-

fähigkeit im Bezug auf Durchsatz bzw. Datenrate unterziehen. Die hierfür notwendigen Belastungstests werden mit dem Programm "iperf" jeweils mit uni- und mit bidirektionalem Datenfluss durchgeführt. Bei FDMA zeigt sich, dass im unidirektionalen Betrieb der Durchsatz vergleichbar ist mit dem der Standard-Medienzugriffskontrolle. Da FDMA allerdings auf bidirektionalen Betrieb ausgelegt ist, ist dieses Ergebnis nahezu irrelevant. Die bidirektionalen Messungen bleiben weit hinter den durch den unidirektionalen Test geweckten Erwartungen zurück, der Durchsatz liegt um ein Vielfaches niedriger als der durch die Referenzmessung ohne alternative Medienzugriffskontrolle vorgegebene Wert.

Das Token Pasing-Verfahren hatte ebenfalls Performance-Schwierigkeiten im bidirektionalen Betrieb. Außerdem zeigte sich, dass der Betrieb der WLAN-Schnittstellen im 802.11n-Modus zu deutlichen Leistungseinbußen führt. Die Ergebnisse stammen also aus Messungen im 802.11a-Modus. Eine interessante Erkenntnis ist jedoch diesen Messungen ist jedoch, dass die Symmetrie zwischen den beiden gerichteten Datenströmen mit dem Verhältnis der in den Tokencontroller-Elementen festgelegten Schwellwerten skaliert. Weitere Messungen, in denen das Verhältnis während der Messung angepasst wurde, belegen dies ebenfalls. So ließe sich beispielsweise durch einen adaptiven Algorithmus das Verhältnis zur Laufzeit an geänderte Verkehrsverhältnisse anpassen.

Zusammenfassend lässt sich sagen, dass das Projekt zumindest teilweise erfolgreich abgeschlossen werden kann. Zwei neue Protokolle wurden konstruiert, wenn auch mit deutlichen Schwächen der Leistungsfähigkeit. Auf der anderen Seite wurden viele Erkenntnisse gewonnen über das Zusammenspiel zwischen den verschiedenen Software- und Hardware-Komponenten der aufgebauten Testumgebung. Diese sowie das erarbeitete Fachwissen können gegebenenfalls die Einarbeitungszeit in künftige Problemstellungen deutlich beschleunigen.

Auch die im Rahmen dieser Arbeit gewonnen Erkenntnisse im Bezug auf das Verhalten der beiden implementierten Protokolle können zu weiteren Forschungsarbeit anregen. Im Bereich des Token Passing ist beispielsweise vorstellbar, mit Raw Frame Injection weiterzuforschen oder adaptive Algorithmen zu entwickeln, die die Größe der Sendefenster automatisch an gegebenen Verkehrsbedingungen ausrichten. Auch eine Forcierung der Raw Packet Injection zusammen mit dem Click Modular Router und eine Implementierung entsprechender Click-Elemente ist vorstellbar.

Literaturverzeichnis

[1] D. Camps-Mur, "Linux wi-fi open source drivers - mac80211, ath9k/ath5k," 2011. [Online]. Available: http://www.campsmur.cat/files/mac80211_intro.pdf

[2] Linux wireless drivers. [Online]. Available: http://wireless.kernel.org/en/users/Drivers

[3] Wikidevi atheros ath9k. [Online]. Available: https://wikidevi.com/wiki/Ath9k

[4] Wikidevi broadcom brcmsmac. [Online]. Available: https://wikidevi.com/wiki/Brcmsmac

[5] Wikidevi intel iwlwifi. [Online]. Available: https://wikidevi.com/wiki/Iwlwifi

[6] Wikidevi marvell mwl8k. [Online]. Available: https://wikidevi.com/wiki/Mwl8k

[7] Wikidevi ralink rt2800pci. [Online]. Available: https://wikidevi.com/wiki/Rt2800pci

[8] Bundesministerium für Verkehr und digitale Infrastruktur. (2014, Nov.) Breitbandatlas. [Online]. Available: http://www.zukunft-breitband.de/Breitband/DE/Breitbandatlas/BreitbandVorOrt/breitband-vor-ort_node.html

[9] M. Rademacher, M. Kretschmer, and K. Jonas, "Exploiting ieee802. 11n mimo technology for cost-effective broadband back-hauling," in *e-Infrastructure and e-Services for Developing Countries*. Springer, 2014, pp. 1–11.

[10] M. Chauchet, "Alternative mac layer protocols for ieee 802.11 wireless long distance point-to-point links."

[11] F.-J. Kauffels.

[12] H. Orlamünder.

[13] J. F. Kurose and K. W. Ross.

[14] I. Hussain, N. Sarma, and D. Saikia, "Tdma mac protocols for wifi-based long distance networks: A survey," *International Journal of Computer Applications*, vol. 94, no. 19, pp. 1–8, 2014.

[15] B. Raman and K. Chebrolu, "Design and evaluation of a new mac protocol for long-distance 802.11 mesh networks," in *Proceedings of the 11th annual international conference on Mobile computing and networking*. ACM, 2005, pp. 156–169.

[16] R. K. Patra, S. Nedevschi, S. Surana, A. Sheth, L. Subramanian, and E. A. Brewer, "Wildnet: Design and implementation of high performance wifi based long distance networks." in *NSDI*, vol. 1, no. 1, 2007, p. 1.

[17] Y. Ben-David, M. Vallentin, S. Fowler, and E. Brewer, "Jaldimac: taking the distance further," in *Proceedings of the 4th ACM workshop on networked systems for developing regions*. ACM, 2010, p. 2.

[18] S. Nedevschi, R. K. Patra, S. Surana, S. Ratnasamy, L. Subramanian, and E. Brewer, "An adaptive, high performance mac for long-distance multihop wireless networks," in *Proceedings of the 14th ACM international conference on Mobile computing and networking*. ACM, 2008, pp. 259–270.

[19] A. Dhekne, N. Uchat, and B. Raman, "Implementation and evaluation of a tdma mac for wifi-based rural mesh networks," *NSDR'09*, 2009.

[20] S. Leffler, "Tdma for long distance wireless networks," 2009.

[21] L. Eznarriaga, C. Senguly, N. Bayer, J. I. Moreno, P. Lozano, and M. Simón, "Experiences with softtoken: a token-based coordination layer for wlans," *International Journal of Communication Systems*, 2013.

[22] S. Choi, "Wireless mac protocol based on a hybrid combination of slot allocation, token passing, and polling for isochronous traffic," Sep. 21 2004, uS Patent 6,795,418.

[23] P. Dutta, S. Jaiswal, D. Panigrahi, and R. Rastogi, "A new channel assignment mechanism for rural wireless mesh networks," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE.* IEEE, 2008.

[24] R. M. Karp, *Reducibility among combinatorial problems.* Springer, 1972.

[25] M. Duarte and A. Sabharwal, "Full-duplex wireless communications using off-the-shelf radios: Feasibility and first results," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on.* IEEE, 2010, pp. 1558–1562.

[26] J. I. Choi, M. Jain, K. Srinivasan, P. Levis, and S. Katti, "Achieving single channel, full duplex wireless communication," in *Proceedings of the sixteenth annual international conference on Mobile computing and networking.* ACM, 2010, pp. 1–12.

[27] G. Orfanos, J. Habetha, and L. Liu, "Mc-cdma based ieee 802.11 wireless lan," in *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on.* IEEE, 2004, pp. 400–405.

[28] The madwifi project. [Online]. Available: http://madwifi-project.org/

[29] M. Vipin and S. Srikanth, "Analysis of open source drivers for ieee 802.11 wlans," in *Wireless Communication and Sensor Computing, 2010. ICWCSC 2010. International Conference on.* IEEE, 2010, pp. 1–5.

[30] F. Gringoli, "A glimpse into the linux wireless core: From kernel to firmware," University of Brescia, 2011.

[31] S. M. Gunther, M. Leclaire, J. Michaelis, and G. Carle, "Analysis of injection capabilities and media access of ieee 802.11 hardware in monitor mode," in *Network Operations and Management Symposium (NOMS), 2014 IEEE.* IEEE, 2014, pp. 1–9.

[32] A. S. Tanenbaum.

[33] J. Liedtke, "On microkernel construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, Dec. 1995. [Online]. Available: http://l4ka.org/publications/

[34] J. Ousterhout, "Why aren't operating systems getting faster as fast as hardware," in *In Summer USENIX'90*, 1990.

[35] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald, "Softmac-flexible wireless research platform," in *Proc. HotNets-IV*, 2005.

[36] A. Sharma, M. Tiwari, and H. Zheng, "Madmac: Building a reconfiguration radio testbed using commodity 802.11 hardware," in *Networking Technologies for Software Defined Radio Networks, 2006. SDR'06.1 st IEEE Workshop on.* IEEE, 2006, pp. 78–83.

[37] ——, "Madmac: Building a reconfigurable radio testbed using commodity 802.11 hardware - presentation," 2007. [Online]. Available: http://moment.cs.ucsb.edu/ asharma/madmac-presentation.pdf

[38] A. Sharma and E. M. Belding, "Freemac: framework for multi-channel mac development on 802.11 hardware," in *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*. ACM, 2008, pp. 69–74.

[39] A. Rao and I. Stoica, "An overlay mac layer for 802.11 networks," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM, 2005, pp. 135–148.

[40] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[41] Pc engines: Apu 1 series system board. [Online]. Available: http://www.pcengines.ch/pdf/apu1.pdf

[42] Routerboard r11e-5hnd specifications. [Online]. Available: http://routerboard.com/R11e-5HnD

[43] Rf elements: Stationbox alu. [Online]. Available: http://www.rfelements.com/en/products/enclosures/stationbox-alu/

[44] Vmware workstation 10 documentation: Connecting usb devices to virtual machines. [Online]. Available: https://pubs.vmware.com/workstation-10/index.jsp#com.vmware.ws.using.doc/GUID-E003456F-EB94-4B53-9082-293D9617CB5A.html

[45] Ubuntu long term support. [Online]. Available: https://wiki.ubuntu.com/LTS

[46] Unetbootin - homepage and downloads. [Online]. Available: http://unetbootin.sourceforge.net/

[47] Fedora liveusb creator. [Online]. Available: https://fedorahosted.org/liveusb-creator/

[48] Pendrive linux: Universal usb installer. [Online]. Available: http://www.pendrivelinux.com/universal-usb-installer-easy-as-1-2-3/

[49] Apu + ubuntu 14.04 lts - install via serial console. [Online]. Available: http://pcengines.info/forums/?page=post&id=E25612E9-84F0-4DCF-A876-1E92FD1D065C

[50] Lwn: Atheros releases ath5k hal code. [Online]. Available: http://lwn.net/Articles/300758/

[51] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli, "Wireless mac processors: programming mac protocols on commodity hardware," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1269–1277.

[52] Linux wireless: Regulatory. [Online]. Available: http://linuxwireless.org/en/developers/Regulatory/

[53] Binary factory: Setting tx power as high as your card can go. [Online]. Available: http://blog.binaryfactory.ca/2012/09/setting-txpower-as-high-as-your-card-can-go/

[54] Override regulatory setting. [Online]. Available: http://pages.cs.wisc.edu/ govindan/steps

[55] How to: Compile linux kernel modules. [Online]. Available: http://www.cyberciti.biz/tips/compiling-linux-kernel-module.html

[56] Click modular router online documentation. [Online]. Available: http://www.read.cs.ucla.edu/click/click

[57] Gmane.org: [patch 2/2] mac80211: Set up tx-queue-mapping in subif_start_xmit. [Online]. Available: http://permalink.gmane.org/gmane.linux.kernel.wireless.general/64587

[58] Iperf - the tcp/udp bandwidth measurement tool. [Online]. Available: https://iperf.fr/

Anhang A

Appendix

A.1 Inbetriebnahme Testumgebung

A.1.1 Installation Betriebssystem

Nach [49]:

Listing A.1: Anpassung der Datei isolinux.cfg

```
#D-I config version 2.0
CONSOLE 0
SERIAL 0 115200 0
include menu.cfg
default vesamenu.c32
prompt 0
timeout 0
```

Listing A.2: Anpassung der Datei txt.cfg

```
default install
label install
menu label \^Install Ubuntu Server
kernel /install/vmlinuz
append  file=/cdrom/preseed/ubuntu-server.seed vga=788 initrd=/install/initrd.gz --
     console=ttyS0,115200n8 quiet -
```

Listing A.3: Anpassung der Datei syslinux.cfg

```
#D-I config version 2.0
CONSOLE 0
SERIAL 0 115200 0

default menu.c32
prompt 0
menu title UNetbootin
timeout 100
label unetbootindefault
kernel /install/netboot/ubuntu-installer/amd64/linux
append initrd=/install/netboot/ubuntu-installer/amd64/initrd.gz tasks=standard
    pkgsel/language-pack-patterns= pkgsel/install-language-support=false vga=788 --
     console=ttyS0,115200n8 -- quiet
```

A.1.2 Konfiguration nach der Installation

Listing A.4: Aktualisieren der Paketquellen und Installation des Texteditors "vim"

```
~# sudo su -
~# passwd

~# aptitude update && apt-get dist-upgrade
~# aptitude install vim
```

Listing A.5: Anpassen der SSH-Konfiguration (PermitRootLogin=yes) und Erstellen eines neuen Public-Private-Key-Pärchens

```
~# vim /etc/ssh/sshd_config
~# service ssh restart
~# ssh-keygen
```

Listing A.6: Auszug aus der Datei /etc/hostname

```
apu-a4f2
```

Listing A.7: Auszug aus der Datei /etc/network/interfaces

```
# the loopback network interface
auto lo
iface lo inet loopback
# The primary network interface
auto p4p1
# Beim Bootvorgang automatisch starten
iface p4p1 inet static
        address 10.20.111.100
        netmask 255.255.255.0
        gateway 10.20.111.1
        dns-nameservers 194.95.66.9
        dns-search inf.fh-bonn-rhein-sieg.de
```

Listing A.8: Auszug aus der Datei /etc/motd

```
WiBACK, www.wiback.org
(c) Fraunhofer FOKUS 2013

Testbed Masterprojekt
Wintersemester 2014/2015
Martin Chauchet
martin.chauchet@smail.inf.h-brs.de

Node alpha (apu-a4f2)
```

Listing A.9: Auszug aus der Datei /etc/udev/rules.d/70-persistent-rules

```
# This file was automatically generated by the /lib/udev/write_net_rules
# program, run by the persistent-net-generator.rules rules file.
#
# You can modify it, as long as you keep each rule on a single
# line, and change only the value of the NAME= key.


# PCI device 0x168c:0x0033 (ath9k)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="4c:5e:0c:10:fa:7c",
     ATTR{dev_id}=="0x0", ATTR{type}=="1", KERNEL=="wlan*", NAME="ath0"

# PCI device 0x168c:0x0033 (ath9k)
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="?*", ATTR{address}=="4c:5e:0c:10:fa:b9",
     ATTR{dev_id}=="0x0", ATTR{type}=="1", KERNEL=="wlan*", NAME="ath1"
```

Listing A.10: Auszug aus der Datei /root/.ssh/authorized_keys

```
ssh-dss AAAAB3NzaC1 ... C1DclNnQ== root@wibackauth
ssh-dss AAAAB3NzaC1 ... +GpgyL114= updates@wiback.org
ssh-rsa AAAAB3NzaC1 ... BlH8PwjRiN root@apu-a648
ssh-rsa AAAAB3NzaC1 ... kiCjuTskmh root@node-B
ssh-rsa AAAAB3NzaC1 ... 3dZXmxjWK1 root@node-beta
```

Listing A.11: Installation zusätzlich benötiger Software

```
~# aptitude install iw wireless-regdb
~# aptitude install gcc g++ build-essential git make autoconf
```

Listing A.12: Anlegen der Arbeitsumgebung

```
~# cd ~
~# mkdir src config config/click capture
```

A.2 Click Modular Router

A.2.1 Download & Installation

Listing A.13: Check-Out und Installation des Click Modular Router

```
~# cd ~/src && git clone git://github.com/kohler/click
~# cd click
~# ./configure --enable-userlevel --enable-linuxmodule --enable-all-elements --
    enable-local
~# make && make install
```

A.2.2 Eigene Click-Elemente

Listing A.14: Header-Datei Token-Controller (elements/local/tokencontroller.hh)

```
#ifndef CLICK_TOKENCONTROLLER_HH
#define CLICK_TOKENCONTROLLER_HH
#include <click/element.hh>
#include <click/task.hh>
#include <click/notifier.hh>
#include <clicknet/ether.h>
#include <click/etheraddress.hh>
CLICK_DECLS

/*
=c
TokenController

=s local
receives Token in input 2 and unqueues Elements from above pull element (input 1)
    and pushes them out (output 1)

=d
Emits Token when timeout reached or maxData has been sent or no Data is left

=n

=a
Unqueue
*/

class TokenController : public Element { public:

        TokenController() CLICK_COLD;

        const char *class_name() const  { return "TokenController"; }
        const char *port_count() const  { return "2/1"; }
        const char *processing() const  { return "lh/h"; }

        int configure(Vector<String> &, ErrorHandler *) CLICK_COLD;
        int initialize(ErrorHandler *) CLICK_COLD;
        void add_handlers() CLICK_COLD;

        bool run_task(Task *);
        void push(int, Packet *);

        private:
                bool _active;
                enum state {
                        RX, TX
                };
                enum state _state;
                int32_t _maxData;
                int32_t _maxByte;
                int32_t _maxTime;
                int32_t _tokenPassed;
                Task _task;
                NotifierSignal _signal;
                bool _tokenPresent;
                uint32_t _headroom;
```

```
                EtherAddress _peer;

                String _data;

                enum {
                                h_active, h_reset, h_maxTime, h_maxData, h_maxByte,
                                     h_tokenPresent,h_state
                };
                static int write_param(const String &, Element *, void *,
                    ErrorHandler *) CLICK_COLD;
};

CLICK_ENDDECLS
#endif
```

Listing A.15: Programm-Datei Token-Controller (elements/local/tokencontroller.cc)

```
// -*- c-basic-offset: 4 -*-
/*
* tokencontroller.{cc,hh} -- element pulls as many packets as possible from
* its input, pushes them out its output
* Eddie Kohler
*
* Copyright (c) 1999-2000 Massachusetts Institute of Technology
* Copyright (c) 2002 International Computer Science Institute
*
* Permission is hereby granted, free of charge, to any person obtaining a
* copy of this software and associated documentation files (the "Software"),
* to deal in the Software without restriction, subject to the conditions
* listed in the Click LICENSE file. These conditions include: you must
* preserve this copyright notice, and you cannot mention the copyright
* holders in advertising related to the Software without their permission.
* The Software is provided WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED. This
* notice is a summary of the Click LICENSE file; the license in that file is
* legally binding.
*/

#include <click/config.h>
#include "tokencontroller.hh"
#include <click/args.hh>
#include <click/error.hh>
#include <click/standard/scheduleinfo.hh>
#include <click/etheraddress.hh>

CLICK_DECLS

TokenController::TokenController()
: _task(this), _timer(this)
{
}

int
TokenController::configure(Vector<String> &conf, ErrorHandler *errh)
{
_maxTime = 100;
_maxData = 100;
_maxByte = 10000;
_interval = 1000;
_active = true;
return Args(conf, this, errh)
.read_p("MAXTIME", _maxTime)
.read_p("MAXDATA", _maxData)
.read_p("MAXBYTE", _maxByte)
.read("PEER", _peer)
.read("ACTIVE", _active).complete();
}

int
TokenController::initialize(ErrorHandler *errh)
{
ScheduleInfo::initialize_task(this, &_task, _active, errh);
_signal = Notifier::upstream_empty_signal(this, 0, &_task);
_state = RX;
_tokenPassed = 0;
```

```
_tokenPresent = false;
_data = "TOKENTOKENTOKEN";
_headroom = Packet::default_headroom;
_timer.initialize(this);
if (_active)
_timer.schedule_after_ms(_interval);
return 0;
}

bool
TokenController::run_task(Task *)
{
int worked_packets = 0, worked_bytes = 0;
Packet *token;
click_ether ethh;
uint8_t ethaddr [6] = {0,0,0,0,0,0};
memcpy(ethh.ether_shost, ethaddr,6);;
memcpy(ethh.ether_dhost,_peer.data(),6);
ethh.ether_type=htons(0x08FF);
WritablePacket *q;

if (!_active)
return false;
else {
//Implement State machine here
switch(_state) {
case RX:
//WAIT FOR TOKEN
if(_tokenPresent)
_state = TX;
break;
case TX:
while (worked_packets < _maxData) {
if (Packet *p = input(0).pull()) {
worked_packets++;
worked_bytes+=p->length();
output(0).push(p);
}
else if (!_signal) {
////NoData
break;
}
}
//LIMIT reached
_state = SENDTOKEN;
break;
case SENDTOKEN:
//Generate & pass token
token = Packet::make(_headroom, _data.data(), _data.length(), 0);
if (q = token->push_mac_header(14)) {
memcpy(q->data(), &ethh, 14);
}

output(0).push(q);
_tokenPassed++;

// switch state
_tokenPresent = false;
_state = RX;
_timer.reschedule_after_ms(_interval);
break;
default:
return false;
}
_task.fast_reschedule();
}
out:
return 0;
}

void
TokenController::push(int, Packet *p)
{
```

```
if(_state == RX) {
//*p = NULL;
_tokenPresent = true;
}
}


void
TokenController::run_timer(Timer *)
{
if(!_active)
return;
if(_state == RX && !_tokenPresent)
{
_state = SENDTOKEN;
_timer.reschedule_after_ms(_interval);
}
}


int
TokenController::write_param(const String &conf, Element *e, void *user_data,
    ErrorHandler *errh)
{
TokenController *t = static_cast<TokenController *>(e);
switch (reinterpret_cast<intptr_t>(user_data)) {
case h_active:
if (!BoolArg().parse(conf, t->_active))
return errh->error("syntax␣error");
break;
case h_tokenPresent:
if (!BoolArg().parse(conf, t->_tokenPresent))
return errh->error("syntax␣error");
break;
case h_maxTime:
if (!IntArg().parse(conf, t->_maxTime))
return errh->error("syntax␣error");
break;
case h_maxData:
if (!IntArg().parse(conf, t->_maxData))
return errh->error("syntax␣error");
break;
case h_maxByte:
if (!IntArg().parse(conf, t->_maxByte))
return errh->error("syntax␣error");
break;
}

if (t->_active && !t->_task.scheduled())
t->_task.reschedule();
return 0;
}

void
TokenController::add_handlers()
{
add_data_handlers("active", Handler::OP_READ | Handler::CHECKBOX, &_active);
add_data_handlers("maxtime", Handler::OP_READ, &_maxTime);
add_data_handlers("maxdata", Handler::OP_READ, &_maxData);
add_data_handlers("maxbyte", Handler::OP_READ, &_maxByte);
add_data_handlers("tokenpassed", Handler::OP_READ, &_tokenPassed);
add_data_handlers("tokenpresent", Handler::OP_READ, &_tokenPresent);
add_data_handlers("peer", Handler::OP_READ, &_peer);
//add_data_handlers("state", Handler::OP_READ, &_state);
add_write_handler("active", write_param, h_active);
add_write_handler("reset", write_param, h_reset, Handler::BUTTON);
add_write_handler("maxtime", write_param, h_maxTime);
add_write_handler("maxdata", write_param, h_maxData);
add_write_handler("maxbyte", write_param, h_maxByte);
add_write_handler("tokenpresent", write_param, h_tokenPresent);
add_task_handlers(&_task, &_signal);
}
```

```
CLICK_ENDDECLS
EXPORT_ELEMENT(TokenController)
ELEMENT_MT_SAFE(TokenController)
```

A.2.3 Konfigurationsdateien

Listing A.16: Click-Konfiguration: Passthrough (WifiDefault_kernel.click)

```
dev_fake_in :: FromHost(fake0, PREFIX 172.16.0.1/24, ETHER ath0);
dev_fake_out :: ToHost(fake0);

dev_in :: FromDevice(ath0);
dev_out :: ToDevice (ath0);
q :: Queue;

c :: Classifier(12/0806 20/0001, -);
ar :: ARPResponder(172.16.0.2/24 4c:5e:0c:10:fa:82);

dev_fake_in -> c;
c[0] -> ar -> dev_fake_out;
c[1] -> q -> dev_out;

dev_in -> dev_fake_out;
```

Listing A.17: Click-Konfiguration: FDMA mit ARP-Auflösung (WifiDuplex_kernel_arp.click)

```
dev_fake_in :: FromHost(fake0, PREFIX 172.16.0.1/24, ETHER ath1);
dev_fake_out :: ToHost(fake0);

dev_in :: FromDevice(ath0);
dev_out :: ToDevice (ath1);
q :: Queue;

c :: Classifier(12/0806 20/0001, -);
ar :: ARPResponder(172.16.0.2/24 4c:5e:0c:10:fa:ef);

dev_fake_in -> c;
c[0] -> ar -> dev_fake_out;
c[1] -> q -> dev_out;

dev_in -> dev_fake_out;
```

Listing A.18: Click-Konfiguration Token Passing (WifiToken_kernel.click)

```
dev_fake_in :: FromHost(fake0, PREFIX 172.16.0.1/24, ETHER ath0);
dev_fake_out :: ToHost(fake0)
dev_in :: FromDevice(ath0)
dev_out :: ToDevice (ath0)

q_pre :: Queue
q_post :: Queue

tctl :: TokenController(PEER 4c-5e-0c-10-fa-82, MAXDATA 20);
tcl :: Classifier(12/08FF, -);

c :: Classifier(12/0806 20/0001, -);
ar :: ARPResponder(172.16.0.2/24 4c:5e:0c:10:fa:82);

dev_fake_in -> c;
c[0] -> ar -> dev_fake_out;
c[1] -> q_pre -> [0]tctl -> q_post -> dev_out;

dev_in -> tcl[1] -> dev_fake_out;
tcl[0] -> [1]tctl;
```

A.2.4 Interface-Initialisierung

Listing A.19: Script: Anpassen der CRDA-Datenbank (fixreg.sh)

```
#!/bin/sh
```

```
# Create operating directory
mkdir ~/crda && cd ~/crda

# Fetch prerequisites
apt-get install python-m2crypto libgcrypt11 libgcrypt11-dev libnl-dev openssl

# Fetch Wireless-Regdb source code from Kernel.org
wget http://wireless.kernel.org/download/wireless-regdb/debs/wireless-regdb_2009
    .11.25-1.tar.gz

# Untar wireless-regdb source code
tar -xzf wireless-regdb_2009.11.25-1.tar.gz && cd wireless-regdb/

# fix regdb, add regulatory domain "WB" for WiBACK purposes
echo "country␣WB:
(2402␣-␣2482␣@␣40),␣(N/A,␣30)
(5170␣-␣5835␣@␣40),␣(N/A,␣30)" >> db.txt

# compile & install
make && make-install

# change to operating direcory again
cd ~/crda

# Fetch CRDA source code
wget http://wireless.kernel.org/download/crda/crda-latest.tar.bz2

# untar source code
tar -xjf crda-latest.tar.bz2 && cd crda-1.1.3

# copy self-signed certificates
cp ../wireless-regdb/*.pem pubkeys/

# compile && install
make && make install

# set regulatory domain to WiBACK and show results
iw reg set WB && iw reg get
```

Listing A.20: Konfiguratiosscript: Herunterfahren der Schnittstellen (interfaces_cleanup.sh)

```
#!/bin/bash
echo "Unloading␣CLICK␣config␣..."
click-uninstall
sleep 2
echo
echo

echo "Bringing␣down␣interfaces␣..."
ifconfig ath0 0.0.0.0 > /dev/null 2>&1
ifconfig ath1 0.0.0.0 > /dev/null 2>&1
ifconfig ath0 down > /dev/null 2>&1
ifconfig ath1 down > /dev/null 2>&1
ifconfig ath0raw down > /dev/null 2>&1
ifconfig ath1raw down > /dev/null 2>&1
sleep 1
echo "."
iw ath0raw del > /dev/null 2>&1
iw ath1raw del > /dev/null 2>&1
sleep 1
echo "."
sleep 1
echo "Down!"
echo
```

Listing A.21: Konfiguratiosscript: IBSS ohne Click (interfaces_nothing.sh)

```
#!/bin/bash
echo "ERASING!!!!!"
/root/config/interfaces_cleanup.sh

echo "Set␣regulatory␣Domain"
iw reg set WB && iw reg get
```

```
sleep 1
echo
echo

echo "Configure␣interface␣ath0␣for␣IBSS␣..."
iw ath0 set type ibss

echo "Bring␣up␣interface␣ath0␣..."
ifconfig ath0 up
sleep 2

echo "Joining␣IBSS:␣upstream␣..."
# Fuer 40 MHz-Kanal
iw ath0 ibss join upstream 5745 HT40+
# Fuer 20 MHz-Kanal
#iw ath0 ibss join upstream 5745 HT20
# Fuer 11a-Mode
#iw ath0 ibss join upstream 5745 NOHT

while [ "`iw␣ath0␣link`" == "Not␣connected." ]
do
sleep 1
echo .
done
iw ath0 link
echo
echo


echo "Creating␣Monitor␣interface␣..."
iw phy0 interface add ath0raw type monitor flags none
sleep 2
echo "."
ifconfig ath0raw up



echo "Interfaces␣all␣set␣up␣and␣ready␣for␣work!"
echo
echo

echo "Setting␣network␣config␣..."
ifconfig ath0 172.16.0.1/24
echo "."
sleep 1
echo "."
sleep 1
echo "."
sleep 1
echo "Done!"
echo
echo "Congratulations,␣your␣node␣is␣up␣and␣ready␣to␣rock␣and␣roll!"
```

Listing A.22: Konfigurationsscript: Click-Passthrough (interfaces_default.sh)

```
#!/bin/bash
echo "ERASING!!!!!"
/root/config/interfaces_cleanup.sh


echo "Set␣regulatory␣Domain"
iw reg set WB && iw reg get
sleep 1
echo
echo

echo "Configure␣interface␣ath0␣for␣IBSS␣..."
iw ath0 set type ibss

echo "Bring␣up␣interface␣ath0␣..."
ifconfig ath0 up
sleep 2

echo "Joining␣IBSS:␣upstream␣..."
```

```
iw ath0 ibss join upstream 5745 HT40+
while [ "`iw␣ath0␣link`" == "Not␣connected." ]
do
sleep 1
echo .
done
iw ath0 link
echo
echo

echo "Creating␣Monitor␣interface␣..."
iw phy0 interface add ath0raw type monitor
sleep 2
echo "."
ifconfig ath0raw up



echo "Interfaces␣all␣set␣up␣and␣ready␣for␣work!"
echo
echo

echo "Installing␣CLICK␣kernel␣config␣..."
click-install /root/config/click/WifiDefault_kernel.click
echo "."
sleep 1
echo "."
sleep 1
echo "."
sleep 1
echo "Done!"
echo
echo "Congratulations,␣your␣node␣is␣up␣and␣ready␣to␣rock␣and␣roll!"
```

Listing A.23: Konfiguratiosscript: Click-FDMA (interfaces_fdma.sh)

```
#!/bin/bash
echo "ERASING!!!!!"
/root/config/interfaces_cleanup.sh

echo "Set␣regulatory␣Domain"
iw reg set WB && iw reg get
sleep 1
echo
echo

echo "Configure␣interface␣ath0␣for␣IBSS␣..."
iw ath0 set type ibss

echo "Bring␣up␣interface␣ath0␣..."
ifconfig ath0 up
sleep 2

echo "Joining␣IBSS:␣upstream␣..."
iw ath0 ibss join upstream 5745 HT20
while [ "`iw␣ath0␣link`" == "Not␣connected." ]
do
sleep 1
echo .
done
iw ath0 link
echo
echo


echo "Configure␣interface␣ath1␣for␣IBSS␣..."
iw ath1 set type ibss

echo "Bring␣up␣interface␣ath1␣..."
ifconfig ath1 up
sleep 2

echo "Joining␣IBSS:␣downstream␣..."
iw ath1 ibss join downstream 5825 HT20
```

```
while [ "`iw␣ath1␣link`" == "Not␣connected." ]
do
sleep 1
echo .
done
iw ath1 link
echo
echo


echo "Creating␣Monitor␣interfaces␣..."
iw phy0 interface add ath0raw type monitor
iw phy1 interface add ath1raw type monitor
sleep 2
echo "."
ifconfig ath0raw up && ifconfig ath1raw up



echo "Interfaces␣all␣set␣up␣and␣ready␣for␣work!"
echo
echo

echo "Installing␣CLICK␣kernel␣config␣..."
click-install /root/config/click/WifiDuplex_kernel.click
echo "."
sleep 1
echo "."
sleep 1
echo "."
sleep 1
echo "Done!"
echo
echo "Congratulations,␣your␣node␣is␣up␣and␣ready␣to␣rock␣and␣roll!"
```

Listing A.24: Konfiguratiosscript: Click-Token Passing (interfaces_token.sh)

```
#!/bin/bash
echo "ERASING!!!!!"
/root/config/interfaces_cleanup.sh

echo "Set␣regulatory␣Domain"
iw reg set WB && iw reg get
sleep 1
echo
echo

echo "Configure␣interface␣ath0␣for␣IBSS␣..."
iw ath0 set type ibss

echo "Bring␣up␣interface␣ath0␣..."
ifconfig ath0 up
sleep 2

echo "Joining␣IBSS:␣upstream␣..."
iw ath0 ibss join upstream 5745 NOHT
while [ "`iw␣ath0␣link`" == "Not␣connected." ]
do
sleep 1
echo .
done
iw ath0 link
echo
echo

echo "Creating␣Monitor␣interface␣..."
iw phy0 interface add ath0raw type monitor
sleep 2
echo "."
ifconfig ath0raw up



echo "Interfaces␣all␣set␣up␣and␣ready␣for␣work!"
```

```
echo
echo

echo "Installing␣CLICK␣kernel␣config␣..."
click-install /root/config/click/WifiToken_kernel.click
echo "."
sleep 1
echo "."
sleep 1
echo "."
sleep 1
echo "Done!"
echo
echo "Congratulations,␣your␣node␣is␣up␣and␣ready␣to␣rock␣and␣roll!"
```

A.2.5 Iperf-Scripte

Listing A.25: Konfiguratiosscript: iperf mit MCS-Erhöhung unidirektional (iperf_client_mcs_half.sh)

```
#!/bin/bash
iperf_bin="/usr/bin/iperf"
iw_bin="sbin/iw"
logfile=$1

ssh root@10.20.111.101 "$iw_bin␣ath0␣set␣bitrates␣mcs-5␣0␣&&␣$iw_bin␣ath1␣set␣
    bitrates␣mcs-5␣0"
$iw_bin ath0 set bitrates mcs-5 0 && $iw_bin ath1 set bitrates mcs-5 0

$iperf_bin -u -c 172.16.0.2 -i 1 -b 65M -t 100 | tee $logfile &

for i in {1..7}
do
        sleep 10
        echo "MCS:␣$i"
        ssh root@10.20.111.101 "$iw_bin␣ath0␣set␣bitrates␣mcs-5␣$i␣&&␣$iw_bin␣ath1␣
            set␣bitrates␣mcs-5␣$i"
        $iw_bin ath0 set bitrates mcs-5 $i && $iw_bin ath1 set bitrates mcs-5 $i
done
```

Listing A.26: Konfiguratiosscript: iperf mit MCS-Erhöhung bidirektional (iperf_client_mcs_full.sh)

```
#!/bin/bash
iperf_bin="/usr/bin/iperf"
iw_bin="sbin/iw"
logfile=$1

ssh root@10.20.111.101 "$iw_bin␣ath0␣set␣bitrates␣mcs-5␣0␣&&␣$iw_bin␣ath1␣set␣
    bitrates␣mcs-5␣0"
$iw_bin ath0 set bitrates mcs-5 0 && $iw_bin ath1 set bitrates mcs-5 0

$iperf_bin -u -c 172.16.0.2 -i 1 -b 65M -t 100 -d | tee $logfile &

for i in {1..7}
do
sleep 10
echo "MCS:␣$i"
        ssh root@10.20.111.101 "$iw_bin␣ath0␣set␣bitrates␣mcs-5␣$i␣&&␣$iw_bin␣ath1␣
            set␣bitrates␣mcs-5␣$i"
        $iw_bin ath0 set bitrates mcs-5 $i && $iw_bin ath1 set bitrates mcs-5 $i
done
```

Listing A.27: Konfiguratiosscript: iperf mit Datenrate-Erhöhung bidirektional (iperf_client_noht_full.sh)

```
#!/bin/bash
iperf_bin="/usr/bin/iperf"
iw_bin="sbin/iw"
logfile=$1

ssh root@10.20.111.101 "iw␣ath0␣set␣bitrates␣legacy-5␣6"
```

```
iw ath0 set bitrates legacy-5 6

$iperf_bin -u -c 172.16.0.2 -i 1 -b 54M -t 100 -d | tee $logfile &

for i in  9 12 18 24 36 48 54
do
        sleep 10
        echo "DATARATE: $i"
        ssh root@10.20.111.101 "iw ath0 set bitrates legacy-5 $i"
        iw ath0 set bitrates legacy-5 $i
done
```

Listing A.28: Konfiguratiosscript: iperf mit Token-Erhöhung bidirektional (iperf_client_token_full_noht.sh)

```
#!/bin/bash
iperf_bin="/usr/bin/iperf"
iw_bin="sbin/iw"
logfile=$1

ssh root@10.20.111.101 "$iw_bin ath0 set bitrates legacy-5 24"
$iw_bin ath0 set bitrates legacy-5 24

$iperf_bin -u -c 172.16.0.2 -i 1 -b 24M -t 100 -d | tee $logfile &

for i in  20 30 40 50
do
        sleep 20
        echo "TOKEN: $i"
        echo $i > /click/tctl/maxdata
done
```