Portierung und Evaluation einer Soft-Routerplattform auf das eingebettete Betriebssystem eCos

Studienarbeit im Fach Informations- und Kommunikationstechnik

von

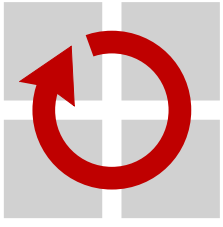
Thomas Hauenstein

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreut durch:

Dipl.-Ing. Martin Hoffmann
Dr.-Ing. Rüdiger Kapitza
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat

Beginn der Arbeit: 1. März 2011
Ende der Arbeit: 30. September 2011



Hiermit versichere ich, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich durch Angabe der Quelle als Entlehnung kenntlich gemacht.

Erlangen, den 30. September 2011 _____

Kurzzusammenfassung

Mit der immer wachsenden Anzahl an vernetzten Geräten steigt auch stetig der Bedarf an eingebetteten Lösungen, die verschiedene Netze trennen und koppeln können. In kleinen Netzen sind Flexibilität und Kosteneffizienz oftmals wichtiger als Hochgeschwindigkeitsverbindungen oder minimale Latenzzeiten. Aus diesem Grund werden Routinglösungen immer häufiger als Software realisiert, die auf gewöhnlicher Hardware lauffähig ist. Das Betriebssystem, das als Verbindung zwischen Anwendung und Hardware fungiert, muss ebenfalls flexibel sein - die aktuelle Entwicklung zeigt, dass CPU-Architekturen wie beispielsweise ARM, die nicht x86-konform sind, auch zunehmend Verwendung finden.

Die vorliegende Arbeit zeigt, wie das Softwarerouter-Projekt *Click* auf das eingebettete Echtzeitbetriebssystem *eCos* portiert wurde. Nach einer Architekturbeschreibung beider Systeme, wird aufgezeigt, wie die Verbindung zwischen Anwendung und Betriebssystem realisiert werden kann. Im Anschluss wird dargestellt, wie der Nutzer mit einem laufenden Router über eine Socket-Verbindung interagieren kann. Die abschließende Evaluation vergleicht die Leistung der Implementierung mit anderen Lösungen auf Basis eines Linux-System und identifiziert mögliche Flaschenhälse des Systems.

Abstract

Due to the ongoing growth of connected devices, embedded solutions capable of connecting and isolating different networks are in great demand. Flexibility and cost efficiency are often more important than high speed connections or minimal latency when referring to small networks. For this reason, routing solutions are more and more implemented in software running on commodity hardware. The operating system acting as a link between application and hardware needs to be flexible as well - current development shows, that non-x86 CPU architectures like ARM are on the rise.

In this thesis we will show how the existing software router project *Click* is adopted to run on the embedded real-time operating system *eCos*. After an explanation of the architecture of both software projects, we will show how the connection between application and operating system can be established. Subsequent, we will introduce a way for the user to interact with a running router via a socket connection. The following evaluation compares the performance of the implementation with other solutions based on a Linux system and identifies possible bottlenecks of the system.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Seit der Erfindung der ersten Computer spielt der Datenaustausch zwischen einzelnen Systemen eine inhärent wichtige Rolle. Weil dieser in der Pionierzeit noch mühsam manuell von menschlichen Operatoren durchgeführt werden musste, wurden schon relativ frühzeitig Versuche unternommen homo- und heterogene Rechensysteme miteinander zu vernetzen. Diese Entwicklung, die spätestens mit der Entwicklung des Arpanets begann, hat sich bis in die heutige Zeit immer weiter fortgesetzt. Längst sind es nicht mehr nur Universitäten, militärische Einrichtungen oder global agierende Firmen, sondern auch Privatpersonen, die über eine Vielzahl von vernetzten Geräten verfügen. Mit diesem Fortschritt steigt auch stetig der Bedarf an Geräten, die verschiedene Netze koppeln oder trennen können. An zentralen Knotenpunkten im Internet, die in Spitzenzeiten mehr als 1 TBit/s an Daten weiterleiten müssen, kommen deshalb spezielle Hardwarelösungen zum Einsatz, die auf das Weiterleiten von Paketen optimiert sind. Die benötigte Rechenleistung wird hier größtenteils von speziellen Netzwerkinterfaces erbracht, die unabhängig von einem zentralen Prozessor arbeiten. Komplett anderes sieht es im Bereich kleinerer Local Area Networks (LAN) aus - hier werden keine teuren auf Durchsatz und Geschwindigkeit optimierten Speziallösungen benötigt, sondern vielmehr kleine, günstige, aber dennoch relativ robuste und vielseitige Geräte gebraucht. Privatpersonen, die über einen DSL-Zugang verfügen, besitzen in aller Regel schon einen Router. Diese eingebetteten Systeme verfügen meist über ein integriertes Modem sowie eine auf Network Adress Translation (NAT) basierende Routingfunktionalität. Leider lassen diese Geräte oftmals tiefergehende Konfigurationsmöglichkeiten vermissen. Auch aus diesem Grund existiert eine Vielzahl an Softwarelösungen, die gewöhnliche PCs um Routingfunktionalität ergänzen, oder sie sogar in vollwertige Router verwandeln können. Zusätzlich zu den traditionellen Routingfunktionen, die unixoide Betriebssysteme wie Linux oder BSD mit

sich bringen, gibt es Softrouter-Anwendung, die auf den Betriebssystemen aufsetzen und so eine einfach zu nutzende und erweiterbare Routingplattform bieten. Die Vorteile solcher Softrouter-Anwendungen lassen sich gerade in Kombination mit beispielsweise auf Kosteneffizienz optimierter Hardware optimal ausnutzen. Gerade im Bereich eingebetteter Systeme spielen alternative Architekturen, wie beispielsweise ARM, neben der dominierenden x86-Architektur, schon heute eine wichtige Rolle. Auf eingebettete Systeme zugeschnittene Betriebssysteme wie **FreeRTOS**[1] oder **eCos**[2] bilden dann das notwendige Bindeglied zwischen der Hardware und der Softrouter-Anwendung. Das Ziel dieser Arbeit ist die Portierung des Softwarerouters Click auf das eingebettete Echtzeitbetriebssystem eCos

1.2 Aufbau dieser Arbeit

Kapitel 2 gibt zunächst einen groben Überblick über Routing und Softrouter im Allgemeinen. Im Anschluss folgt eine detailliertere Betrachtung des Softrouters Click sowie des Echzeitbetriebssystems eCos. Kapitel 3 behandelt die Frage, unter welchen Kriterien Click unter anderen verfügbaren Softroutern ausgewählt wurde. Das darauf folgende Kapitel behandelt die Anbindung des Click-Routers an den Netzwerkstack von eCos. Zudem zeigt es mit dem *ControlSocket*-Element eine Möglichkeit auf, mit einem laufenden Click-Router zu interagieren. Im Anschluss wird die entwickelte Lösung in Kapitel 5 evaluiert und ein Vergleich mit anderen Implementierungen auf Basis eines Linux-Systems angestellt. Den Abschluss bildet das Fazit, das mögliche Ansatzpunkte für zukünftige Verbesserungen aufzeigt.

[1] http://www.freertos.org
[2] http://ecos.sourceware.org

2 Allgemeines

Folgendes Kapitel gibt zunächst einen kurzen Überblick über das Thema Softrouter im Allgemeinen und stellt exemplarisch einige interessante Softrouter-Projekte vor. Im Anschluss findet eine detailliertere Betrachtung des Click-Projekts statt, die die wichtigsten Aspekte des Systems beschreibt. Den Abschluss bildet eine Übersicht über das Echtzeitbetriebssystems eCos. Hier wird neben der grundlegenden Architektur des System auch dessen Konfiguration für verschiedene Anwendungszwecke beschrieben.

2.1 Softrouter

Routing, also eine möglichst optimale Wegwahl zwischen zwei verschiedenen Stationen eines Netzwerks, spielt mit der wachsenden Anzahl an vernetzten Geräten eine immer wichtigere Rolle. Bedingt durch die Heterogenität von Teilnehmern und Netzwerken sind die Anforderungen an eine Routing-Lösung ebenso vielseitig, wie die Gegebenheiten und Einschränkungen die eine vorhandene Infrastruktur mit sich bringt. So stellt die Vernetzung zweier autonomer Systeme (AS), die jeweils von einem Internet Service Provider verwaltet werden, ganz andere Anforderungen an die am Routing beteiligte Hard- und Software sowie an die zugehörigen Routing-Protokolle, als ein lokales Heimnetzwerk.

Im OSI-Referenzmodell ist das Routing einer der Hauptaufgaben der Schicht 3 (Vermittlungsschicht) und entspricht im heute dominierenden TCP/IP-Referenzmodell ebenfalls der Vermittlungsschicht, die hier allerdings in der zweiten Ebene angeordnet ist. Für unsere heutigen IP-basierten Netze bedeutet dies, dass die Wegwahl auf Basis von IP-Adressen getroffen wird. Eine Ausnahme bilden hierbei natürlich diejenigen Mechanismen, die für eine Weiterleitung von Paketen auf Basis von Hardwareadressen sorgen, wie sie beispielsweise in Layer-2-Switches zu finden sind.

Obwohl sich Routingverfahren in vielerlei Hinsicht unterscheiden, lassen sie sich

#	OSI-Schicht	TCP/IP-Schicht	#
7	Anwendung	Anwendung	4
6	Darstellung		
5	Sitzung		
4	Transport	Transport	3
3	Vermittlung	Internet	2
2	Sicherung	Netzzugang	1
1	Bitübertragung		

Abbildung 2.1: OSI- und TCP/IP-Referenzmodell

zunächst in **dynamische** und **statische** Verfahren unterteilen. Dynamische Verfahren kommen v.a. dann zum Einsatz, wenn die Netztopologie sehr umfangreich ist. So sorgt das Border Gateway Protocol (BGP) heute praktisch ausschließlich dafür, dass die autonomen Systeme von Internetprovidern miteinander verbunden werden [1]. Innerhalb eines solchen AS ist Open Shortest Path First (OSPF) das am weitesten verbreitete Protokoll. Aufgrund der Größe solcher Netze und den dabei zu bewältigenden Datenmengen werden Routingaufgaben hier immer von dedizierten Hardwareroutern durchgeführt, die für die Verarbeitung großer Datenmengen und -raten optimiert wurden. Kleine Netzwerke, wie sie im privaten Umfeld oder SoHo-Bereich eingesetzt werden stellen aufgrund ihrer Größe, die meist nur wenige Rechner umfasst, natürlich viel geringere Anforderungen. Dies betrifft sowohl die verwendete Hardware, als auch die verwendeten Routingverfahren - sollten solche überhaupt von Nöten sein. So ist die Netztopologie meist zum Zeitpunkt der Konfiguration bekannt und Änderungen derselben treten höchst selten auf. Darum kann eine Konfiguration hier i.d.R. statisch erfolgen. Abbildung 2.2 zeigt die schematische Darstellung eines solchen Netzwerks. Drei Router verbinden hier drei /24-Netze zu einem größeren Netzwerk. Die viel geringeren Anforderungen machen es möglich, dass hier keine dedizierten Hardwarerouter zum Einsatz kommen, sondern normale PC-Hardware mit entsprechender Software um Routingfähigkeiten erweitert wird. Man spricht dann von sog. Softroutern. Neben den geringeren Kosten haben diese Geräte den großen Vorteil, dass sie sich sehr flexibel verwenden lassen. So ist es nicht unüblich weitere Aufgaben, wie die Bereitstellung von Serverdiensten oder Firewallfunktionalitäten, auf einem Softrouter zu vereinen. Obwohl eine saubere Trennung dieser Komponenten prinzipiell immer wünschenswert ist, wird hier oftmals aus Kostengründen

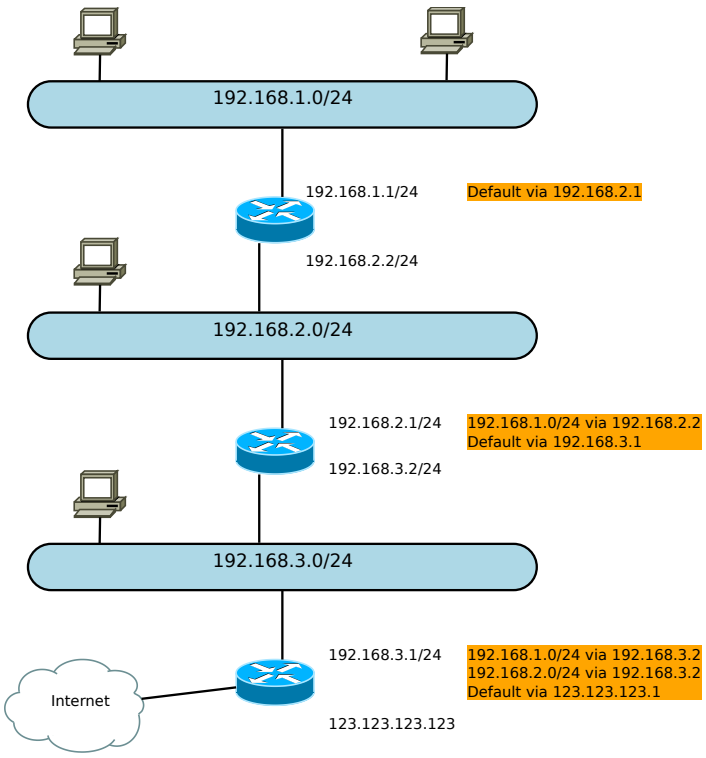


Abbildung 2.2: Statisches Routing in einem lokalen Netzwerk

darauf verzichtet.

Moderne Betriebssysteme können einen Rechner meist schon mit Boardmitteln in einen voll funktionsfähigen Softrouter verwandeln. So verfügt Linux mit den **iproute2-tools**[1] und dem **Netfilter-Projekt**[2] über mächtige Werkzeuge, um auch komplizierteste Netzkonfigurationen zu ermöglichen. Als Nachteil einer solchen Lösung könnte man anführen, dass die Einrichtung je nach der Komplexität der zugrunde liegenden Anforderungen nicht trivial ist und keine zusammenhängende einheitliche Konfigurationssprache existiert. Vielmehr müssen die benötigten Werkzeuge mehrere Male mit unterschiedlichen Parametern hintereinander ausgeführt werden um jeweils einen Teilaspekt des Gesamtkonzepts zu konfigurieren. Sollten ferner Anforderungen existieren, die eine Erweiterung des vorhanden Codes verlangen, muss diese Erweiterung als neues Kernelmodul erstellt werden, was nicht immer praktikabel ist. Aus diesen Gründen existieren eine Vielzahl an Softrouterlösungen, die sich allerdings im Hinblick auf ihre Zielsetzung teilwei-

[1] http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2
[2] http://www.netfilter.org/

se stark voneinander unterscheiden. Während klassische Routing-Daemons, wie beispielsweise *gated* darauf bedacht sind, Unterstützung für dynamische Routingprotokolle zu liefern, können modular aufgebaute Anwendungen wie **Click**[3] ein viel breiteres Spektrum an Aufgaben erfüllen.

Neben dem Click-Projekt, das letztendlich als Softrouter-Lösung für eCos genutzt wurde, sollen im Folgenden zwei auch Alternativprojekte vorgestellt werden.

2.1.1 BIRD Internet Routing Daemon

Der BIRD Internet Routing Daemon ist ein an der Karls-Universität in Prag entwickelter, freier Softrouter, der unter der GNU General Public License lizensiert ist. BIRD bietet Unterstützung für dynamische Routingprotokolle wie BGP oder OSPF sowie für statische Routen und versteht sich auf IPv4 und IPv6. Zu bemerken ist hierbei, dass BIRD entweder mit Unterstützung für IPv4 oder für IPv6 kompiliert werden muss. Sollen beide Protokolle parallel unterstützt werden, müssen zwei Instanzen von BIRD laufen - eine für IPv4- und eine für IPv6-Routing. Die Konfiguration erfolgt über eine Konfigurationsdatei, die in einer C-ähnlichen Syntax verfasst wird. Verändert der Anwender die Konfiguration zur Laufzeit kann diese diese ohne einen Neustart des Daemons übernommen werden. BIRD ist in C geschrieben und lässt sich auf Linux- sowie BSD-Systemen verwenden. Durch den modularen Aufbau des Quellcodes, können große Teile des Codes wiederverwendet werden. Lediglich die betriebssystemspezifischen Schnittstellen für die Kommunikation mit dem Netzwerkstack müssen für an das jeweilige System angepasst werden. Eine Analyse des Quelltexts ergab, dass praktisch alle POSIX-Funktionen, die in BIRD Verwendung finden auch unter eCos zur Verfügung stehen. Die wenigen Funktionen, die nicht in der POSIX-API von eCos vorhanden sind, waren an unkritischen Stellen im Quelltext zu finden. So verwendet BIRD beispielsweise **fork**[4] um den Prozess zu "dämonisieren", d.h. ihn von der startenden Shell loszulösen, was unter eCos nicht nötig gewesen wäre. Obwohl die Aussichten auf eine erfolgreiche Portierung damit sehr gut waren, wurde aus zweierlei Gründen am Ende einem anderen Projekt der Vorzug gegeben. Zum einen bietet BIRD nicht die vielfältigen Möglichkeiten, die Click durch seine sehr

[3] http://read.cs.ucla.edu/click/click
[4] man 2 fork

große Auswahl an Modulen bieten kann und zum anderen war es für die spätere Verwendung als Demonstratorobjekt wünschenswert, dass die Routersoftware in C++ programmiert ist.

2.1.2 eXtensible Open Router Platform

Die eXtensible Open Router Platform (XORP) ist eine unter der BSD-Lizenz lizenzierte, offene Routing-Software, die an der University of California, Berkeley entwickelt wird. Ziel des Projekt ist es, Forschern die Möglichkeit zu geben neue Routingprotokolle und -mechanismen unter realen Bedingung zu testen. Bedingt durch die geschlossene Architektur von kommerziell produzierten Routerlösungen wird bei der Analyse neuer Protokolle bisher oftmals auf Ergebnisse zurückgegriffen, die mit Hilfe von Netzwerksimulatoren, wie beispielsweise **ns**[5] gewonnen wurden. Obwohl die Simulatoren wertvolle Dienste leisten, ist es gerade für neue Entwicklungen im Bereich Routing unumgänglich, auch Tests unter realen Bedingungen durchzuführen. XORP lässt sich grob in zwei Schichten einteilen: die

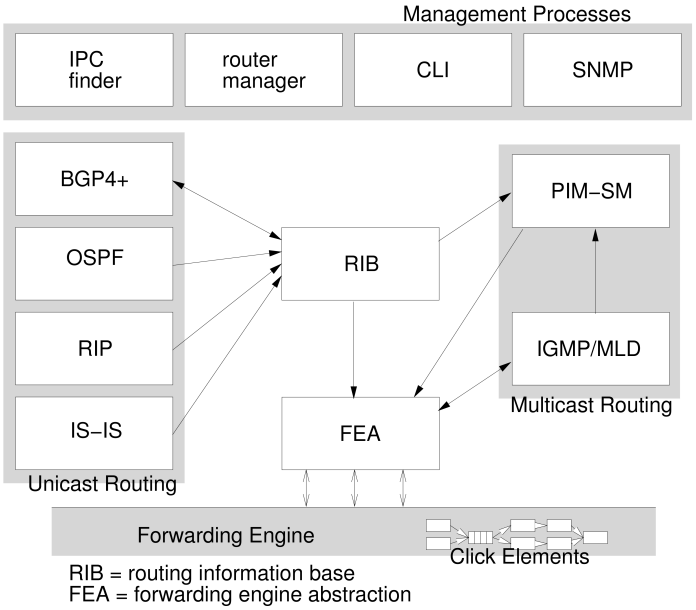


Abbildung 2.3: Architekturüberblick über XORP [2]

obere Schicht (user-level XORP) implementiert die einzelnen Routingprotokolle, während die untere Schicht sich um das eigentliche Weiterleiten der Pakete

[5] http://www.nsnam.org/

kümmert. User-level XORP umfasst mehrere Prozesse - so läuft jedes Routingprotokoll in einem eigenen Prozess. Zusätzlich existieren eigenständige Management- und Konfigurationsprozesse. Die Kommunikation der einzelnen Prozesse erfolgt hierbei über sog. XORP Resource Locators (XRLs), die eine Ähnlichkeit zu URLs aufweisen, wie sie im Internet gebräuchlich sind [2]. Die *Forwarding Abstraction Engine* (FEA) stellt eine Abstraktionsebene zwischen den Routerprozessen und der *Forwarding Plane* dar. Die Art der Forwarding Plane, die sich verantwortlich für den Empfang und das Versenden von Paketen von den Netzwerkschnittstellen zeichnet, ist somit transparent für die Routingprozesse. Als Forwarding Plane kann auf Click oder auf klassische Funktionalitäten im Netzwerkstack des Betriebssystems zurückgegriffen werden. XORP ist sehr modular organisiert und in C++ implementiert. Die Konfiguration erfolgt über eine einheitliche Konfigurationsdatei mit C-ähnlicher Syntax. Die Tatsache, dass XORP weitere Software wie Click als Forwarding Plane nutzt, hat dazu geführt, dass es als Primärziel sinnvoll erschien, den Fokus darauf zu legen, zunächst nur Click als Software unter eCos zu verwenden.

2.2 Click

Click ist ein ursprünglich am Massachusetts Institute of Technology initiiertes Softrouter-Projekt, das zur Zeit hauptsächlich am Institut für Informatik der University of California Los Angeles unter einer MIT/BSD-ähnlichen Lizenz[6] weiterentwickelt wird. Eines der Hauptmerkmale, das Click von den bisher vorgestellten Projekten unterscheidet wird deutlich sichtbar, wenn man die Konfiguration eines Click-Routers betrachtet. So umfasst eine typische Click-Konfiguration nicht nur die Definition von Regeln, die auf Pakete anzuwenden sind, sondern vielmehr zusätzlich den Pfad, den ein Paket auf dem Weg durch die einzelnen Module des Routers nehmen muss. Diese Module, die bei Click schlicht Elemente genannt werden, erfüllen jeweils nur einen kleinen Teilaspekt der Gesamtaufgabe. Soll beispielsweise die *Time To Live* (TTL) eines IP-Pakets dekrementiert werden, muss ein entsprechendes Element in die Konfiguration eingefügt werden. Bedingt durch diese Architektur lässt sich Click sehr flexibel einsetzen und um neue Funktionalitäten erweitern. Eine Konfigurationsdatei beschreibt dabei

[6] http://read.cs.ucla.edu/click/license

einen gerichteten Graphen, dessen Knoten die einzelnen Elemente beschreiben, während die Wege zwischen Elementen von den Kanten des Graphs repräsentiert werden. Click ist in C++ implementiert und auf Linux-, sowie mit Einschränkungen auf BSD-Systemen einsetzbar. Aktuell unterstützt Click drei verschiedene Betriebsarten. Im Userlevel-Modus kann Click Pakete auf zweierlei Art empfangen. Unter Linux-Systemen können *RAW Sockets* verwendet werden um Pakete zu empfangen. Sollte diese Schnittstelle nicht zur Verfügung stehen, kann als Alternative auf die **PCAP Library**[7] zurückgegriffen werden. Auch wenn sich der Userlevel-Modus gut zum Testen von verschiedenen Konfigurationen eignet, ist die erreichbare Leistung architekturbedingt nicht sehr gut. Der Hauptgrund dafür ist, dass die Pakete zunächst große Teile des Netzwerkstacks des jeweiligen Betriebssystems durchlaufen müssen, bis sie am Ende die Click-Anwendung erreichen. Deshalb bietet Click für Linux ein Kernelmodul an, das es ermöglicht der Bearbeitung von Paketen durch das Betriebssystem zuvorzukommen, indem die Pakete dem Netzwerkstack vorenthalten werden. Für FreeBSD existiert ebenfalls ein Kernelmodul, das eine ähnliche Funktionalität ausweist - allerdings ist die Unterstützung durch die Entwickler sehr begrenzt, wodurch ein produktiver Einsatz dieses Moduls, ohne eine vorherige Überarbeitung des Quellcodes nicht möglich ist. Der modulare Aufbau macht es möglich, dass praktisch alle Elemente, die an der Paketverarbeitung beteiligt sind, unabhängig vom jeweiligen Betriebssystem verwendet werden können. Eine Ausnahme bilden hierbei diejenigen Elemente, die mit dem Betriebssystem interagieren, um beispielsweise Pakete direkt in den Netzwerkstack zu injizieren.

2.2.1 Architektur

2.2.1.1 Elemente

Die Konfiguration eine Click-Routers ist ein gerichteter Graph, dessen Knoten Elemente genannt werden [3]. Jedes Element erfüllt dabei einen Teilaspekt der Gesamtaufgabe, wie beispielsweise das Dekrementieren der TTL eines IP-Pakets. Die immer unidirektional verlaufenden Verbindungen zwischen den Knoten stellen den möglichen Weg eines Pakets zwischen zwei Elementen dar. Intern ist jedes aktive Element die Instanz einer C++-Klasse, die von einer Basisklasse *Element*

[7] http://www.tcpdump.org

abgeleitet werden muss. Die Kanten zwischen Elementen werden durch Zeiger auf die Element-Instanzen realisiert. Dafür verfügen die Elemente über sog. *Ports*. Je nach Typ hat ein Element beliebig viele Eingangs- und Ausgangsports, wobei jeder dieser Ports eine Verbindung mit einem Port eines anderen Elements herstellt. Da viele Elemente eine vom Benutzer vorgegebene Konfiguration benötigen, kann

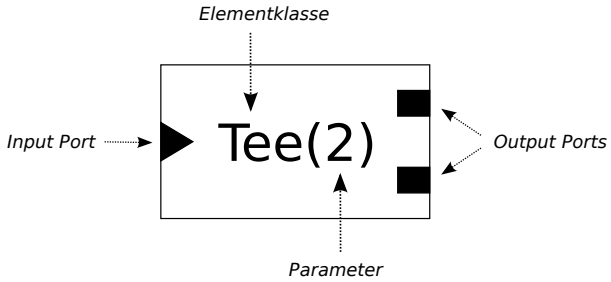


Abbildung 2.4: Beispiel des Click-Elements *Tee*[3]

jeder neuen Elementinstanz bei der Instantiierung ein optionaler Konfigurationsparameter als String übergeben werden. Dieser wird dann, je nach Inhalt, von einem einfachen Parser in beliebig viele einzelne Parameter unterteilt. Abbildung 2.4 zeigt die Elementklasse *Tee*, deren Aufgabe darin besteht, ein ankommendes Paket zu kopieren und an n Ausgangsports weiterzuschicken. Eingangsports werden als ein Dreieck und Ausgangsports als ein Viereck dargestellt. Folglich verfügt *Tee* über einen Eingangsport und hier über zwei Ausgangsports, wobei der Parameter *2* vom Benutzer konfiguriert wird. Um verschiedenen Anforderungen gerecht zu werden, existieren verschieden Arten von Ports - *push*, *pull* und *agnostic*. Eine genauere Betrachtung findet sich im folgenden Abschnitt (2.2.1.2). Elemente können sog. *Handler*, also Schnittstellen nach außen anbieten. So lassen sich nicht nur Informationen von den Elementen im laufenden Betrieb des Routers erhalten, sondern es können auch geringfügige Änderungen am Status eines Elements durchgeführt werden. Elementspezifische Handler sind immer in den Elementklassen definiert, nur Standard-Handler, über die sich beispielsweise der Klassenname eine Elements abfragen lässt, werden automatisch erzeugt. Handler können verschiedene Ausprägungen haben, die beiden wichtigsten sind *Read* und *Write*. Ein typischer Read-Handler könnte die Anzahl der vom Element bearbeiteten Pakete zurückliefern, während ein Write-Handler dazu dienen könnte, diesen Paketzähler zu manipulieren.

2.2.1.2 Ports

Die unidirektionalen Verbindungen zwischen Elementen in Click bestimmen mögliche Wege, die ein Paket im Router durchlaufen kann. Da jedes Element eine andere Aufgabe erfüllt, erscheint es sinnvoll beim Entwurf darauf zu achten, welcher Art die Verbindungen eines Elements sind. So kann ein Element, das die Prüfsumme eine IP-Pakets erstellt, zu jedem Zeitpunkt aufgerufen werden um seine Berechnung durchzuführen, während ein Element, das Pakete an eine Netzwerkschnittstelle weiterreicht ggf. warten muss, bis das Gerät bereit zum Versenden neuer Daten ist. Deshalb können die Verbindungen in Click zwei Ausprägungen haben: *push* oder *pull*. Bei push-Verbindungen startet ein Quellelement den Datenaustausch und reicht ein Paket an das nachfolgende Element weiter - pull-Verbindungen funktionieren entgegengesetzt - hier fordert die Paketsenke von der Paketquelle ein Paket an. Ob eine Verbindung vom Typ push oder pull ist, wird durch die Ports eine Elements bestimmt. Jeder Port eines Elements wird durch eine Objekt vom Typ *Element::Port* repräsentiert und weiß jeweils mit welchem Port eines anderen Elements er verbunden ist. Für die Verbindungen existieren einige Regeln, deren Nichtbeachtung dazu führt, dass die Initialisierung eines Routers nicht stattfinden kann:

- Ports sind vom Typ *push*, *pull* oder *agnostic*. agnostic bedeutet, dass ein Port entweder den Status *push* oder *pull* einnehmen kann.

- Sollen zwei Ports verbunden sein, müssen sie vom gleichen Typ sein, d.h. ein push-Port darf nur mit einem push-Port verbunden werden und ein pull-Port dementsprechend nur mit einem andern pull-Port. Eine Ausnahme bilden agnostic-Ports - diese dürfen mit einem push-, pull- oder agnostic-Port verbunden werden. Verbindungen zwischen push- und pull-Ports sind verboten.

- Verbindungen nehmen je nach Art der beteiligten Ports immer exklusiv den Status push oder pull an. Dies gilt auch, wenn agnostic-Ports an der Verbindung beteiligt sind.

- Jeder Port darf nur mit **einem** andern Port verbunden werden, d.h. eine Verbindung besteht immer genau zwischen zwei Ports.

Intern werden die Verbindungen durch virtuelle Funktionsaufrufe realisiert, die Zeiger auf Objekte vom Typ *Packet* zwischen den Elementen weiterreichen.

2.2.1.3 Kontrollfluss

Click verfügt intern über einen Scheduler für die Elemente, der allerdings nicht verdrängend arbeiten kann und somit auf die Kooperation der Elemente angewiesen ist. So ist es dem Scheduler beispielsweise unmöglich eine Paketweitergabe zu unterbrechen, die nur durch push-Aufrufe realisiert wird. Allerdings ist eine solche Vorgehensweise wenig praktikabel, da somit für jedes Paket zunächst die gesamte Elementkette durchlaufen werden müsste, bis das nächste Paket verarbeitet werden kann. Gerade bei langen Elementketten oder hohen Paketraten würde das zu einem großen Paketverlust führen. Darum gibt es in Click explizite Speicherelemente, die eine solche push-Aufrufkette unterbrechen indem sie empfangene Pakete speichern, statt sie an das Nachfolgeelement weiterzuleiten. Das gibt dem Scheduler die Möglichkeit ein anderes Element einzulasten. Abbildung 2.5 zeigt

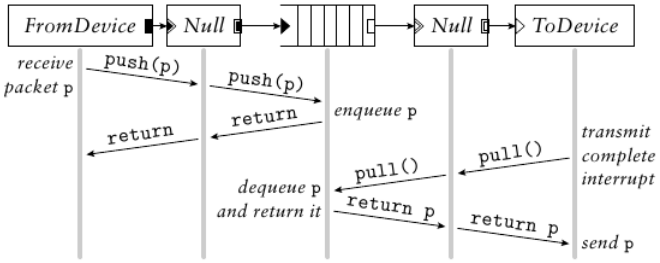


Abbildung 2.5: Kontrollfluss einer Elementkette [3]

hierfür ein Beispiel. *FromDevice* empfängt Pakete von der Netzwerkschnittstelle. Diese werden mit push-Aufrufen bis zum Speicherelement in der Mitte weitergeleitet. Danach kehren alle push-Aufrufe zurück und ein neues Element - hier *ToDevice* wird vom Scheduler ausgewählt. Dieses initiiert nun durch einen pull-Aufruf ebenfalls eine Aufrufkette, die diesmal allerdings rückwärts verläuft, bis sie am Speicherelement angekommen ist. Hier wird ein vorher gespeichertes Paket aus der Warteschlange entfernt und nach oben zurückgeliefert. Zu bemerken ist, dass Elemente, die nur push-, oder pull-Methoden implementieren, nicht vom Click-Scheduler kontrolliert werden. Lediglich Elemente, die eine *run_task*-Methode

anbieten können somit explizit gescheduled werden. In unserem Beispiel wären dies die Elemente *FromDevice* und *ToDevice*. Diese führen ein implizites Scheduling der Nachfolge- und Vorgängerelemente durch, indem sie in ihrer *run_task*-Methode push- bzw. pull-Aufrufe tätigen. Die bedeutet auch, dass immer nur ein Task aktiv ist und der Scheduler auf das kooperative Verhalten dieser Tasks angewiesen ist. So kann ein Element, dass sich nicht kooperativ verhält den gesamten Kontrollfluss stoppen und den Router zum Erliegen bringen.

2.2.1.4 Pakete

Click verwendet eine eigene Paketklasse, die je nach Betriebsmodus ein anderes Verhalten zeigt. Da für die Anwendung unter eCos der BSD-Modus weiter von Bedeutung ist, wird die Betrachtung auf diesen beschränkt. Jedes Netzwerkpaket wird innerhalb eine Click-Routers durch eine Instanz der C++-Klasse *Packet* repräsentiert. Diese beinhaltet als wichtigstes Element einen Datenzeiger auf eine mbuf-Struktur, wie sie in Abschnitt 4.1.4 beschrieben ist. Durch diese Kapselung ist ein Paket in Click für gewöhnlich nur einige Bytes groß, da auf die eigentlichen Paketdaten nur verwiesen wird. Dies verhindert unnötiges Kopieren und steigert die Performance. Eine Besonderheit ist das *Data-Sharing-Konzept* von Paketen. Beim Kopieren eines Pakets wird zunächst nur eine neue *Packet*-Instanz erzeugt, deren Datenzeiger weiterhin auf die gleiche mbuf-Struktur zeigt, wie der des Originalpakets. Zusätzlich wird der Referenzzähler im Paket erhöht. Durch diese Methode wird es erst nötig eine Kopie der Paketdaten anzufertigen, wenn die Paketdaten von mehr als einer *Packet*-Instanz referenziert werden und schreibend auf die Daten zugegriffen wird. Zusätzlich enthält die *Packet*-Klasse weitere Informationen, wie beispielsweise Zeiger, die direkt auf die Netzwerk- oder Transport-Header des Pakets verweisen. Außerdem werden sog. *Packet Annotations* sowie ein Zeitstempel im Paket gespeichert.

2.2.2 Konfiguration

Click verfügt über eine einfache, relativ leicht zu erlernende Konfigurationssprache, deren Hauptmerkmale im Folgenden kurz erläutert werden sollen. Die zwei wichtigsten Bestandteile einer Click-Konfiguration sind *Deklaration* und *Verbindungen*. Deklarationen legen fest, dass ein Element erstellt werden soll, während

Verbindungen definieren, wie die Elemente miteinander verbunden werden. Auflistung 2.1 zeigt eine einfache Konfiguration, die die beiden Bestandteile veranschaulichen soll.

```
/* Deklarationen */
fd :: FromDevice(eth0);
print :: Print();
discard :: Discard;

/* Verbindungen*/
fd[0] -> [0]print;
print[0] -> [0]discard;
```

Listing 2.1: Beispielkonfiguration

Diese simple Konfiguration empfängt Pakete von der Netzwerkschnittstelle *eth0*, gibt den Paketinhalt auf dem Standardausgabekanal aus und verwirft die Paket anschließend wieder. Die hier in Kleinbuchstaben geschriebenen Bezeichner vor den Doppelpunkten sind Namen für die jeweiligen darauf folgenden Elemente. *eth0* ist ein Konfigurationsparameter, der dem Element *FromDevice* bei seiner Initialisierung mit übergeben wird. Die Elemente *Print* und *Discard* brauchen hier keine solchen Konfigurationsparameter - darum können die Klammern wie bei *Print* leer, oder wie bei *Discard* gleich ganz weggelassen werden. Bei der Definition der Verbindungen bezeichnen die in eckigen Klammern eingeschlossenen nummerischen Werte jeweils die Ports der Elemente, die miteinander verbunden werden sollen. Stehen die eckigen Klammern vor dem Bezeichner, handelt es sich um die Eingangsports eines Elements, stehen sich nach dem Bezeichner, handelt es sich folglich um die Ausgangsports. Kommentare lassen sich wie in C++ definieren. Einzeilige Kommentare werden mit // eingeleitet, sich über mehrere Zeilen erstreckende Kommentare müssen von /* */ umschlossen werden. Um die Konfigurationen für den Benutzer zu vereinfachen existieren einige Regeln, die es ermöglichen die Beispielkonfiguration aus Auflistung 2.1 auch wie folgt zu schreiben.

```
FromDevice(eth0) -> Print -> Discard;
```

Listing 2.2: Vereinfachte Beispielkonfiguration

Die Deklarationen geschehen hier implizit indem nur die Klassennamen der gewünschten Elemente angegeben werden. Im laufenden Betrieb erhalten die Elementinstanzen allerdings einen eindeutigen Bezeichner, der sie von anderen Elementen des gleichen Typs unterscheidbar macht. Der Aufbau folgt dabei dem Muster *Klassenname@[0-9]+*, d.h. hier beispielsweise *FromDevice@1*. Weiter ist zu bemerken, dass beim Verbinden der Elemente auf die Angabe der Ports verzichtet wird - in diesem Fall wird die Annahme getroffen, dass Port 0 gemeint ist. Auffällig ist ebenfalls, dass die Deklaration mitten in der Verbindung der drei Elemente geschieht, allerdings ist es gerade bei umfangreichen Konfigurationen sinnvoll, Deklarationen und Verbindungen zu Gunsten einer besseren Lesbarkeit sauber zu trennen. Neben den eben vorgestellten Kernelementen der Konfigurationssprache gibt es weitere syntaktische Regeln, die das Erstellen von Konfigurationen für den Benutzer vereinfachen sollen. Erwähnenswert sind hier noch die Verbundelemente (*Compound Elements*), während weitere Sprachelemente der ausführlichen Dokumentation[8] entnommen werden können. Verbundelemente sind eine Gruppierung von Elementen, die sich nach außen wie ein einziges Element verhalten. Syntaktisch ist ein Verbundelement eine den Regeln für die Konfiguration folgende Kollektion von Elementen, die in geschweiften Klammern eingeschlossen sind. Innerhalb der Klammern existieren zwei spezielle Bezeichner, *input* und *output*, die für die Verbindungen des Verbundelements nach außen hin zuständig sind. Verbundelemente lassen sich schachteln und innerhalb einer Click-Konfiguration wie ein normale Elemente verwenden. Auflistung 2.3 zeigt eine Verbundkonfiguration, die äquivalent zu der Konfiguration aus Auflistung 2.4 ist.

```
compound :: {
  input -> X -> output;
  input [1] -> Y -> [1] output;
};
a -> compound -> b;
c ->[1] compound [1] -> d;
```

Listing 2.3: Verbundkonfiguration

```
a -> X ->b;
c -> Y-> d;
```

[8] http://read.cs.ucla.edu/click/docs/language

Listing 2.4: Äquivalent zur Verbundkonfiguration

Verbundelemente haben auf die Laufzeit des Routers keine Nachteile, da sie nur in der Konfiguration bestehen. Bei der Initialisierung eines Click-Routers werden die Verbundelemente aufgelöst und die darin enthaltenen Klassen ganz normal instantiiert.

2.3 eCos

eCos ist ein Echtzeitbetriebssystem, das unter der **eCos License**[9], einer modifizierten Version der **GNU General Public License** lizensiert ist. eCos bietet Unterstützung für eine immer wachsende Anzahl an Prozessorarchitekturen wie beispielsweise x86, ARM oder SPARC. Die große Stärke von eCos ist der hohe Grad an Konfigurierbarkeit. Diese erlaubt es Konfigurationen für Systeme mit minimaler Hardwareausstattung zu erstellen, ermöglicht es aber auch eCos auf Zielsystemen mit leistungsfähiger Hardware zu nutzen.

2.3.1 Von der Konfiguration zur Anwendung

Durch das mitgelieferte Konfigurationssystem lässt sich eCos flexibel an verschiedene Anwendungsfälle anpassen. So ist es möglich sehr schlanke Konfigurationen zu erstellen, die nicht benötigte Funktionen deaktivieren. Sollte die spätere Anwendung beispielsweise keine aktiven Netzwerkverbindungen benötigen, kann die gesamte Netzwerkunterstützung bei der Konfiguration deaktiviert werden, was zu einer Verkleinerung des Ressourcenbedarfs führt. Neben der Aus-, bzw. Abwahl bestimmter Komponenten lassen sich einzelne Teilaspekte des Betriebssystem auch separat umkonfigurieren. Dies erlaubt es die Größe der zu verwendenden Netzwerkpuffer ebenso zu konfigurieren, wie die Art des eingesetzten Schedulers. Um ein eCos-System zu bauen muss zunächst eine Konfigurationsdatei mit dem mitgelieferten Konfigurationswerkzeug erstellt werden. Hier lassen sich verschiedene *Templates* als Grundgerüst auswählen. Durch ein solches Template wird eine Auswahl an Parametern für das jeweilige Zielsystem getroffen. Verfügt

[9] http://ecos.sourceware.org/license-overview.html

die Hardware über Netzwerkkarten des Typs E1000 kann ein passendes Template gewählt werden, das die entsprechenden Treiber enthält. Nachdem diese Grundauswahl getroffen ist, können weitere Änderungen an der Konfiguration vorgenommen werden. Hier lassen sich je nach späterem Anwendungszweck bestimmte Systemfunktionen (de)aktivieren, oder Standardparameter für einzelne Komponenten - beispielsweise die IP-Adressen der Netzwerkschnittstellen - festlegen. Nachdem der Konfigurationsprozess abgeschlossen ist, kann das System mit der eCos-Toolchain[10] gebaut werden. Dies bedeutet, dass zunächst nur die eCos-Bibliothek erstellt wird - die später zu verwendende Anwendung ist noch nicht beteiligt. Die durch den Konfigurationsprozess gewählten Optionen werden mit Hilfe des C-Präprozessors in den Quelltext integriert. Nach der Konfiguration werden hierfür entsprechende Header erstellt, die in den Quelldateien inkludiert werden. Nicht benötigte Komponenten werden somit schon vor der Kompilierung aus dem Quellcode entfernt - man spricht hier dann von sog. *compile-time control* [4]. Wäre im Beispiel aus Auflistung 2.5 die Option *INCLUDE_PRINTF_DEBUGGING* im Konfigurationswerkzeug deaktiviert worden, würde der entsprechende Header kein dementsprechendes Symbol definieren. Der Aufruf von *printf* würde nicht mit kompiliert werden, was einerseits Auswirkungen auf das spätere Verhalten der Funktion und andererseits auf die daraus resultierende Codegröße hätte.

```
function foo(){
  ...
  #ifdef INCLUDE_PRINTF_DEBUGGING
  printf ("__FUNCTION__ = %s\n", __FUNCTION__);
  #endif
  ...
}
```

Listing 2.5: Beispiel für compile-time control

Neben dieser sehr feingranularen Konfigurationsmöglichkeit kommt noch eine grobgranularere Methode zum Einsatz. Durch *selective linking*, wie es beispielsweise der GNU linker *ld* unterstützt, können nicht referenzierte Funktionen komplett aus dem Code entfernt werden. Während des eCos-Buildprozesses kommen beide Methoden zur Anwendung.

[10] umfasst u.a. GNU binutils, GNU gcc und GNU g++ → http://ecos.sourceware.org/build-toolchain.html

Nachdem die eCos-Bibliothek von der Toolchain erstellt worden ist kann die eigentliche Anwendung kompiliert und assembliert werden. In einem letzten Schritt werden dann die Anwendung sowie die eCos-Bibliothek zusammen mit dem benötigten Startup-Code in das finale Programm gebunden. Dieses lässt sich dann auf der Zielplattform beispielsweise mit Hilfe eine Bootloaders wie **grub**[11] starten.

2.3.2 Architektur

Die Architektur von eCos baut auf verschiedenen Paketen wie dem Hardware Abstraction Layer (HAL) oder dem Kernel auf. Jedes dieser Pakete ist in mehrere Komponenten aufgeteilt, welche wiederum verschiedene Konfigurationsoptionen bieten. Innerhalb von eCos wäre *eCos Kernel* ein Paket, *Kernel schedulers* eine Komponente des Pakets und *Multilevel queue scheduler* bzw. *Bitmap scheduler* Konfigurationsoptionen. Diese "Baueinheiten" werden schichtweise angeordnet und ergeben am Ende ein lauffähiges Gesamtsystem. Abbildung 2.6 zeigt ein Beispiel wie die einzelnen Elemente übereinander gelagert ein lauffähiges Gesamtsystem ergeben. Diese Schichtenarchitektur erlaubt es hier im Beispiel die Komponente *Web server* zu entfernen, ohne dass dies Auswirkungen auf die darunter liegenden Schichten hätte.

Innerhalb eines eCos-Entwicklungsverzeichnisses befindet sich unterhalb des Verzeichnisses *packages* eine Datei *ecos.db* die als Datenbank für die Konfigurationswerkzeuge dient und Einträge für alle Pakete und Komponenten enthält. Im gleichen Verzeichnis befindet sich zusätzlich eine hierarchisch gegliederte Verzeichnisstruktur mit Ordnern für die Pakete und Komponenten. Diese Ordner enthalten dann neben den eigentlich Quelltextdateien für jede Komponente eine oder mehrere Dateien, die die Konfigurationsoptionen enthalten. Dies erlaubt es dem Entwickler, Erweiterungen der Codebasis wie beispielsweise neue Gerätetreiber ohne großen Aufwand in das System zu integrieren.

Dieser modulare Aufbau spiegelt sich auch in den bereits vorhanden Komponenten von eCos wieder. Exemplarisch sei hier die Netzwerkunterstützung des Systems genannt. So stehen zwei alternative Netzwerkstacks in eCos zur Verfügung. Einfache Systeme, die nur über wenig Hauptspeicher verfügen, aber dennoch nicht auf Netzwerkkonnektivität verzichten wollen, können auf den **lwIP**[12]-

[11] http://www.gnu.org/software/grub/
[12] http://www.sics.se/~adam/lwip

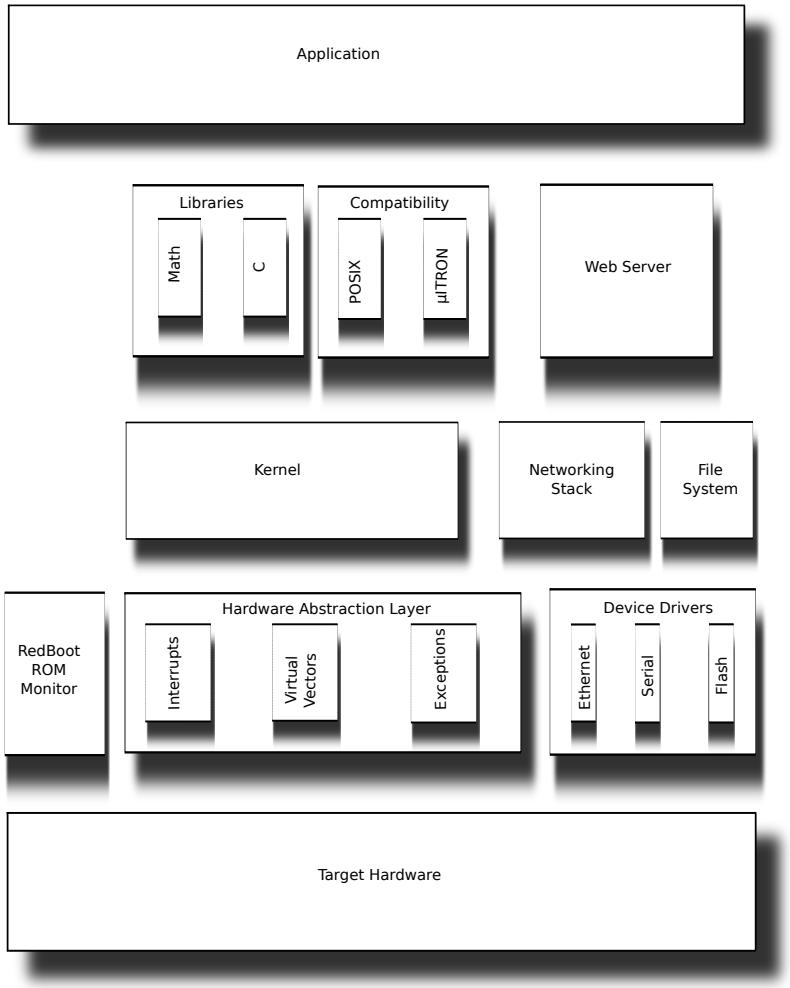


Abbildung 2.6: Beispiel für das Zusammenspiel verschiedener eCos-Pakete [4]

Stack zurückgreifen. Sollen die Anforderungen an die Netzwerkfähigkeiten wachsen, kann alternativ die Portierung eines FreeBSD-Netzwerkstacks verwendet werden.

2.4 Zusammenfassung

Nach einem Überblick über Routing und Softrouter im Allgemeinen, wurde das Click-Projekt in diesem Kapitel näher betrachtet. Dabei wurden mit den Elementen, Ports, Paketen sowie dem Kontrollfluss die wichtigsten Komponenten und Konzepte des Systems erklärt. Im Anschluss folgte eine grundlegende Beschrei-

bung der Konfigurationssprache. Neben Click ist das Echtzeitbetriebssystem eCos de zweite wichtige Grundlage für diese Arbeit. So wurde hier erläutert, welche Schritte notwendig sind, um von einer eCos-Konfiguration zu einem lauffähigem Gesamtsystem zu gelangen. Den Abschluss bildete ein Überblick über den modularen Aufbau der eCos-Architektur.

3 Konzept

In diesem Kapitel soll erläutert werden, weshalb das Click-Projekt am geeignetsten für den Einsatz unter eCos erschien. Die Projekte wurden im Hinblick auf ihre POSIX-Kompatibilität, ihre Unterstützung für BSD-basierte Netzwerkstacks sowie auf ihre Modularität und dadurch bedingte Flexibilität untersucht. Im Anschluss findet eine Beschreibung der verwendeten Hardware sowie der Virtualisierungslösungen statt.

3.1 Auswahl der Routersoftware

Obwohl die Anzahl der Projekte im Bereich der Softrouter relativ groß ist, zeichnete es sich schnell ab, dass viele Projekte für den späteren Verwendungszweck nicht geeignet waren. Um näher in Betracht gezogen zu werden, gab es einige Bedingungen, die von den Projekten idealerweise erfüllt werden sollten:

- Die Codebasis sollte POSIX-konform sein.
- Verwendung möglichst nur solcher POSIX-Funktionen, die auch in eCos verfügbar sind.
- Modular aufgebauter, in C++ geschriebener Quellcode.
- Unterstützung für statisches Routing.
- Support für die BSD-Betriebssystemfamilie, im Idealfall für FreeBSD.

Die Vorauswahl der Software fiel auf Click, XORP und bird, wobei keine der Anwendungen alle o.g. Punkte vollständig erfüllen konnte. Im Folgenden soll nun geklärt werden warum Click aufgrund der o.g. Kriterien gewählt wurde.

3.1.1 POSIX

Das Portable Operating System Interface (POSIX) ist eine standardisierte Programmierschnittstelle, die eine Schnittstelle zwischen Applikation und Betriebssystem zur Verfügung stellt. Damit lassen sich Anwendungen die sich an den POSIX-Standard halten meist ohne viel Aufwand auf andere Betriebssysteme portieren, die ebenfalls den POSIX-Standard unterstützen [5]. Mit dem *POSIX Compatibility Layer*[1] bietet eCos Unterstützung für die POSIX-Spezifikation nach *ISO/IEC 9945-1*. Dies umfasst verschiedene Funktionen u.a. aus den Bereichen Threads, Synchronisation und I/O. Der Dokumentation[2] kann der aktuelle Status des POSIX-Supports entnommen werden - hier finden sich nach Bereichen aufgegliedert Informationen welche Funktionen unter eCos zur Verfügung stehen, welche nicht und ob es Einschränkungen bei Verwendung gibt. Da es für die spätere Funktionalität des Softrouters wichtig zu wissen war, inwiefern die Anwendungen Gebrauch von nicht unterstützten POSIX-Funktionen machen wurde der Quellcode daraufhin untersucht.

Die in Abbildung 3.1 aufgelisteten Funktionen werden laut Dokumentation nicht von eCos unterstützt, teilweise aber dennoch von den drei Projekten genutzt. Diese Liste wurde automatisiert aus den Quellcodedateien erstellt und ist daher nur bedingt aussagekräftig. So muss in Betracht gezogen werden, dass Teile des Quellcodes durch Präprozessoranweisungen vor dem Kompilieren ggf. aus den Quelldateien entfernt werden. Ebenso sind Aufrufe in Modulen, die keine Kernfunktionalität der Anwendung zur Verfügung stellen nur von minderer Bedeutung, da diese Module ggf. einfach weggelassen werden können - ein Modul das Paketdaten in eine Datei schreibt macht wenig Sinn, wenn kein Dateisystem vorhanden ist. Diese Gründe machten eine manuelle Analyse der Fundstellen notwendig. Als Ergebnis zeigte sich, dass bird hier mit Abstand am besten abschnitt. Die Aufrufe der wenigen nicht von eCos unterstützten Funktionen befanden sich an Stellen, die für eine eCos-Anwendung nicht von Belang waren. Click lag bei dieser Betrachtung zwischen bird und XORP, allerdings waren die meisten nicht unterstützten Funktionsaufrufe entweder optional und über einen Konfigurationseintrag deaktivierbar oder in Userlevel-Modulen zu finden. So betrafen viele dieser Aufrufe das Benutzer- und Gruppenmanagement auf Unix-Systemen, was

[1] http://ecos.sourceware.org/docs-latest/ref/posix-compatibility.html
[2] http://ecos.sourceware.org/docs-3.0/ref/posix-standard-support.html

Funktion	Click	Bird	XORP	Funktion	Click	Bird	XORP
alarm			X	munmap	X		
execlp			X	pause			X
execve			X	pipe	X		X
execvp	X		X	setgid	X		
execv			X	setpgid			X
exit			X	setsid		X	X
fchmod			X	setuid	X		
fork	X	X	X	sigaction	X		X
geteuid	X		X	sigaddset	X		X
getegid	X			sigemptyset	X		X
getgid	X		X	sigfillset			X
getpid	X		X	sigismember	X		X
getpgrp	X			siglongjmp			X
getpwnam_r			X	sigprocmask			X
getpwnam	X		X	sigsetjmp			X
getpwuid_r			X	sleep			X
getpwuid			X	tcgetpgrp	X		
getuid			X	umask			X
kill	X		X	waitpid	X		X
mmap	X			wait			X

Abbildung 3.1: Auflistung von POSIX-Funktionen die eCos nicht unterstützt, die aber von den Projekten teilweise dennoch verwendet wurden

für eine Anwendung unter eCos natürlich ignoriert werden konnte.

3.1.2 BSD-Unterstützung

Im Konfigurationsprozess von eCos stehen dem Benutzer zwei verschiedene Netzwerkstacks zur Verfügung. Neben dem leichtgewichtigen **lwIP**[3]-Netzwerkstack steht noch eine Portierung des FreeBSD-Netzwerkstacks aus dem **KAME**-Projekt[4] zur Verfügung. Obwohl die Version des FreeBSD-Stacks laut Dokumentation aktuell ist, hat sich während einer näheren Betrachtung gezeigt, dass es sich tatsächlich um eine ältere Version aus dem Jahr 2002 handelt. Da ein Softrouter für eine möglichst effiziente Verarbeitung von Paketen direkt mit dem Stack interagieren soll, d.h. ohne Umwege über beispielsweise RAW-Sockets, wie sie im Userlevelm-

[3] http://www.sics.se/~adam/lwip/
[4] http://www.kame.net/

odus von Click zum Einsatz kommen, war es wünschenswert, dass bereits eine Anbindung an einen BSD-Netzwerkstack im jeweiligen Softrouter-Projekt vorgesehen ist. Da diese Anbindung v.a. die Forwarding Engine betrifft, wurde XORP hier von der Betrachtung ausgeschlossen. Sowohl für bird als auch für Click existieren bereits BSD-Module. Die BSD-Module von Click wurden allerdings lange Zeit nicht gepflegt, was dazu geführt hat, dass sie unter aktuellen BSD-Versionen nicht lauffähig sind, da das *netisr-Framwork* in FreeBSD 8.0 neu implementiert wurde [5]. Da der BSD-Stack von eCos allerdings nicht dem eines aktuellen BSD-Systems entspricht, war dieser Umstand nicht unbedingt als Nachteil anzusehen.

3.1.3 Modularität

Die Architektur aller drei Projekte ist als modular zu bezeichnen. Dennoch gibt es zwischen den Projekten große Unterschiede im Bezug auf den Begriff Modularität. Der auffälligste Unterschied ist zunächst die verwendete Programmiersprache. Click und XORP sind beide, von einigen wenigen Ausnahmen abgesehen, komplett in C++ implementiert, während bird rein in C geschrieben ist. Dass bird ein *Routing Daemon* ist, der eine ähnlich Zielsetzung wie **Zebra**[6] oder **gated** verfolgt, nämlich die Bereitstellung von dynamischem Routing in produktiven Umgebungen, zeigt sich spätestens bei Betrachtung der Module. Diese sind hier auf die Bereitstellung der wichtigsten, hauptsächlich dynamischen Routingprotokolle beschränkt. Eine Erweiterung, um neben Routing auch andere Anwendungsgebiete wie z.B. *Traffic Shaping* mit abzudecken ist in der Architektur nicht vorgesehen. Auch XORP legt seinen Fokus auf Routing wobei es explizit auch als Testplattform für experimentelle Routingprotokolle dienen soll. Der Quellcode ist auch hier gut strukturiert, so dass sich neue Protokolle leicht den vorhandenen hinzufügen lassen. Wie in 2.1.2 beschrieben besitzt XORP allerdings den für eine Portierung auf eCos gravierenden Nachteil, dass XORP keine eigene Forwarding Engine mit sich bringt und somit auf Click oder die Funktionalitäten eines UNIX-Kernels angewiesen ist. Im Gegensatz dazu ist Click als komplett zu bezeichnen, d.h. dass keine weitere Software benötigt wird um einen voll funktionsfähigen Router zu konfigurieren. Click kann aufgrund seiner Architektur (siehe 2.2.1) sehr flexibel eingesetzt werden. So sind die Anwendungsbereiche nicht auf Routingaufgaben

[5] http://www.freebsd.org/releases/8.0R/relnotes.html
[6] http://www.zebra.org/

beschränkt. Ebenso lassen sich IP-Filter, oder Traffic Shaper konfigurieren. Durch die Art der Verschaltung der Elemente gestaltet sich auch eine Erweiterung der Funktionalität auf weitere Szenarien für den Entwickler als sehr einfach. Dieser Umstand war der ausschlaggebende Grund, warum die Wahl letztendlich auf Click fiel - weder bird noch XORP konnten sich in Bezug auf die Vielseitigkeit gepaart mit dem überzeugenden Konzept der Elemente mit Click messen.

3.2 Hardware und Virtualisierung

Das Ziel dieser Studienarbeit war es, einen Softrouter auf Basis von Click und eCos zu entwickeln. Dieser sollte als Demonstratorobjekt für das DANCEOS-Projekt[7] sowohl in einer virtualisierten Umgebung wie auch nativ auf echter Hardware laufen. Im Folgenden soll nun ein kurzer Überblick über die verwendete Hardwareplattform wie auch die Virtualisierungslösungen gegeben werden.

3.2.1 Die Hardware

Abbildung 3.2: Abbildung der verwendeten Hardware

[7] http://www.danceos.org

Im Bereich eingebetteter Systeme kommt häufig sehr ressourcenarme und dadurch leistungsschwache Hardware zum Einsatz. Die vorliegende Demonstratorhardware ist im Gegensatz dazu als sehr leistungsstark zu bezeichnen. Es handelt sich um einen Intel Atom K510 Zweikern-Prozessor mit 1,66 GHz pro Kern. Der verfügbare Hauptspeicher hat eine Größe von 2 GB. Als Zusatzausstattung verfügt das System über eine 3-Port-Gigabit-Netzwerkkarte mit Ethernet-Controllern vom Typ *Intel 82541PI*, die über den PCI-Bus mit dem System verbunden ist. Diese großzügige Ausstattung machte es möglich, die entstandene Implementierung später mit anderen Implementierungen auf Basis eine Linux-Systems auf der gleichen Hardware zu vergleichen.

3.2.2 Bochs

Bochs ist ist ein freier x86-Emulator, der unter der GNU Lesser General Public License (LGPL) lizensiert ist. Da Bochs jede x86-Instruktion emuliert, kann es sich performancetechnisch nicht mit Virtualisierungslösungen wie **VirtualBox**[8] oder **QEMU**[9] messen [6]. Auch ist der Umfang der emulierbaren Hardware nicht sonderlich umfangreich. So ermöglicht es Bochs nur eine einzige Netzwerkkarte vom Typ *NE2000* zu emulieren. Um diese für einen Softrouter nicht sinnvolle Limitation aufzuheben, war es zunächst nötig Bochs mit einem an der Universität von Tübingen entstandenen Patch[10] um die Fähigkeit zu erweitern, bis zu 4 Netzwerkkarten zu unterstützen. Obwohl die Voraussetzungen als Virtualisierungslösung trotzdem aufgrund der geringen Ausführungsgeschwindigkeit und der Beschränkung der Netzwerkkarten auf den o.g. Typ nicht optimal erschienen, sprachen zwei Gründe für einen Einsatz von Bochs. Zum einen eignet sich Bochs aufgrund seiner umfangreichen Debuggingfähigkeiten gut zum Debuggen von Betriebssystemen wie eCos und den darauf laufenden Anwendungen, zum anderen wurde Bochs beim DANCEOS-Projekt als Plattform für Fehlerinjektion gewählt.

3.2.3 QEMU

QEMU ist eine freie virtuelle Maschine, die unter der GPL lizenziert ist. QEMU unterstützt verschiedene CPU-Architekturen, u.a. PowerPC, SPARC und

[8] http://www.virtualbox.org
[9] http://www.qemu.org
[10] http://www.teifel-net.de/projects/bochs-4nic/index.html

x86 und erreicht durch die Verwendung von *dynamic binary translation* höhere Ausführungsgeschwindigkeiten als beispielsweise Bochs [6]. Durch die optionale Verwendung der *Kernel-Based Virtual Machine* kann Code fast genauso schnell wie auf der nativen Hardware ausgeführt werden. Da QEMU auch Netzwerkkarten vom gleichen Typ wie die der in 3.2.1 beschriebenen Hardware emulieren kann und mit seiner Anbindung an den GNU Debugger ebenfalls über gute Debuggingmöglichkeiten verfügt, wurde an einigen Stellen auch auf QEMU als Testplattform zurückgegriffen.

3.3 Zusammenfassung

Abschließend lässt sich feststellen, dass trotz der Vielfalt an freien Softroutern viele dieser Projekte aufgrund ihrer Zielsetzung, die oftmals nur aus der Unterstützung dynamischer Routingprotokolle besteht, nicht näher in Betracht gezogen wurden. Obwohl eine solche Ausrichtung durchaus legitim und je nach Anwendungszweck auch sinnvoll ist, waren die Anforderungen an den Softrouter für den Einsatz unter eCos vielfältiger. So kann das letztendlich gewählte Projekt Click durch seinen modularen Aufbau neben klassischem Routing auch weitere Aufgaben wie beispielsweise Paketfilterung erfüllen. Die Verschaltung der Elemente macht es möglich diese Funktionalitäten beliebig zu erweitern. Durch die POSIX-Kompatibilität, sowie die Unterstützung für FreeBSD wurde eine erfolgreiche Portierung auf eCos ermöglicht.

4 Implementierung

Im folgenden Kapitel wird zunächst detailliert auf die Funktionsweise des Netzwerkstack in eCos eingegangen. Im Anschluss wird geklärt, wie die Elemente des Click-Routers direkt mit dem Netzwerkstack kommunizieren können. Den Abschluss bildet die Betrachtung des ControlSocket-Elements, das eine Schnittstelle für die Kommunikation mit einem laufenden Click-Router anbietet.

4.1 Der Netzwerkstack

Um zu verstehen wie die Anbindung von Click an eCos funktioniert, benötigt man zunächst einen Überblick über die Organisation des Netzwerkstack von eCos. Grundsätzlich sind die Funktionalitäten des Stacks auf zwei Threads - *Network Alarm* und *Network Support* - innerhalb von eCos aufgeteilt. Zusätzlich findet selbstverständlich eine notwendige Behandlung von Interrupts auf einer weiteren Ebene statt. Nach einem kurzen Überblick über die generelle Behandlung von Interrupts in eCos, soll im Folgenden anhand eines eintreffenden Pakets skizziert werden, welche grundlegenden Schritte vom Erhalt des signalisierenden Interrupts bis zur Verarbeitung der Paketdaten durchlaufen werden müssen.

4.1.1 Interruptbehandlung

Interrupts, also asynchron auftretende Programmunterbrechungen und deren Bearbeitung sind ein elementarer Bestandteil aller modernen Betriebssysteme [5]. So erlauben es Interrupts zeitnah und ohne Techniken wie aktives Warten auf externe Ereignisse zu reagieren. Als Beispiel sei ein Timer-Interrupt genannt, der in periodischen Abständen generiert wird und es dem Betriebssystem erlaubt präemptives Scheduling zu realisieren. Da viele Geräte Interrupts generieren können, existieren mehrere Interruptleitungen, zur Unterscheidung von Interruptquellen. Allerdings kommt es dennoch häufig vor, dass sich mehrere Geräte eine

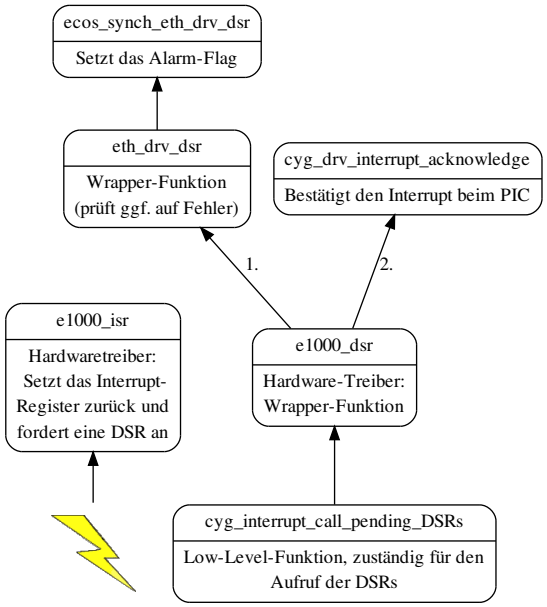


Abbildung 4.1: Interruptbehandlung: der linke Graph zeigt den Aufruf der ISR, der rechte die Funktionsaufrufe der DSR

Interruptleitung teilen müssen - dieser Umstand wird hier allerdings vernachlässigt. Beim Start des Betriebssystems wird die Interruptvektortabelle zunächst mit den Adressen der der *Interrupt Service Routinen* (ISR) für die verwendete Hardware gefüllt. Generiert eine Hardwarekomponente nun eine Interruptanforderung, kann die richtige Behandlungsroutine anhand der Interruptnummer herausgefunden und angesprungen werden. Während der Behandlung einer solchen Routine werden Interrupts niedrigerer Priorität global gesperrt. Für kurze ISRs mag dies kaum ein Problem darstellen, sobald eine Interruptbehandlungsroutine aber komplexere Aufgaben bewerkstelligen muss, kann dies problematisch werden. Aufgrund der langen Sperrzeiten können ggf. wichtige Interrupts verloren gehen. Aus diesem Grund kommt in eCos eine zweigeteilte Interruptbehandlung zum Einsatz, die die durch Interrupts verursachte Latenz minimieren soll. Tritt nun ein Interrupt auf, wird nun zunächst ebenfalls die ISR aufgerufen - diese führt aber höchstens sehr kurze Aufgaben selbst durch. Sollte eine umfangreichere Bearbeitung nötig sein, kann die ISR eine *Deferred Service Routine* (DSR) anfordern. Danach kehrt die ISR zurück und Interrupts werden wieder erlaubt. Die zugehörige DSR kann nun meistens sofort ausgeführt werden, außer der Scheduler ist gesperrt, weil ein Thread beispielsweise gerade einen Systemaufruf durchführt.

In diesem Fall wird die Ausführung der DSR so lange verzögert, bis die Schedulersperre wieder aufgehoben ist. Mit diesem Basiswissen kann nun die Behandlung eine auftretenden Interrupts der Netzwerkkarte betrachtet werden. Wie in Abbildung 4.1 dargestellt, signalisiert die Netzwerkkarte der CPU zunächst, dass ein externes Ereignis vorliegt, das bearbeitet werden muss. Nach dem Eintritt in die ISR, die hier im Treiber für die Netzwerkkarte definiert wird, wird eine DSR angefordert. Diese DSR bestätigt den Erhalt des Interrupts und setzt ein Flag, das dem Alarm-Thread signalisiert, dass ein neues Ereignis vorliegt. Damit ist die Interruptbehandlung abgeschlossen.

4.1.2 Network Alarm

Der *Network-Alarm-Thread* läuft in einer Endlosschleife die das in der DSR gesetzte Alarm-Flag überprüft. Über einige Indirektionen wird, wie in Abbildung 4.2 dargestellt, bei einer positiven Überprüfung eine Funktion des Netzwerkkartentreibers aufgerufen. Diese ermittelt zunächst die Länge der verfügbaren Daten. Mit dieser Information ruft die Treiberfunktion eine hardwareunabhängige Funktion auf, die einen Speicherbereich mit der ermittelten Größe allokiert. In den höher gelegenen Schichten des FreeBSD-Netzwerkstack werden Paketdaten zusammen mit vielen Zusatzinformationen immer in einer speziellen Struktur, genannt *Memory Buffer* (*mbuf*) vorgehalten. Tiefere Schichten wie beispielsweise der Treiber operieren auf einer viel einfacheren Datenstruktur - einem Array mit Elementen vom Typ *struct eth_drv_sg*. Jeder Arrayeintrag besteht hier nur aus einem Datenzeiger, sowie einem Eintrag für die Länge des referenzieren Datenbereichs. Um ein mehrmaliges Kopieren der Paketdaten im Stack zu verhindern, werden bei der Allkokation des o.g. Speichers bereits Memory Buffers verwendet. Die Einträge im Array verweisen dann jeweils auf die Datenbereiche der Memory Buffers. Daraufhin werden die Daten von der Hardware in den Speicher kopiert. Hier musste eine geringfügige Anpassung im Stack erfolgen. Click erwartet, dass die Paketdaten im Memory Buffer den Ethernet-Header enthalten. Im eCos-Stack wird der Ethernet-Header allerdings in einen gesonderten Speicherbereich kopiert, der nicht im Memory Buffer enthalten ist. Dieser Umstand muss später beachtet werden, wenn Click die Bearbeitung einzelner Pakete dem Netzwerkstack überlässt. Bevor die nun vorliegenden Daten eine Funktion durchlaufen, die eine erste Klassifikation der Pakete nach dem Ethernet-Typ-Feld

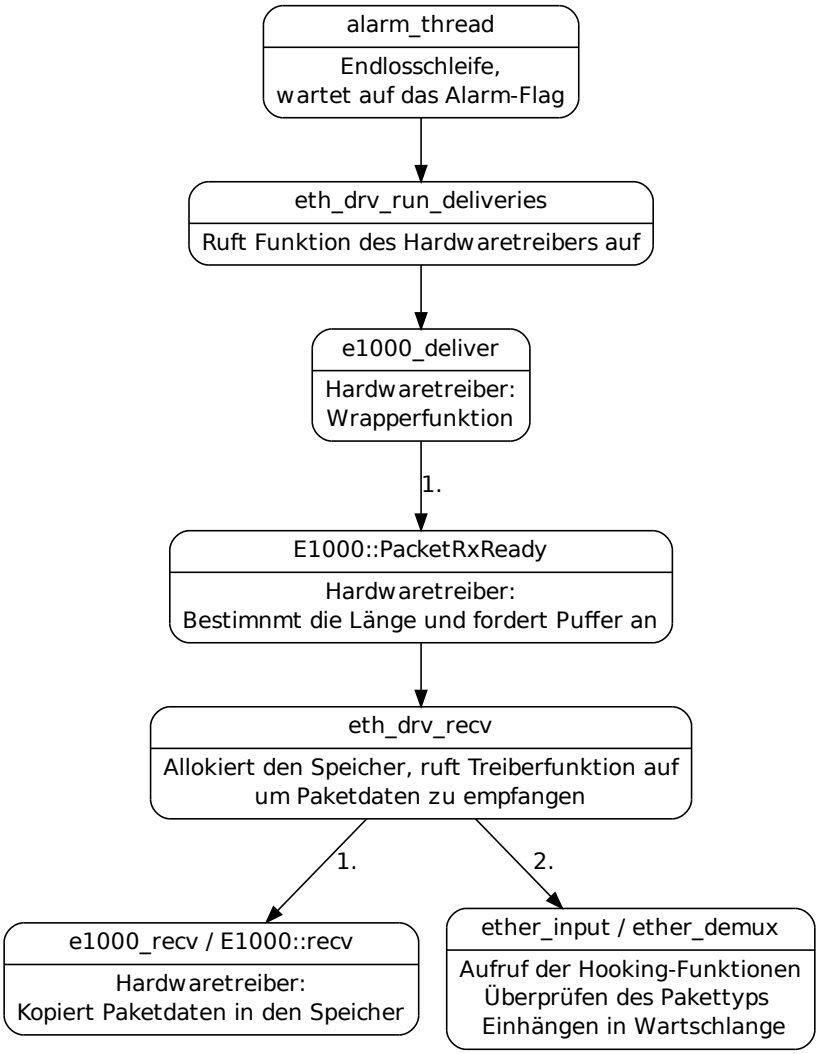


Abbildung 4.2: Network Alarm

des Pakets durchführt, existiert die Möglichkeit, die Paketdaten einer benutzerdefinierten Bearbeitungsroutine zu übergeben. Dies wird hier erwähnt, da das einer der wichtigsten Ansatzpunkte für Click ist. Mit der Typinformation wird das Paket danach in eine Warteschlange eingereiht, wobei für jeden erkannten Typ eine eigene Warteschlange existiert. Zusätzlich wird in einer Statusvariable ein Bit gesetzt, um den *Network-Support-Thread* zu benachrichtigen, dass neue Daten in der Warteschlange für den ermittelten Typ vorliegen.

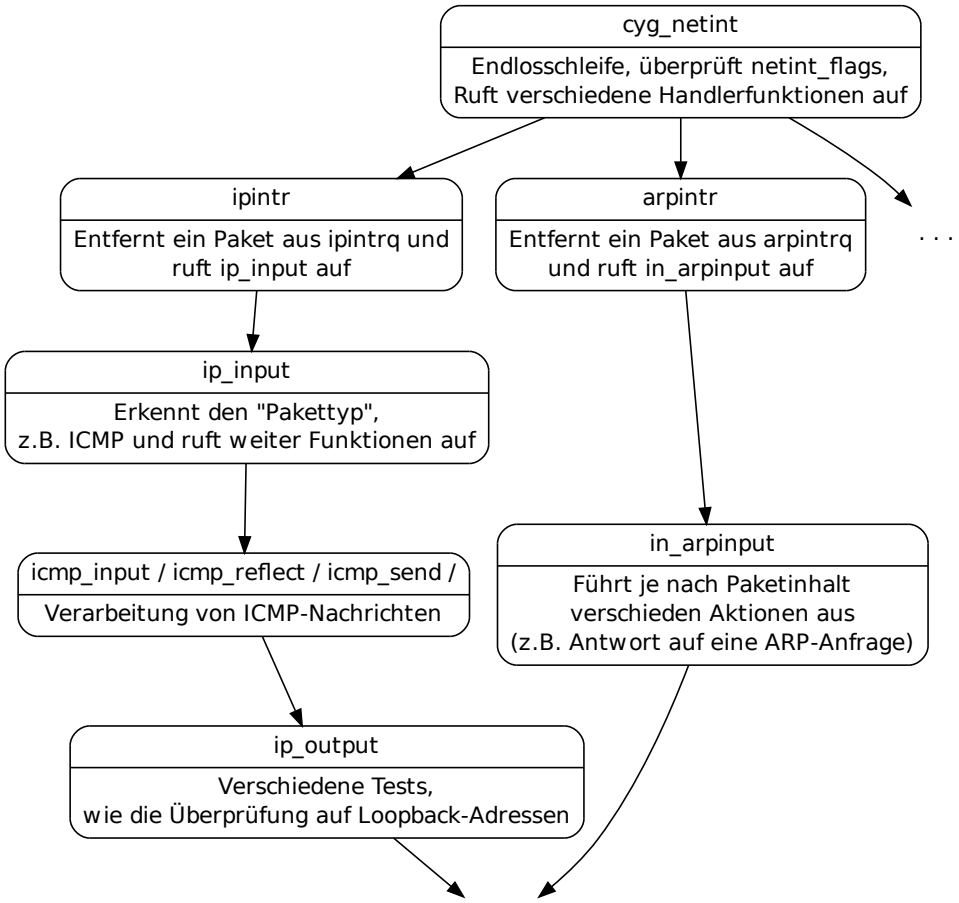


Abbildung 4.3: Network-Support-Thread: unterschiedliche Behandlung verschiedener Pakettypen, hier am Beispiel von IP- bzw. ARP-Paketen

4.1.3 Network Support

Der *Network-Support-Thread* komplettiert den eCos-Netzwerkstack durch die eigentliche Bearbeitung der Paketdaten. Die Betrachtung geschieht hier prinzipiell nur der Vollständigkeit halber, da die Verarbeitung der Pakete später nahezu komplett von Click übernommen wird. Der Network Alarm Thread läuft wie der Network-Support-Thread ebenfalls als Endlosschleife und überprüft mit einer blockierenden Operation die Statusvariable, die vom Alarm-Thread gesetzt wird. Sollte diese signalisieren, dass neue Daten vorliegen, wird ein entsprechender Handler aufgerufen. Jeder dieser Handler entfernt nun Pakete aus einer der Warteschlangen und reicht sie an Funktionen zur Verarbeitung weiter. Abbildung 4.3 zeigt, wie verschiedene Pakettypen jeweils von anderen Funktionen behandelt

werden. Danach werden die Pakete von diesen Funktionen weiter klassifiziert und dementsprechend verarbeitet. Dies ist ebenfalls in Abbildung 4.3 zu sehen. Sie zeigt die weitere Verarbeitung einer ICMP-Echo- bzw. einer ARP-Anfrage. Da die Bearbeitung von ICMP-Nachrichten komplexer ist, sind mehr Funktionen daran beteiligt. Nachdem diese paketspezifischen Behandlungen abgeschlossen sind, werden eventuelle Antwortpakete wie in Abbildung 4.4 dargestellt ab einem gewissen Punkt wieder gemeinsam behandelt, da für ein Versenden des Pakets die gekapselten Daten der Transport- bzw. Anwendungsschicht nicht von Bedeutung sind. Auch hier findet zunächst eine hardwareunabhängige Verarbeitung statt, die für alle Netzwerkkartentreiber gleich abläuft. Erst beim eigentlichen Versand der Pakete müssen hardwarespezifische Funktionen verwendet werden, beispielsweise um der Netzwerkkarte zu signalisieren, dass neue Paketdaten zum Versand bereit sind.

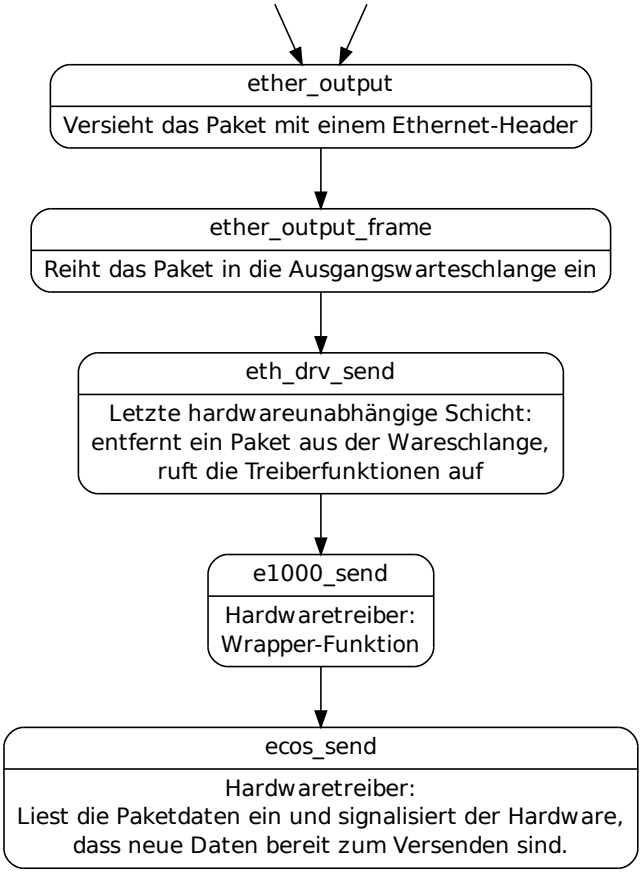


Abbildung 4.4: Fortsetzung zu Abbildung 4.3: Versand eines Netzwerkpaketpakets

4.1.4 Memory Buffers

Die zentrale Struktur innerhalb des eCos-Netzwerkstack sind die aus BSD stammenden *Memory Buffers* (*mbuf*). So gibt es für jedes Netzwerkpaket, das gerade den Stack durchläuft mindestens einen solchen mbuf. Allerdings ist es auch möglich, dass ein Paket sich über mehrere Memory Buffers erstreckt - in diesem Fall spricht man von einer *mbuf-chain* (verkettete Liste von mbufs). Auflistung 4.1 zeigt den grundlegenden Aufbau der Struktur.

```
struct mbuf {
  struct  m_hdr m_hdr;
  union {
    struct {
      struct  pkthdr MH_pkthdr; /* M_PKTHDR set */
      union {
        struct  m_ext MH_ext; /* M_EXT set */
        char  MH_databuf[MHLEN];
      } MH_dat;
    } MH;
    char  M_databuf[MLEN];      /* !M_PKTHDR, !M_EXT */
  } M_dat;
};
```

Listing 4.1: Aufbau eins Memory Buffers

Jeder mbuf verfügt über einen Header *m_hdr*, der u.a. die Zeiger aufs nächste Element der mbuf-chain, sowie die die Länge des Datenbereichs des aktuellen mbufs enthält. Handelt es sich um das erste Element in einer mbuf-chain, findet sich eine Struktur vom Typ *struct pkthdr* im mbuf. Diese enthält ebenfalls verschiedene Informationen, beispielsweise einen Zeiger auf die Verwaltungsstruktur der empfangenden Netzwerkschnittstelle. Sind die zu speichernden Paketdaten nicht zu groß, können sie in einem kleinen Speicherbereich direkt im mbuf abgelegt werden. Sollte der Platz nicht ausreichen, können weitere mbufs der Kette hinzugefügt werden. Ebenfalls möglich ist die Speicherung in einem externen Speicherbereich, der dann im mbuf referenziert wird. Um die Arbeit mit mbufs möglichst einfach und performant zu gestalten, existieren eine Vielzahl an Präprozessormakros, die typische Operationen, wie das Entfernen oder Hinzufügen von Headerdaten zu einem Paket möglichst effizient implementieren. Wie in Kapitel 2.2.1.4 beschrieben, kapselt Click die mbuf-Struktur in einer Paketklasse, ver-

wendet aber zur Manipulation der Daten ebenfalls die vom Stack zur Verfügung gestellten Makros. Diese Vorgehen hat den Vorteil, dass unnötige Kopieraktionen vermieden werden, was einen positiven Effekt auf die Leistung hat.

4.1.5 Netzwerkkarten

Im Konfigurationsprozess von eCos lässt sich die Anzahl der bereitgestellten Netzwerkschnittstellen konfigurieren. Diese Anzahl ist jedoch auf zwei Netzwerkkarten beschränkt. Da die Hardware aber wie in 3.2.1 beschrieben über eine 3-Port-Netzwerkkarte verfügt, hätte dies zur Folge gehabt, dass sich tatsächlich nur zwei der verfügbaren drei Port nutzen lassen. Um diese Beschränkung aufzuheben, muss das System an zwei grundlegenden Punkten angepasst werden. Der erste Punkt betrifft den Treiber. Jede Schnittstelle benötigt hier eine eigene private Datenstruktur zur Verwaltung - so auch die neu hinzuzufügende. Der zweite Ansatzpunkt befindet sich in den Initialisierungsroutinen für die Netzwerk-Unterstützung. Hier werden die vom Nutzer definierten Vorgaben, wie IP-Adressen und Netzmasken den Schnittstellen zugewiesen. Diese Abschnitte müssen je um eine Initialisierungsanweisungen erweitert werden.

Für den Fall, dass in der verwendeten Hardware erheblich mehr Schnittstellen zur Verfügung stehen, würde es sich anbieten, die entsprechenden Codeabschnitte dynamischer zu gestalten, um theoretisch eine beliebige Anzahl an Ports unterstützen zu können. Für den vorliegenden Fall war dies aber nicht notwendig.

4.2 Anbindung an eCos

Da Netzwerkpakete, die von Click verarbeitet werden, in aller Regel keiner weiteren Bearbeitung durch das Betriebssystem bedürfen, muss die Verbindung zwischen Click und dem Netzwerkstack zwei wesentliche Punkte erfüllen. Zum einen müssen Pakete möglichst früh, vor einer aufwendigen Bearbeitung im Netzwerkstack abgegriffen werden und zum anderen muss es dennoch möglich sein bestimmte Pakete gezielt durch das Betriebssystem behandeln zu lassen. Beides kann mit den im Folgenden vorgestellten Click-Elementen erreicht werden.

4.2.1 FromDevice

Das FromDevice-Element in Click ist dafür zuständig, Pakete von einer definierbaren Netzwerkschnittstelle zu empfangen. Im Folgenden soll erklärt werden, wie dies realisiert wird. FromDevice ist ein normales Click-Element, also eine

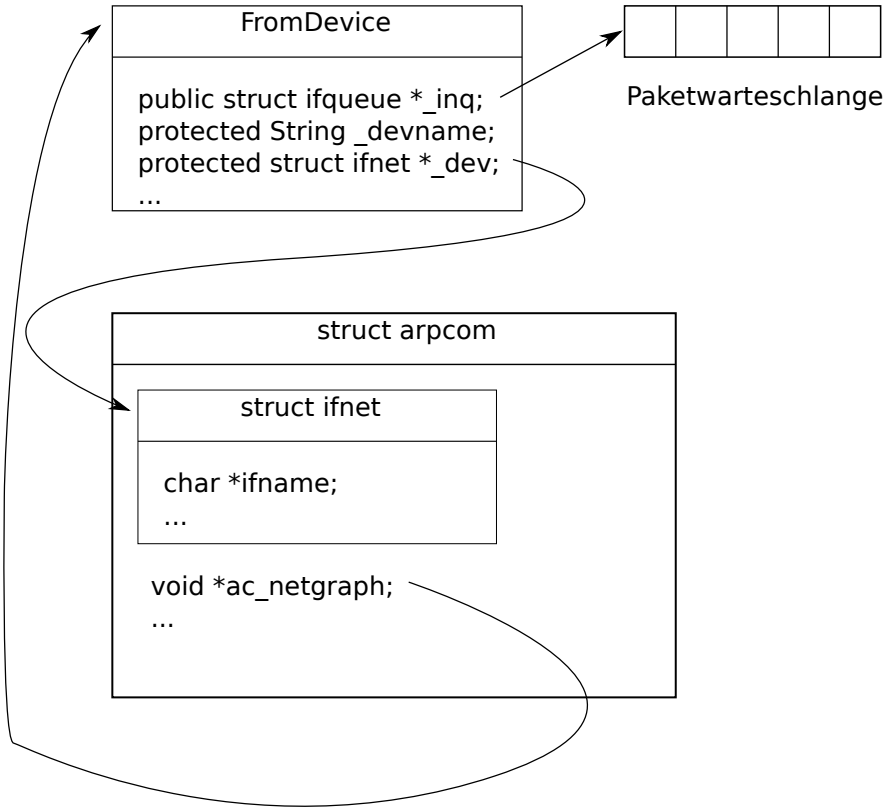


Abbildung 4.5: Verweise zwischen dem FromDevice-Element und der Verwaltungsstruktur der Netzwerkschnittstelle. Für jede Netzwerkschnittstelle, von der Pakete empfangen werden sollen, muss eine solche Verbindung zwischen einem FromDevice-Element und der Verwaltungstruktur hergestellt werden.

C++-Klasse, von der eine Instanz pro Netzwerkschnittstelle beim Start des Routers erstellt werden kann. Wie in Abbildung 4.5 zu sehen, enthält die Klasse als Membervariable u.a. einen Zeiger auf eine Warteschlange vom Typ *struct ifqueue* der *public* deklariert ist - d.h. es kann mit einem Verweis auf die Instanz der Klasse auch von außen darauf zugegriffen werden. Bei der Initialisierung des Elements wird nun zunächst anhand des Namens der Netzwerkschnittstelle die zur Netzwerkschnittstelle gehörige Verwaltungsstruktur ermittelt. Innerhalb dieser Struktur wird dann ein *void*-Zeiger (*ac_netgraph*) auf den *this*-Zeiger der

aktuellen FromDevice-Instanz gesetzt. Somit kann über die Verwaltungsstruktur einer Schnittstelle direkt auf das zugehörige FromDevice-Element und damit auf die enthaltene Warteschlange zugegriffen werden. Mit diesen Vorbereitungen können nun Daten aus dem Stack entfernt und an Click weitergereicht werden. Abbildung 4.6 zeigt einen bereits aus Abbildung 4.2 bekannten Aus-

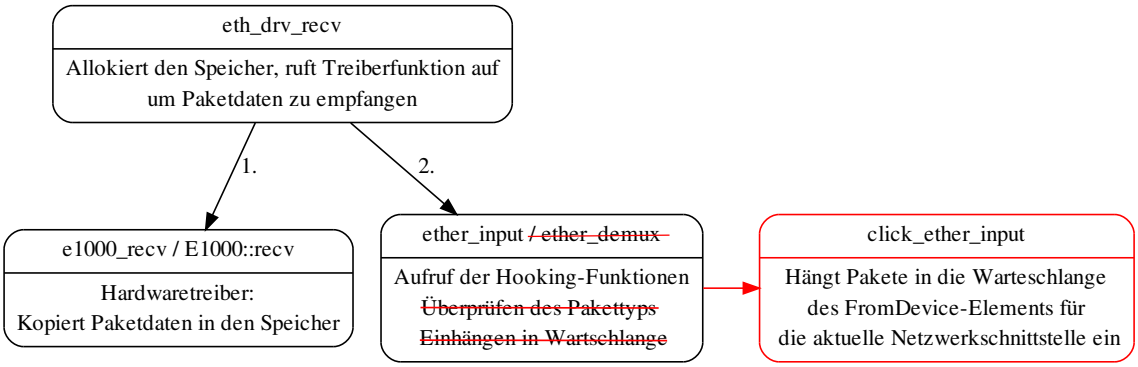


Abbildung 4.6: Eingriff in den Netzwerkstack durch die Hooking-Funktion

schnitt des Netzwerkstacks. Allerdings findet hier nunmehr weder eine Klassifizierung nach dem Ethernet-Typ, noch das Einhängen in eine Warteschlange statt. Stattdessen werden die Pakete nur noch der benutzerdefinierten Funktion *click_ether_input* übergeben. Diese Funktion bekommt neben einem Verweis auf die mbuf-Struktur auch einen Zeiger auf die Verwaltungsstruktur der empfangenden Netzwerkschnittstelle übergeben. Durch die referenzierte Verwaltungsstruktur kann der Zeiger auf die FromDevice-Instanz für das Interface in Erfahrung gebracht werden. Über diesen wird der Memory Buffer in die Warteschlange des FromDevice-Elements eingehängt. Bevor *click_ether_input* zurückkehrt, wird der übergebene Zeiger auf den Memory Buffer auf *NULL* gesetzt. Durch diese Maßnahme wird dem Stack mitgeteilt, dass keine weitere Verarbeitung der Paketdaten durchzuführen ist. Somit kehrt *ether_input* sofort zurück, ohne das Paket in eine der Warteschlangen des Netzwerkstacks einzuhängen.

```
1  if (ng_ether_input_p != NULL) {
2    int retval = (*ng_ether_input_p)(ifp, &m, eh);
3    if (m == NULL) return; //do nothing, click handles further
         processing
4    switch (retval){
5      case 0: //packet is not for click
6        copy_hdr = 1;
```

```
7           break;
8         case 1: //packet comes from ToHost module, which already
              handled the ethernet header
9           copy_hdr = 0;
10          break;
11        default:
12          break;
13      }
14    }
15
16      ...
17
18    if(copy_hdr){
19      memcpy(eh, mtod(m, char *), sizeof(struct ether_header));
20      m_adj(m, sizeof(struct ether_header));
21    }
```

Listing 4.2: Hooking im Netzwerkstack

Dies ist in Auflistung 4.2 zu sehen, *ng_ether_input_p* ist ein Funktionszeiger, der auf *click_ether_input* zeigt. Allerdings existieren weitere Tests für den Fall, dass *m* nach dem Funktionsaufruf ungleich *NULL* ist. Dies kann zweierlei Gründe haben. Einerseits könnte das Paket vom in Abschnitt 4.2.3 beschriebenen *ToHost*-Element stammen, andererseits ist es aber auch möglich, dass für die Netzwerkschnittstelle kein FromDevice-Element vorhanden ist. In diesem Fall muss der Ethernet-Header aus dem mbuf extrahiert und in einem separaten Speicherbereich abgelegt werden - so wie es für die weitere Verarbeitung im Netzwerkstack von eCos erwartet wird. Pakete die von eCos selbst verschickt werden, werden ebenfalls von einer Callback-Funktion abgefangen. Dort werden sie allerdings nur in eine Warteschlange eingereiht. Zusätzlich signalisiert die Callback-Funktion dem Network-Support-Thread, dass neue Daten bereit zum Versand sind.

4.2.2 ToDevice

Das ToDevice-Element erlaubt es das Versenden von Paketen. ToDevice verfügt über eine *run_task*-Methode, die vom Click-Scheduler aufgerufen wird. In dieser Methode wird über einen pull-Aufruf ein Click-Paket vom Vorgängerelement angefordert. Sollte dies Erfolg haben, wird der mbuf-Zeiger aus dem Click-Paket extrahiert und in den eCos-Netzwerkstack injiziert wie in Abbildung 4.7 dar-

gestellt. Im Vergleich mit Abbildung 4.4 ist zu erkennen, dass die aufgerufene Methode *ether_output_frame* bereits ein Paket mit vorangestelltem Ethernet-Header erwartet. Von daher muss das ToDevice-Element keine Manipulation der Paketdaten vornehmen. Der pull-Aufruf geschieht innerhalb einer Schleife mit

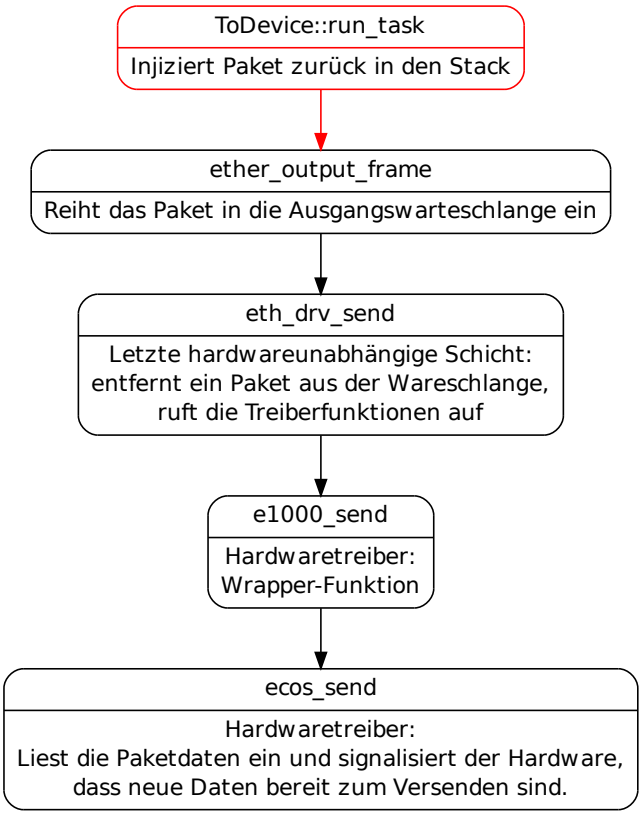


Abbildung 4.7: Injizieren eines Pakets in den Stack

einer variablen Anzahl an Durchgängen, damit die *run_task*-Methode nicht für jedes Paket extra aufgerufen werden muss. Sollte der pull-Aufruf kein Paket zurückliefern, oder die maximale Anzahl an Durchläufen erreicht sein, kehrt die Methode zurück.

4.2.3 ToHost

Da die Aufgabe eines Softrouters das Weiterleiten von Paketen ist, spricht nichts dagegen, dass Click zunächst alle einkommenden Pakete abfängt und verarbeitet. Allerdings existieren auch Szenarien, in denen eine Verarbeitung der Paketdaten durch das Betriebssystem notwendig ist. So ist es keineswegs garantiert,

dass Click die einzige Anwendung ist, die auf dem System läuft. Würde als Zu-

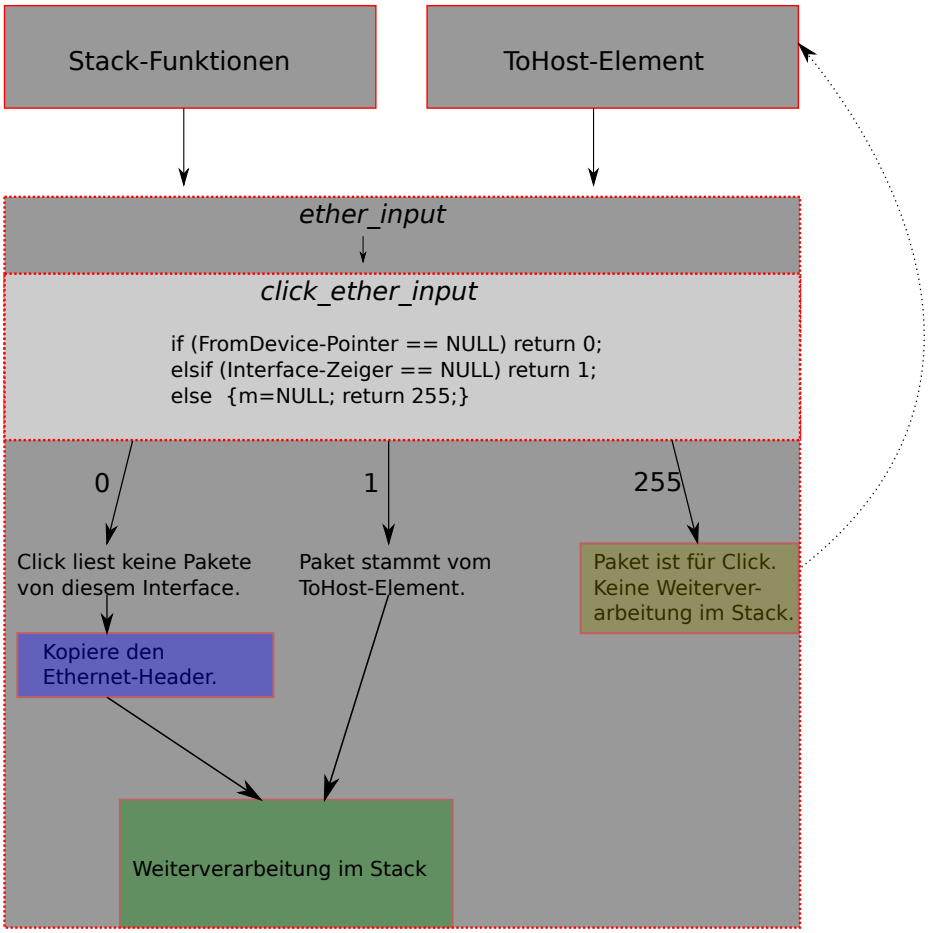


Abbildung 4.8: Soll ein Paket vom eCos-Stack behandelt werden, muss es die *ether_ input*-Funktion zweimal durchlaufen. Im ersten Durchlauf wird das Paket von Click behandelt und an das ToHost-Element weitergeleitet. Dieses setzt den Interface-Zeiger auf *NULL* und ruft nochmals *ether_ input* für das Paket auf.

satzapplikation ein Webserver auf dem System laufen, müssen Pakete, die an die IP-Adresse des Systems geschickt werden, den Netzwerkstack durchlaufen, damit die Anwendung über Socketoperationen auf die Paketdaten zugreifen kann. Für diesen Fall kann das *ToHost*-Element Pakete in den Stack schicken, ohne dass diese wieder von Click behandelt werden. Dazu wird in der push-Methode von ToHost der Ethernet-Header aus dem mbuf entfernt und gespeichert. Nachdem der Zeiger auf die Verwaltunggstruktur der Netzwerkschnittstelle gesichert wurde, wird dieser Zeiger in der mbuf-Struktur auf *NULL* gesetzt. Auf diesem

Weg wird *click_ether_input* signalisiert, dass das Paket vom ToHost-Element stammt. Abbildung 4.8 veranschaulicht dieses Zusammenspiel.

4.3 ControlSocket

Je nach Betriebsart bietet Click unterschiedliche Möglichkeiten um mit einem laufenden Router zu kommunizieren. Die Linux- und BSD-Module nutzen hier als Schnittstelle ein virtuelles Dateisystem. Der Benutzer kann also durch das Lesen und Schreiben von Dateien in diesem Dateisystem mit Click interagieren. Im Userlevel-Modus bietet Click ein Element an, das eine solche Konfigurationsschnittstelle über einen Socket anbietet. Mittels Konsolenanwendungen wie **telnet** oder **netcat**[1] lässt sich so eine Verbindung zum System herstellen um textbasierte Kommandos an dasselbe zu schicken. Zusätzlich existiert eine GUI-Anwendung **clicky**[2] die eigens für die Kommunikation mit Click über das ControlSocket-Element entwickelt wurde. Anders als beispielsweise Linux, ist eCos kein System das typischerweise eine interaktive Bedienung durch einen Benutzer vorsieht. Deshalb wurde das *ControlSocket*-Element für eine Verwendung in eCos angepasst.

4.3.1 Ausgliederung aus der Elementkette

Aufgrund der Art, wie Click Paketdaten vom eCos-Netzwerkstack empfängt, kommt es beim Einfügen des ControlSocket-Elements in eine Click-Konfiguration zu einer Verklemmung. Das ControlSocket-Element nutzt den *read*[3]-Systemaufruf um Daten von einem Socket zu lesen. Da dies blockierend geschieht, blockiert beim Lesen der gesamte Click-Thread. Allerdings werden alle Pakete - auch die, die an die IP-Adresse des Systems adressiert sind (s. 4.2) von Click abgefangen. Hieraus resultiert ein Deadlock. Der Click-Thread wartet an der read-Operation auf Daten, deren Bereitstellung von einem anderen Element im gleichen Thread initiiert werden muss. Als prinzipielle Lösungsmöglichkeiten für dieses Problem standen zwei Ansätze zur Auswahl.

[1] http://netcat.sourceforge.net
[2] http://read.cs.ucla.edu/click/clicky
[3] man 2 read

Statt alle Pakete in den den Hooking-Funktionen an Click weiterzuleiten, hätte man die Paketdaten bereits hier auf ihr Ziel hin untersuchen können. Wäre die Zieladresse mit der IP-Adresse des Systems identisch, wären die Pakete direkt an den Netzwerkstack weitergeleitet worden. Dies hätte den Vorteil gehabt, dass das ControlSocket-Element weiterhin im gleichen Thread wie alle weiteren Click-Elemente hätte laufen können. Gegen dieses Vorgehen sprach, dass der Bearbeitungsaufwand in den Hooking-Funktionen massiv gestiegen wäre sowie die Tatsache, dass Click nicht mehr die Möglichkeit hätte, Pakete, die ans Host-System adressiert sind, zu filtern. Darum wurde hier als Alternative die Arbeit des ControlSocket-Elements in einen separaten Thread ausgelagert. Das Element wird vor dem Start des Routers in der main-Funktion über eine Sprungbrett-Funktion im Kontext eine neuen Threads gestartet. Hier wird nach einer Initialisierung die Hauptfunktion des Elements aufgerufen. Diese kann nun in einer Endlosschleife mittels *read* Eingaben vom erstellten Socket lesen. Eine Steueranweisung die vom Socket gelesen wird, kann sich über mehrere *read*-Aufrufe erstrecken. So versendet beispielsweise *telnet* Benutzereingaben zeilenweise. Darum wird nur das erste *read* blockierend ausgeführt. Sollte dies Erfolg haben, werden die empfangen Daten in eine Puffer geschrieben. Ist das Kommando nicht vollständig, wird der nächste read-Aufruf gestartet, diesmal allerdings in einer nicht-blockierenden Version. Dies wiederholt sich bis das Kommando vollständig empfangen wurde, oder bis der nicht-blockierende Aufruf fehlschlägt und signalisiert, dass er blockieren würde. Danach wird wieder in den blockierenden Modus geschaltet.

4.3.2 Kommunikation mit Click

Auf ein vollständig erhaltenes Kommando muss reagiert werden, allerdings ist zwischen der Art der Kommandos zu unterscheiden. *Read*-Kommandos beeinflussen den Objektzustand eines Elements nicht und können praktisch zu jeder Zeit ausgeführt werden. *Write*-Kommandos greifen hingegen in den Objektzustand ein und dürfen daher nur an fest definierten Stellen vorkommen. Bei einem Read-Kommando wird daher nur der Elementhandler ausfindig gemacht und aufgerufen. Die Ausführung von Write-Kommandos findet hingegen nur zwischen den Task-Aufrufen im Click-Scheduler statt. Write-Kommandos sind daher auf eine Kommunikation mit dem Click-Thread angewiesen. eCos stellt für solche Zwecke die aus der UNIX-Welt bekannten *Message Queues* bereit. Threads können ei-

gene Message Queues erstellen oder schon vorhandene Message Queues anderer Threads öffnen um somit untereinander Daten auszutauschen. Der ControlSocket-Thread erstellt eine Message Queue um mit dem Click-Thread kommunizieren zu können. Der Click-Scheduler wurde dahingehend erweitert, dass er zwischen den Task-Einlastungen die Message Queue auf neue Nachrichten überprüft. Sollte eine neue Nachricht vorliegen, wird das darin enthaltene Kommando ausgeführt und das Ergebnis über eine weitere Message Queue an den ControlSocket-Thread zurückgeliefert. Da die Kommandos nur zwischen den einzelnen Task-Aufrufen stattfinden, ist sichergestellt, dass durch die Ausführung von Write-Kommandos kein inkonsistenter Elementzustand hergestellt wird. Als Besonderheit kann auf diese Weise auch die komplette Routerkonfiguration ausgetauscht werden. Soll eine Neukonfiguration stattfinden, wird dies dem Click-Thread inkl. einer neuen Konfiguration mitgeteilt. Im sog. *Hotconfig*-Handler kann dann diese Konfiguration geparst und ein neuer Router erstellt werden. Im Anschluss wird der noch laufende Router beendet und durch den Neuen ersetzt.

4.4 Zusammenfassung

Die genaue Betrachtung des eCos-Netzwerkstacks hat gezeigt, wie eine funktionierende Verbindung zwischen Click und dem Stack realisiert werden kann. Dabei mussten Besonderheiten, wie die separate Behandlung von Ethernet-Headern im eCos-Netzwerkstack, berücksichtigt werden. Die Weitergabe der Netzwerkpakete an Click erfolgt über ein Warteschlange, die von Click-Elementen abgearbeitet wird. Umgekehrt können die Element aber auch Pakete zurück in den Stack schicken, um somit eine Verarbeitung der Pakete durch das Betriebssystem zu erreichen. Die Interaktion mit einem laufenden Click-Router geschieht über eine Socket-Verbindung, über die textbasierte Kommandos verschickt werden.

5 Evaluation

In diesem Kapitel soll die Leistung des Click-Routers unter eCos evaluiert werden. Hierfür wurde der Router mit zwei unterschiedlichen Konfigurationen betrieben und mit drei verschiedenen Implementierungen auf Basis eines Linux-System verglichen. Diese drei Implementierungen umfassen Clicks Userlevel-Modus, Clicks Linux-Kernelmodul sowie klassisches Linux-Kernel-Routing, wobei bei letzterem eine mit der verwendeten Click-Konfiguration vergleichbare Konfiguration mit Hilfe der **iproute2-tools**[1] erstellt wurde. Auf einen Vergleich mit der BSD-Version von Click wurde verzichtet, da sich hier aufgrund der Instabilität der Implementierung keine sinnvollen Tests durchführen ließen.

5.1 Der Versuchaufbau

Aufgrund beschränkter Ressourcen wurde der Testaufbau so einfach wie möglich gehalten. Wie in Abbildung 5.1 zu sehen verbindet ein Router zwei Rechner aus unterschiedlichen Subnetzen, d.h. der Router ist mit beiden Subnetzen direkt verbunden. Auf der in 3.2.1 beschriebenen Hardware kam wahlweise eCos oder die Linux-Distribution **Grml**[2] zum Einsatz. Im Folgenden wird diese Hardware als Router bezeichnet. Bei den als Clients verwendeten Rechnern handelt es

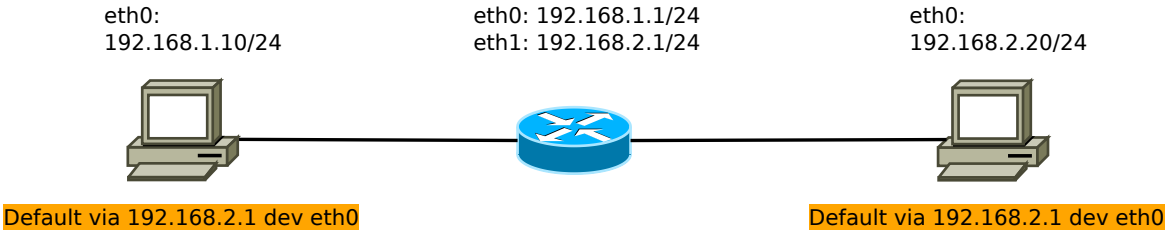


Abbildung 5.1: Testaufbau mit dem Router in der Mitte, sowie den beiden Clients

[1] http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2
[2] http://grml.org

sich um Vierkern-CPUs vom Typ *Intel Core2 Q9550* mit 2.83 GHz, mit 4 GB Hauptspeicher. Als Netzwerkkarten kamen die auf den Mainboards integrierten Gigabit-Netzwerkadapter vom Typ *Intel 82567LM-3* zum Einsatz. Die Rechner wurden mit einer 64bit-Version der Linux-Distribution *Grml* betrieben.

5.2 Die Konfigurationen

5.2.1 Der IP-Router

Wie eingangs erwähnt, wurde Click mit zwei verschiedenen Konfigurationen betrieben. Diese unterscheiden sich zum einen in ihrer Funktionalität und zum andern in der Anzahl der beteiligten Elemente. Die erste Konfiguration implementiert einen IP-Router der statisches Routing unterstützt. Auflistung 5.1 zeigt diese Konfiguration in einer gekürzten Version. Sie zeigt nur die jeweiligen Pfade für die Netzwerkschnittstelle *eth0* und das zugehörige Subnetz - die Einträge für die weiteren Schnittstellen, deren Konfiguration ganz analog geschieht, wurde aus Gründen der Übersichtlichkeit entfernt.

```
1  ip :: Strip(14)
2  -> CheckIPHeader(INTERFACES 172.16.1.1/255.255.255.0
       172.16.2.1/255.255.255.0 172.16.3.1/255.255.255.0)
3  -> rt :: StaticIPLookup(
4  172.16.1.1/32 0,
5  172.16.1.255/32 0,
6  172.16.1.0/32 0,
7  172.16.2.1/32 0,
8  172.16.2.255/32 0,
9  172.16.2.0/32 0,
10 172.16.3.1/32 0,
11 172.16.3.255/32 0,
12 172.16.3.0/32 0,
13 172.16.1.0/255.255.255.0 1,
14 172.16.2.0/255.255.255.0 2,
15 172.16.3.0/255.255.255.0 3,
16 255.255.255.255/32 0.0.0.0 0,
17 0.0.0.0/32 4,
18 0.0.0.0/0.0.0.0 18.26.4.1 4);
19
20 // ARP responses are copied to each ARPQuerier and the host.
```

```
21 arpt :: Tee(4);
22
23 // Input and output paths for eth0
24 c0 :: Classifier(12/0806 20/0001, 12/0806 20/0002, 12/0800, -);
25 FromDevice(eth0) -> c0;
26 out0 :: Queue(1000) -> todevice0 :: ToDevice(eth0);
27 c0[0] -> ar0 :: ARPResponder(172.16.1.1 FE:FD:00:00:00:00) -> out0;
28 arpq0 :: ARPQuerier(172.16.1.1, FE:FD:00:00:00:00) -> out0;
29 c0[1] -> arpt;
30 arpt[0] -> [1]arpq0;
31 c0[2] -> Paint(1) -> ip;
32 c0[3] -> Print("eth0 non-IP") -> Discard;
33
34 // Input and output paths for eth1
35 // ...
36
37 // Input and output paths for eth2
38 // ...
39
40 // Local delivery
41 toh :: ToHost;
42 arpt[3] -> toh;
43 rt[0] -> EtherEncap(0x0800, 1:1:1:1:1:1, 2:2:2:2:2:2) -> toh;
44
45 // Forwarding path for eth0
46 rt[1] -> DropBroadcasts
47 -> cp0 :: PaintTee(1)
48 -> gio0 :: IPGWOptions(172.16.1.1)
49 -> FixIPSrc(172.16.1.1)
50 -> dt0 :: DecIPTTL
51 -> fr0 :: IPFragmenter(1500)
52 -> [0]arpq0;
53 dt0[1] -> ICMPError(172.16.1.1, timeexceeded) -> rt;
54 fr0[1] -> ICMPError(172.16.1.1, unreachable, needfrag) -> rt;
55 gio0[1] -> ICMPError(172.16.1.1, parameterproblem) -> rt;
56 cp0[1] -> ICMPError(172.16.1.1, redirect, host) -> rt;
57
58 // Forwarding path for eth1 ...
59 // Forwarding path for eth2 ...
```

Listing 5.1: IP-Router-Konfiguration

Die Kernkomponenten der Konfiguration sind neben den bereits in 4.2 beschriebenen Elementen, die Elemente *Classifier* und *StaticIPLookup*. Der Classifier ist das erste Element, das die empfangen Pakete verarbeitet. Er klassifiziert die Pakete anhand vorher definierter Muster und leitet sie dann an den dementsprechenden Ports weiter. Der Konfigurationsstring des Classifiers enthält dabei pro Port einen oder mehrere Einträge der Form *Offset/Pattern*. Sollte also ein einkommendes Paket an Stelle *Offset* die Bytesequenz *Pattern* aufweisen, wäre der Test erfolgreich und das Paket würde den Classifier auf dem dazugehörigen Port verlassen. In o.g. Konfiguration besitzt der Classifier dementsprechend drei fest definierte Einträge die zu den Ports *0*, *1* und *2* korrespondieren, sowie einen Standard-Eintrag, der gewählt wird, wenn kein vorheriges Muster auf ein Paket gepasst hat. Die ersten beiden Einträge passen auf ARP-Anfragen bzw. ARP-Antworten, der dritte wählt IP-Pakete aus.

Das Element *StaticIPLookup* ist dafür zuständig, die Route für ein empfangenes Paket zu bestimmen. Als Konfiguration bekommt es mindestens einen Eintrag der Form *ADDR1/MASK1 [GW1] OUT1* übergeben. Ein Paket, dessen Zieladresse aus dem durch *ADDR1/MASK1* definierten Subnetz stammt, verlässt das Element somit auf Port *OUT1*. Sollte das Subnetz nicht direkt mit dem Router verbunden sein, kann mit *GW1* ein *Next Hop*, also ein weiterer Router, definiert werden. In der verwendeten Konfiguration werden Pakete, die ans Host-System adressiert sind auf Port 0 weiter geschickt. Die Ports 1, 2 und 3 entsprechen den Subnetzen *192.168.1.0/24*, *192.168.2.0/24* bzw. *192.168.3.0/24*. Port 4 ist für alle weiteren Pakete gedacht - diese werden allerdings verworfen. Mit dieser Information ist der Weg eine typischen IP-Pakets durch den Router leicht nachzuvollziehen. Der Classifier schickt ein IP-Paket zunächst an das *Strip*-Element, das den Ethernet-Header entfernt, danach entscheidet *StaticIPLookup* über den weiteren *Forwarding Path*. Hier durchläuft das Paket dann weitere Elemente, die das Dekrementieren der TTL oder die Fragmentierung von IP-Paketen übernehmen. Zum Schluss wird das Paket dann in einem Warteschlangenelement zwischengespeichert und schließlich über die gewählte Netzwerkschnittstelle versendet.

Diese Konfiguration kann auch mit klassischem Linux-Kernelrouting nachgestellt werden. Hier muss neben der Konfiguration der Routen noch darauf geachtet werden, dass das Weiterleiten von IP-Paketen aktiviert ist, damit der Rechner als Gateway fungiert.

5.2.2 Der Ethernet-Switch

Die zweite verwendete Konfiguration erfüllt die Aufgabe eine Layer-2-Switches und leitet Pakete somit auf Basis von Hardwareadressen weiter. Die Funktionsweise eines solchen Switches ist leicht zu verstehen. Vereinfacht gesagt verfügt ein Switch intern über eine dynamisch wachsende Liste, genannt *Source Address Table* (SAT). Diese SAT enthält Einträge der Form "Ethernetadresse Netzwerkschnittstelle", die zunächst allerdings leer ist. Empfängt der Switch nun ein Paket über eine Netzwerkschnittstelle, merkt er sich die Quell-MAC-Adresse des empfangenen Pakets sowie die zugehörige Schnittstelle in der SAT. Daraufhin wird die Ziel-MAC-Adresse des Pakets in der SAT gesucht. Hat dies keinen Erfolg, wird das Paket über alle Netzwerkinterfaces weitergeschickt. Befindet sich die Adresse allerdings bereits in der SAT, wird das Paket nur über die zugehörige Netzwerkschnittstelle verschickt. Durch dieses Vorgehen kann das Netzwerk deutlich entlastet werden. Ein solcher Switch lässt sich wie in Abbildung 5.2 dargestellt sehr leicht in Click konfigurieren.

```
1 fd1 :: FromDevice(eth0);
2 fd2 :: FromDevice(eth1);
3 switch :: EtherSwitch;
4 out1 :: Queue(1000) -> ToDevice(eth0);
5 out2 :: Queue(1000) -> ToDevice(eth1);
6
7 fd1 -> [0]switch[0] ->  out1;
8 fd2 -> [1]switch[1] -> out2;
```

Listing 5.2: Ethernet-Switch-Konfiguration

Das *Etherswitch*-Element erfüllt hier die Kernaufgabe, wobei das Element in der SAT den Elementport statt der Netzwerkschnittstelle speichert.

Neben dem generellen Unterschied in Bezug auf die Funktionsweise, sind an dieser Konfiguration auch deutlich weniger Elemente beteiligt. Während Pakete in der Konfiguration aus 5.2.1 von einer Vielzahl an Elementen verarbeitet werden, wird der gesamte Prozess der "Wegfindung" hier von einem einzigen Element erfüllt. Anhand von Messungen soll evaluiert werden inwiefern sich dieser verkürzte Elementpfad auf die Leistung des Routers auswirkt.

Um die Funktion eines solchen Switches mit Linux-Boardmitteln zu erreichen, wurde auf das Programm **brctl**[3] zurückgegriffen, das es erlaubt sog. *Bridges* zu erstellen, deren Funktionsweise der eines Switches entspricht. Einer solchen Bridge werden dann reale oder auch virtuelle Netzwerkschnittstellen als Eingangsport zugewiesen. Da Netzwerkkarten normalerweise nur solche Pakete zu einer Verarbeitung durch die CPU weiterreichen, deren Ziel-Hardwareadresse mit der eigenen übereinstimmt, müssen die beteiligten Netzwerkkarten im sog. *promiscuous mode* betrieben werden. Dadurch verwirft die Karte keine einkommenden Pakete mehr, sondern reicht sie unabhängig vom Ziel an die CPU weiter.

5.3 Iperf

Iperf[4] ist eine offene, in C++ programmierte Anwendung um Leistungsmessungen in Netzwerken durchzuführen. Iperf kann die Aufgabe eines Servers oder eines Clients, wahlweise im TCP- oder UDP-Modus erfüllen und die mögliche Datenrate zwischen zwei Punkten im Netzwerk ermitteln. Ein Host startet hierzu den Server während sich der andere als Client auf diesen verbindet. Im Anschluss sendet der Client Daten an den Server. Die Tests lassen sich mit verschiedenen Parametern durchführen - so kann bei TCP-Test beispielsweise die *Windowsize* variiert werden. Im UDP-Modus ist es möglich, die Größe der zu verschickenden Pakete ebenso zu definieren wie die gewünschte Datenrate. Letzteres wird dadurch erreicht, dass Iperf je nach gewünschter Datenrate *nanosleep*[5] nutzt, um eine kurze Zeitspanne zwischen dem Versenden der Pakete zu warten. Da *nanosleep* je nach Auflösung der zugrunde liegenden Uhr den Zeitparameter auf ein Vielfaches dieser Auflösung rundet, kann es hier zu Abweichungen zwischen der gewünschten und tatsächlichen Datenrate kommen. Zudem können aufgrund von Limitationen, die durch die Hardware vorgegeben sind natürlich nicht beliebig große Datenraten erreicht werden.

[3] http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge
[4] http://sourceforge.net/projects/iperf
[5] man 2 nanosleep

5.4 Flow-Control

Wenn Fast Ethernet und Gigabit-Ethernet im Vollduplexbetrieb arbeiten, sind sie nicht mehr auf das Zugriffsverfahren CSMA/CD angewiesen [7]. Da hier bei immer weiter steigenden Datenraten die Wahrscheinlichkeit steigt, dass eine Station die ankommenden Daten nicht mehr verarbeiten kann, ist das im Standard IEEE 802.3x spezifizierte Flow-Control-Verfahren entstanden. Dieses erlaubt es einer

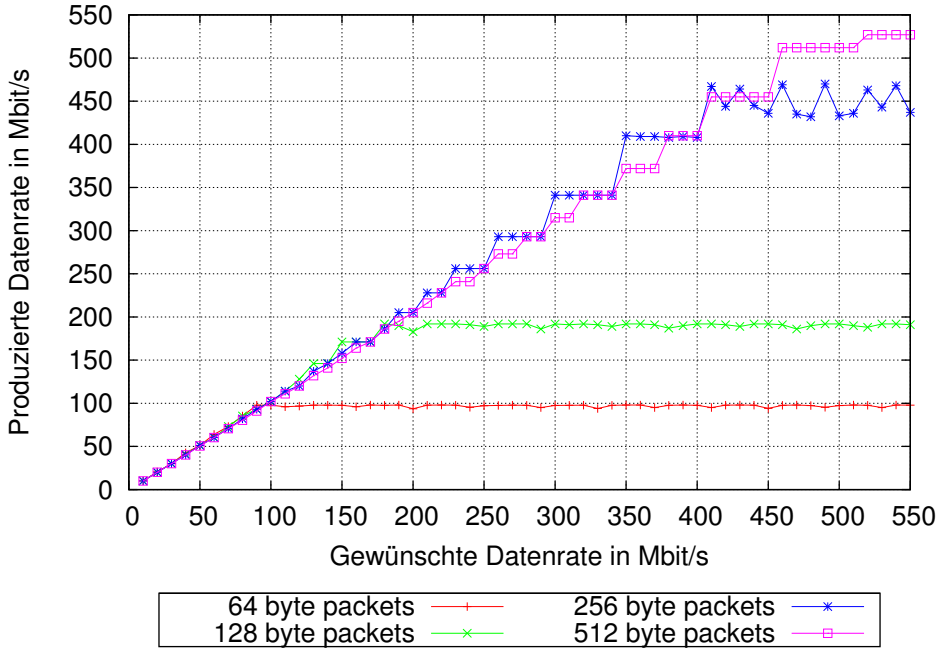


Abbildung 5.2: UDP-Messungen mit einer IP-Router-Konfiguration bei aktivierter Flow-Control

Station im Netzwerk einer anderen sog. Pause-Frames zu senden. Dabei handelt es sich um MAC-Control-Frames, die an die Multicast-Adresse 01-80-C2-00-00-01 geschickt werden. Neben dem Opcode 0x0001, der einen Pause-Frame als solchen kennzeichnet, enthält ein Pause-Frame zusätzlich ein zwei Byte langes Parameterfeld. Dieses enthält einen Wert, der die gewünschte Pause-Zeit als Vielfaches der Slot-Time angibt. Empfängt eine Station nun einen solchen Pause-Frame, stellt sie das Senden von Frames für die gewünschte Zeitspanne ein. Pause-Frames werden immer nur zwischen zwei direkt verbunden Geräten im Netzwerk ausgetauscht, da Switches Pause-Frames grundsätzlich nicht weiterleiten. Allerdings kann ein Switch selbst natürlich Pause-Frames versenden und empfangen. Da der

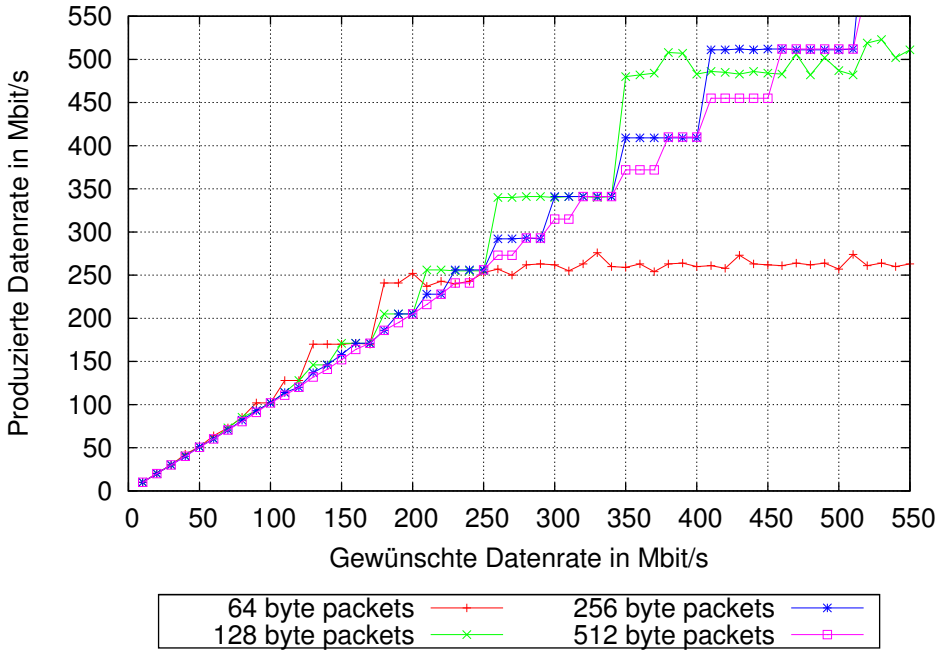


Abbildung 5.3: UDP-Messungen mit einer IP-Router-Konfiguration bei deaktivierter Flow-Control

Netzwerkkartentreiber unter eCos im Gegensatz zum Treiber unter Linux keine Flow-Control unterstützt, wurde diese bei den Test unter Linux mittels **ethtool**[6] deaktiviert. Abbildungen 5.2 und 5.3 zeigen, wie sich aktivierte bzw. deaktivierte Flow-Control auf die Datenraten auswirken. Für beide Messungen wurde der Testaufbau aus 5.1 mit der IP-Router-Konfiguration verwendet, die durch klassischen Linux-Kernelrouting realisiert wurde. Bei kleinen Paketgrößen fällt hier auf, dass mit aktivierter Flow-Control eine viel geringere Datenrate erreicht werden kann, da der Router das Versenden weiterer Frames durch das Versenden von Pause-Frames unterbindet.

5.5 Weitere Limitationen

Bei der im Router verwendeten Netzwerkkarte handelt es sich um eine 3-Port-Gigabit-Karte. Die theoretische Datenrate von einem Gigabit pro Sekunde kann damit allerdings nicht erreicht werden. Wenn der Router selbst an Verbindungs-

[6] http://www.kernel.org/pub/software/network/ethtool

endpunkt dient und nur von einem weiteren Rechner Daten empfängt, wurde unter Linux eine maximal mögliche Datenrate von 745 MBit/s mit Iperf ermittelt. Arbeitet der Router allerdings tatsächlich als Gateway, überträgt also Daten über mindesten 2 Ports gleichzeitig kann diese Datenrate nicht mehr erreicht werden. Hier wurde eine Spitzendatenrate von knapp 400 MBit/s gemessen. Der Grund dafür ist in der verwendeten Hardware selbst zu suchen. Während die Netzwerkkarte theoretisch für höhere Datenraten ausgelegt ist, ist die Anbindung über den PCI-Bus mit 32 Bit Busbreite und 33 MHz Taktung die limitierende Komponente des Systems. Die wirkt sich besonders dann aus, wenn Daten über mehr als eine Schnittstelle gesendet/empfangen werden, da die Daten aller Schnittstellen über den gleichen Bus transportiert werden müssen.

5.6 UDP-Messungen

Da das entscheidende Merkmal eines Routers der mögliche Datendurchsatz ist, wurden zu diesem Zweck Messungen mit UDP-Paketen verschiedener Größen bei verschiedenen Datenraten durchgeführt. UDP eignet sich aufgrund seines verbindungslosen Aufbaus für solche Messungen deutlich besser als TCP, da letzteres aufgrund seines verbindungsorientierten Aufbaus Mechanismen wie Flusskontrolle implementiert, die dem Protokoll eine dynamische Anpassung an das Übertragungsmedium und die beteiligten Teilnehmer erlaubt. UDP kennt solche Maßnahmen selbstverständlich nicht. Sollte eine Station hier mit dem Empfang der Pakete nicht mehr nachkommen, werden diese Pakete schlicht verworfen. Ein solcher Paketverlust ist leicht messbar. Im Folgenden sollen nun alle vier Implementierungen anhand gleicher Messungen verglichen werden. Hierzu wurde im Versuchsaufbau aus 5.1 auf einem Client ein Iperf-Server und auf dem anderen ein Iperf-Client gestartet. Der Client sendet nun jeweils UDP-Pakete einer festen Größe zwischen 64 und 1472 Bytes mit unterschiedlichen Datenraten, ausgehend von 10 MBit/s bis 550 Mbit/s. Aufgrund der schon aufgeführten Limitationen macht es keinen Sinn die Datenrate trotz der Gigabit-Karten weiter zu erhöhen. Sofern nicht anders angegeben, wurden alle folgenden Messungen mit der IP-Router-Konfiguration durchgeführt.

5.6.1 Linux-Kernel-Routing

Um einen Eindruck der möglichen Leistung des System zu erlangen wurden zunächst Messungen mit klassischen Linux-Kernel-Routing durchgeführt. Abbildung 5.4 zeigt, dass der Iperf-Client gerade bei kleinen Paketen mehr Pakete verschicken kann, als der Router bewältigen kann. Dies wird auch in Abbildung 5.5 deutlich. Die teilweise im Zickzack verlaufenden Linien sind das Ergebnis der Ungenauigkeiten, die bedingt durch den Einsatz von nanosleep bei Iperf auftreten können. Die Häufungen von Messpunkten treten je nach Art entweder auch aufgrund der eben genannten Limitationen auf, oder sind das Ergebnis der Tatsache, dass Iperf v.a. bei kleinen Paketen keine höheren Datenraten produzieren kann. Allgemein lässt sich hier feststellen, dass gerade viele kleine Pakete schon

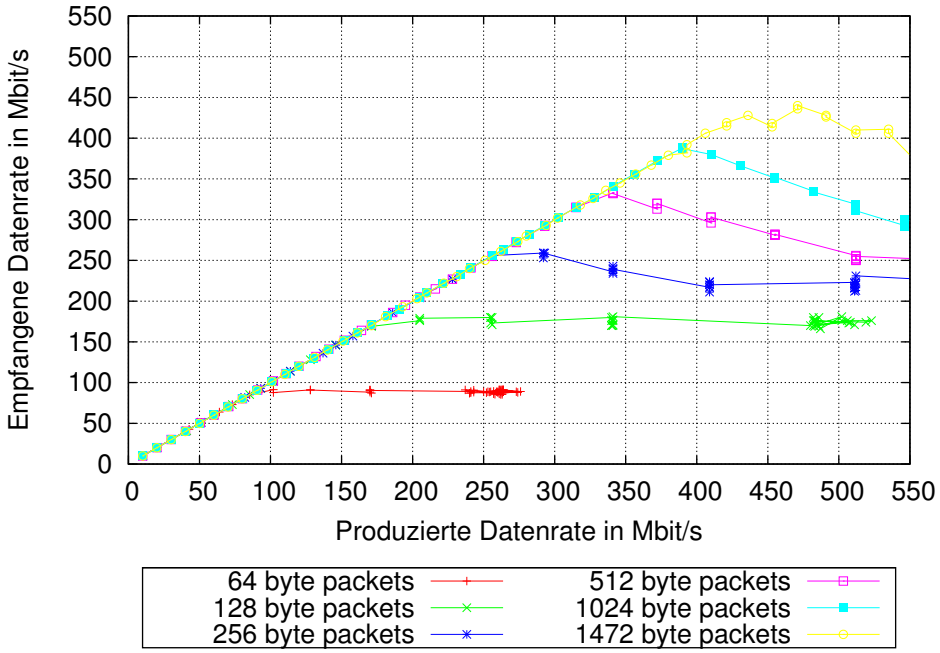


Abbildung 5.4: Produzierte Datenrate aufgetragen gegen die empfangene Datenrate (Linux-Kernel-Routing)

bei relativ niedrigen Datenraten zu hohen Paketverlusten führen. Dies ist auch nicht weiter verwunderlich, da zum Erreichen einer bestimmte Datenraten mit kleinen Paketen viel mehr Pakete verschickt werden müssen. Soll beispielsweise eine Datenrate von 100 MBit/s mit 64-Byte-Paketen erreicht werden, müssen dafür $\frac{100\frac{MBit}{s}}{64Byte*8} = 195313\frac{Pakete}{s}$ verschickt werden. Um die gleiche Datenrate mit

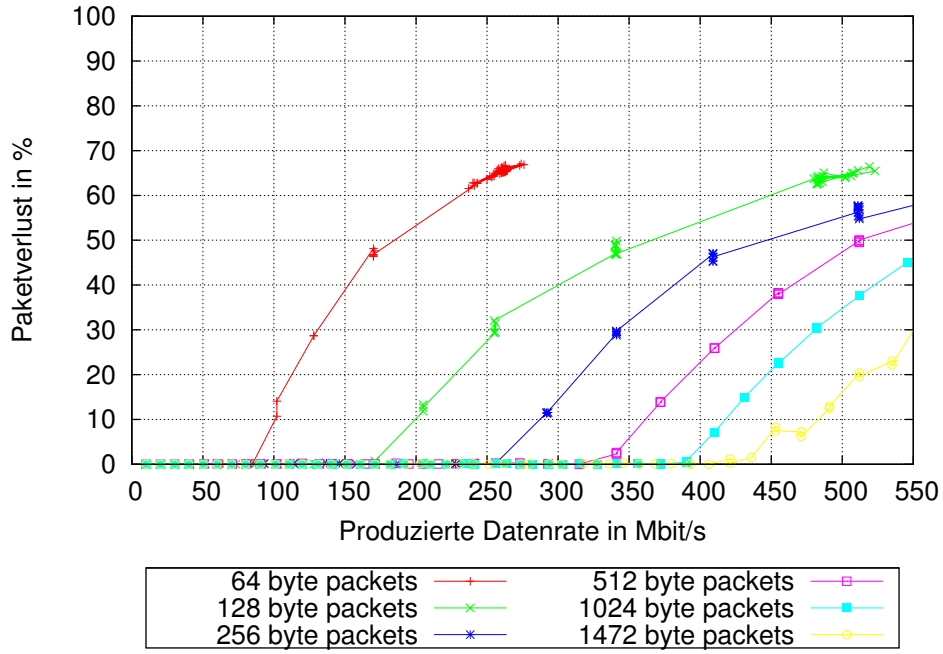


Abbildung 5.5: Paketverlustraten dargestellt für verschiedene Datenraten und Paketgrößen (Linux-Kernel-Routing)

1472-Byte-Pakete zu erreichen genügen jedoch schon $\frac{100\frac{MBit}{s}}{1472Byte*8} = 8492\frac{Pakete}{s}$. Dies wird in Abbildung 5.4 deutlich - bei geringen Datenraten sind produzierte und empfangene Datenrate gleich groß. Sobald aber eine gewisse Datenrate überschritten ist, weicht die empfange Datenrate von der gesendeten ab. Dieses Maximum verschiebt sich bei größeren Paketen hin zu höheren Datenraten.

5.6.2 Click im Userlevel-Modus

Der Userlevel-Modus von Click ist v.a. für das Testen neuer Konfigurationen und Elemente vorhergesehen. Pakete werden über RAW-Sockets empfangen, es wird also kein spezielles Kernelmodul verwendet, um direkt mit dem Kernel zu kommunizieren und Pakete so ggf. vor der Verarbeitung im Stack abzufangen. Obwohl dies gewisse Leistungseinbußen mit sich bringt wurden auch Messungen mit dieser Implementierung durchgeführt um Vergleiche mit Click im Linux-Kernelmodus durchführen zu können. Die übertragene Datenrate ist hier bei kleinen Paketen deutlich geringer als beim Linux-Kernel-Routing aus 5.6.1. Damit einher geht der sehr hohe Paketverlust der für 64-Byte-Pakete bei 70 MBit/s Datenrate

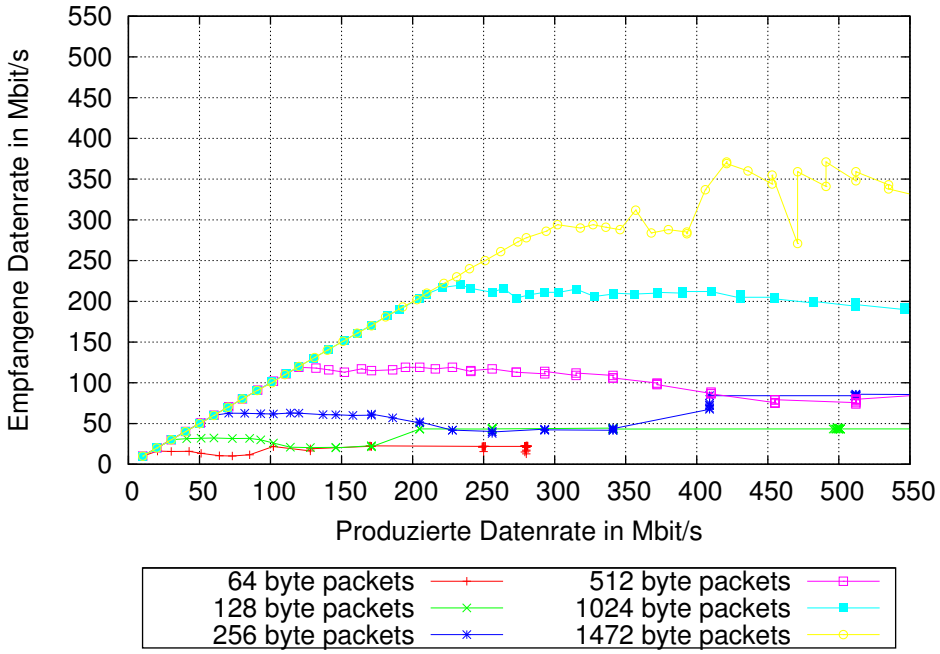


Abbildung 5.6: Produzierte Datenrate und die empfangene Datenrate (Click-Userlevel)

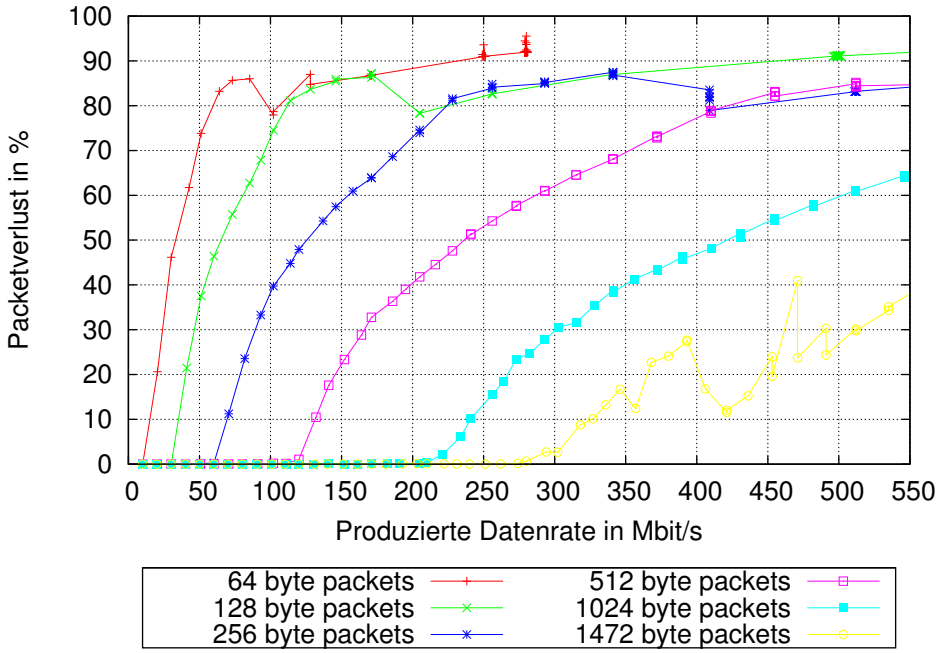


Abbildung 5.7: Paketverlustraten dargestellt für verschiedene Datenraten und Paketgrößen (Click-Userlevel)

schon fast sein Maximum von 90% erreicht hat. In Abbildung 5.6 sind, wie beim Linux-Kernel-Routing auch, die charakteristischen maximalen Datenraten für bestimmte Paketgrößen zu erkennen, allerdings findet hier eine Verschiebung hin zu geringeren Datenraten statt, d.h. die Gesamtleistung des Systems ist insgesamt als geringer einzustufen. Auffällig ist auch die teilweise starke Abweichung zweier benachbarter Punkte. Die Ursachen hierfür können vielfältig sein. So läuft Click als normaler Prozess, ist also dem Scheduling des Betriebssystems unterworfen. Wird Click nun die CPU entzogen und es kommen zu dieser Zeit viele Pakete an, kann der mit dem Empfangssocket assoziierte Puffer überlaufen, weil Click keine Pakete daraus entfernt. Dies führt dazu, dass weitere ankommende Pakete verworfen werden.

5.6.3 Clicks Linux-Kernelmodul

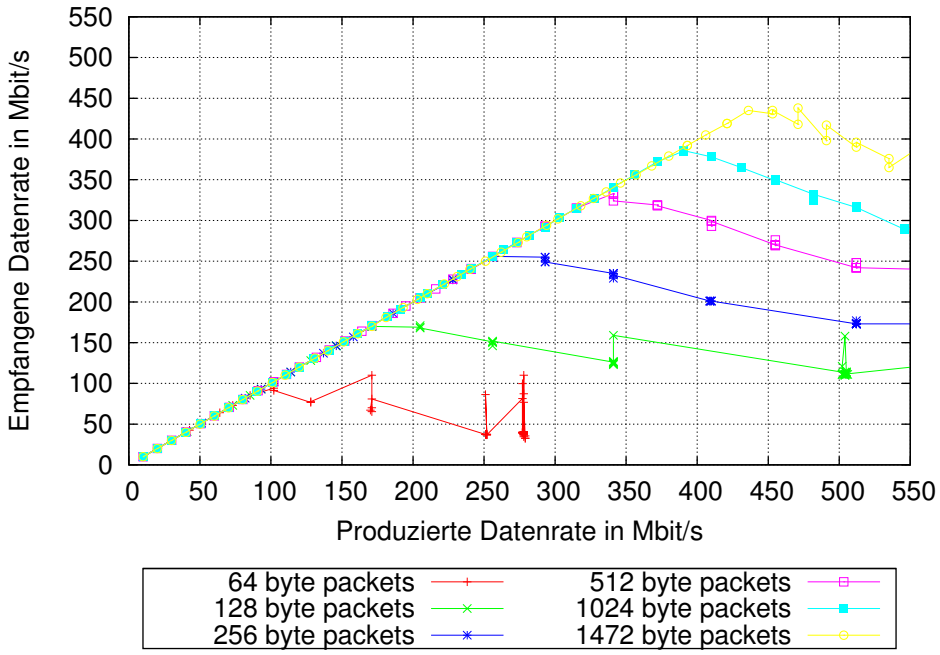


Abbildung 5.8: Produzierte Datenrate aufgetragen gegen die empfangene Datenrate (Click-Linux-Kernelmodul)

Clicks Linux-Kernelmodul stiehlt, ähnlich wie die eCos-Implementierung, ebenfalls Pakete vom Netzwerkstack und umgeht so die Bearbeitung der Pakete durch das Betriebssystem. Die Leistung ist verglichen mit der Userlevel-Implementierung

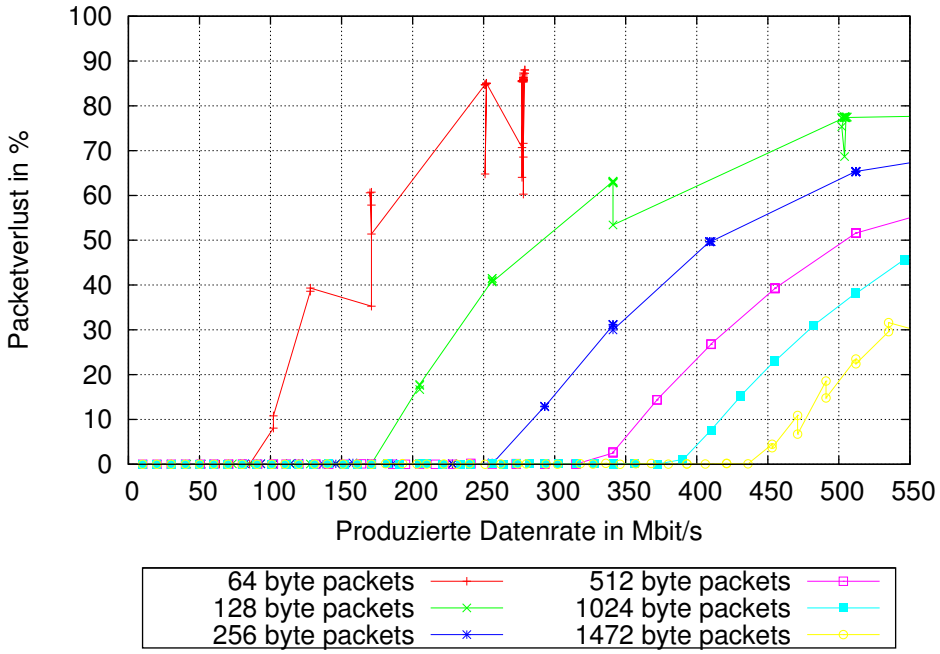


Abbildung 5.9: Paketverlustraten dargestellt für verschiedene Datenraten und Paketgrößen (Click-Linux-Kernelmodul)

deutlich besser - dies liegt v.a. am fehlenden Overhead der im Userlevel durch den Linux-Netzwerkstack verursacht wird. So können mit dem Click-Kernelmodul bei Verwendung von 256-Byte-Paketen bis zu 250 Mbit/s ohne Paketverluste übertragen werden - im Userlevel-Modus liegt diese Grenze mit 60 MBit/s deutlich darunter. Damit einher gehen die allgemein geringeren Paketverlustraten im Vergleich mit der Userlevel-Version. Allerdings kann das Click-Kernelmodul gerade bei sehr kleinen Paketgrößen auch nicht an nicht an das klassische Linux-Kernel-Routing heranreichen - so sind bei einer Paketgröße von 64 Bytes fast so große Paketverluste wie im Userlevel-Modus zu verzeichnen. Bei Paketgrößen ab 512 Byte sind die Unterschiede zwischen dem Linux-Kernel-Routing und Clicks Kernelmodul allerdings nur noch marginal.

5.6.4 Click unter eCos

Die vorherigen Messungen dienten dazu Vergleichsmessung sowie einen Überblick über die generelle mögliche Leistung des Systems geben. Die Evaluation des Click-Routers unter eCos soll nun folgen. Generell ist zu sagen, dass die Leis-

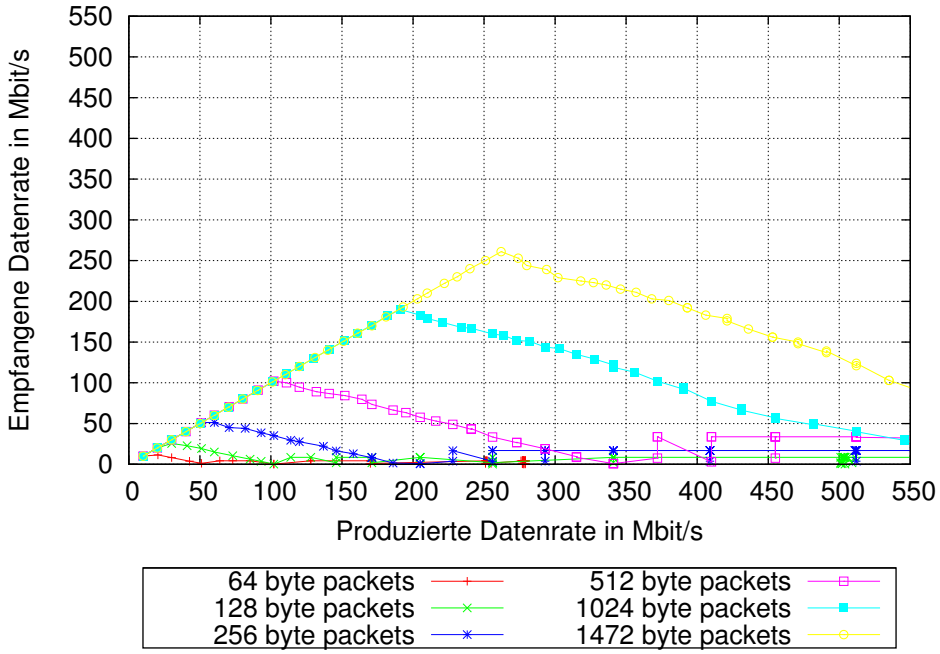


Abbildung 5.10: Produzierte Datenrate aufgetragen gegen die empfangene Datenrate (Click unter eCos)

tung der eCos-Implementierung weit hinter den bisher vorgestellten Versionen zurückbleibt. Im Vergleich mit den Alternativimplementierungen fällt in Abbildung 5.10 auf, dass die maximalen Datenraten für verschiedene Paketgrößen deutlich geringer ausfallen. Ebenfalls auffällig ist die Tatsache, dass die empfangenen Datenraten im Gegensatz zu den bisherigen Ergebnissen teilweise bis auf 0 MBit/s abfallen. Besonders kleine Datenpakete führen schon bei geringen Datenraten zu extrem hohen Paketverlusten. Im Gegensatz zur Userlevel-Implementierung wird allerdings ersichtlich, dass die Paketverlustrate bis auf nahezu 100% steigt und somit praktisch keine Paketweiterleitung mehr stattfindet. Mit wachsenden Paketgrößen verschieben sich diese Verluste hin zu den höheren Datenraten. Allerdings ist auch hier zu beobachten, dass die Paketverlustrate bei fast allen Paketgrößen 100% erreicht, sobald eine gewissen Datenrate überschritten wird. Um zu untersuchen ob die Ursache für diesen Umstand in Click selbst oder in eCos zu suchen ist, wurde eine Server-Client-Anwendung entwickelt, die ähnlich wie Iperf arbeitet. Der Server wurde als einzige Anwendung auf einem eCos-System auf dem Router ausgeführt. Auf einem direkt mit dem Router verbundenen Host wurde der Client gestartet, der eine feste Anzahl an Paketen bestimmter Größe

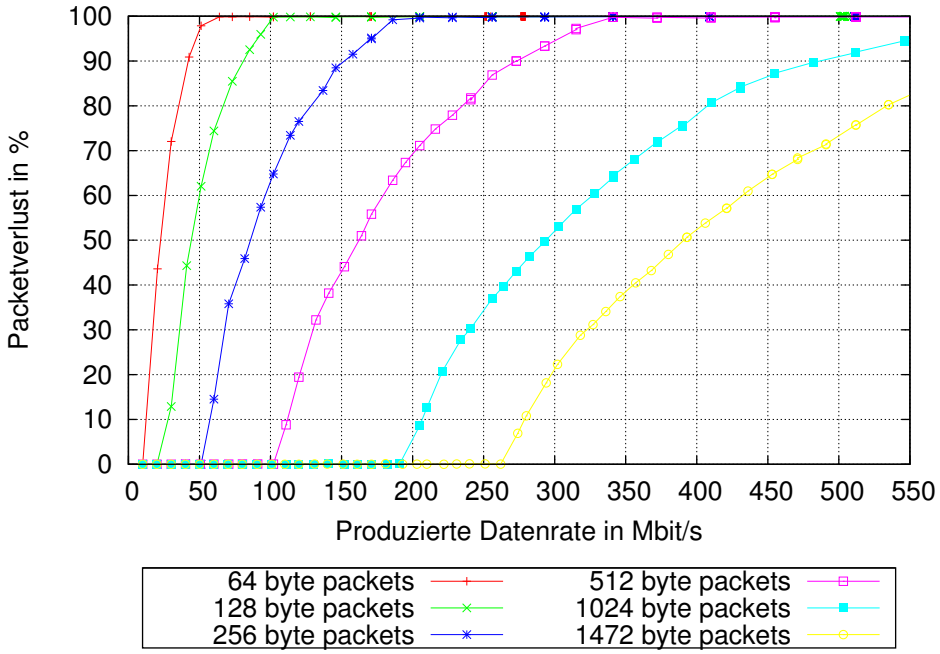


Abbildung 5.11: Paketverlustraten dargestellt für verschiedene Datenraten und Paketgrößen (Click unter eCos)

mit unterschiedlichen Datenraten an den Server schickt. Auf dem Server wurde im Anschluss die Anzahl der empfangen Pakete ausgegeben. Wie deutlich zu sehen ist, ist der Ergebnisgraph nahezu identisch mit Abbildung 5.11. Da der unter eCos zum Einsatz kommende Netzwerkkartentreiber im Bezug auf Leistung nicht optimiert ist, liegt die Vermutung nahe, dass dieser hier den limitierenden Faktor darstellt.

Um den Einfluss der Konfiguration auf die Leistung des Routers untersuchen zu können, wurden Messungen mit der in 5.2.2 vorgestellten Konfiguration durchgeführt. Abbildung 5.13 zeigt hier die Paketverlustraten bei Nutzung der Switch-Konfiguration. Die grau gezeichneten Linien bezeichnen hierbei die Werte, die für die IP-Router-Konfiguration gemessen wurden. Dies veranschaulicht den Einfluss, den die Konfiguration auf die Leistungsfähigkeit des Routers hat - mit wachsendem Elementpfad muss das System mehr Zeit für die Verarbeitung von Paketen aufbringen. Diese fehlende Zeit wirkt sich dann direkt auf die Verlustrate von Paketen aus.

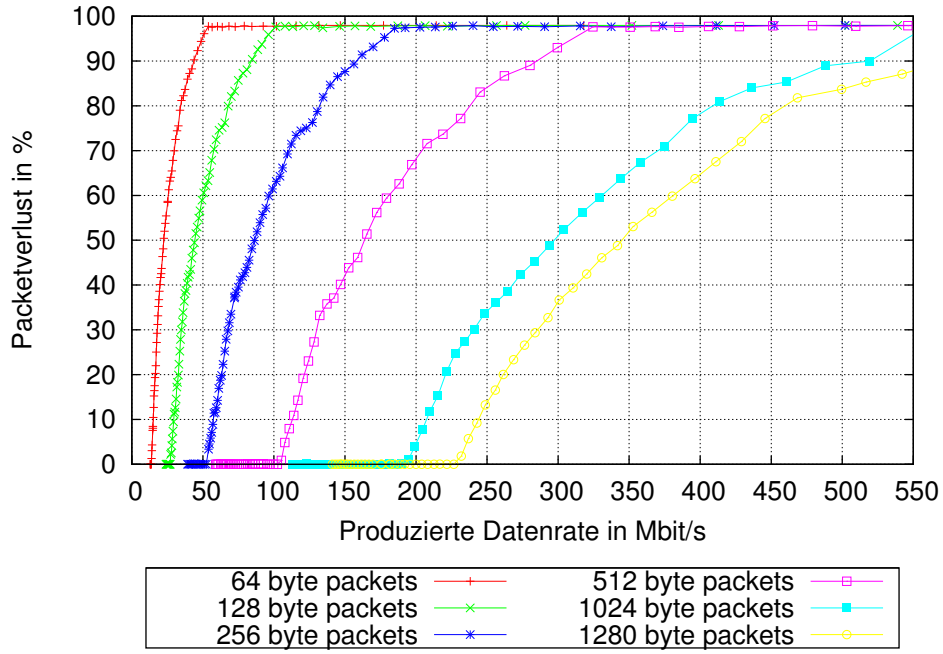


Abbildung 5.12: Ergebnisse der Server-Client-Anwendung mit einem UDP-Server als eCos-Anwendung und einem Client der auf einem Linux-System läuft

5.7 TCP-Messungen

TCP ist als verbindungsorientiertes Transportprotokoll viel komplexer aufgebaut als UDP. So versucht TCP die größtmögliche Übertragungsrate auf einem Pfad im Netzwerk zu erreichen, ohne dass es dabei zu Paketstaus an Flaschenhälsen des Übertragungswegs und damit zu Paketverlusten kommt. Um dies zu erreichen, werden bei TCP zwei Algorithmen eingesetzt, *Slow Start* und *Congestion Avoidance* [8]. Vereinfacht gesagt sorgt ersterer dafür, dass die Übertragung nach einem langsamen Start immer mehr beschleunigt wird, während letzterer dafür sorgt, dass diese Erhöhung gestoppt wird, wenn Paketverluste infolge von Paketstaus auftreten. Somit geben TCP-Messungen einen guten Überblick über die praktisch mögliche Übertragungsrate zwischen den Stationen eines Netzwerks. Wie Abbildung 5.14 zeigt, erreichen Linux-Kernel-Routing und Clicks Linux-modul fast die gleiche Leistung, während die Userlevel-Implementierung auch hier deutlich langsamer ist. Click unter eCos schafft knapp einen Wert von 200 MBit/s und ist damit von den hier präsentierten Lösungen die langsamste. Auch hier wurden Vergleichsmessungen mit anderen Konfigurationen durchgeführt um

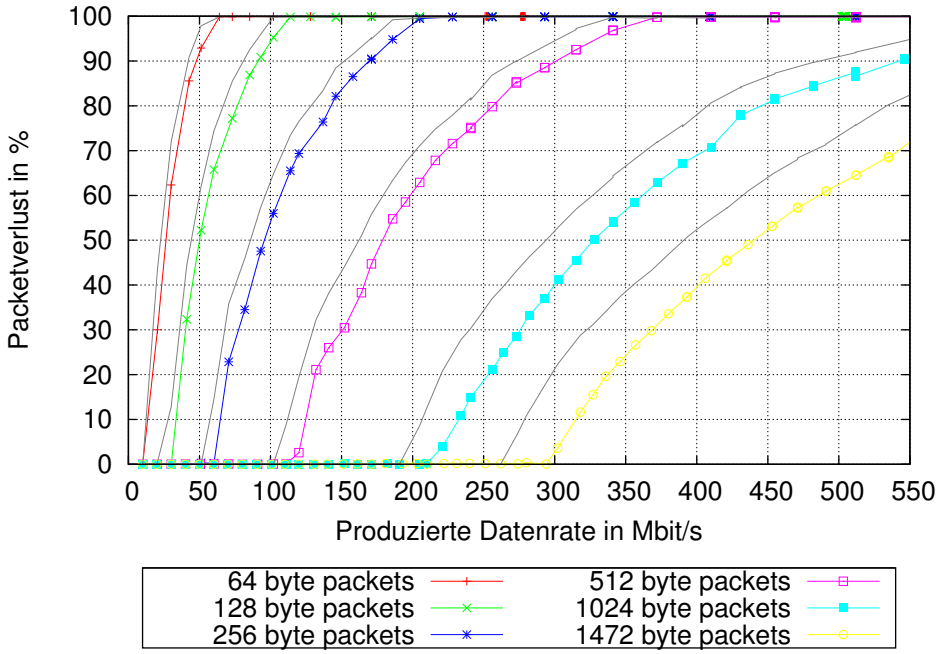


Abbildung 5.13: Paketverlustrate der Switch-Konfiguration im Vergleich zur IP-Router-Konfiguration (grau dargestellt)

den Einfluss der höheren Elementzahl auf die Leistung zu messen. Abbildung 5.15 zeigt die Ergebnisse für die IP-Router-Konfiguration. Die Ergebnisse für die Switch-Konfiguration sind für einen besseren Vergleich grau kariert eingezeichnet. Während die Userlevel-Version mit der IP-Router-Konfiguration deutlich langsamer ist, ist der Unterschied bei den im Kernel realisierten Lösungen nur marginal. Dass die eCos-Version hier mit 11,5 % im Vergleich zur Userlevel-Version mit 20,5 % einen nur geringen Leistungsverlust aufweist spricht für die Tatsache, dass Click im eCos-Router nicht die limitierende Komponente ist, sondern die Ursache im Treiber der der Netzwerkkarte zu suchen ist.

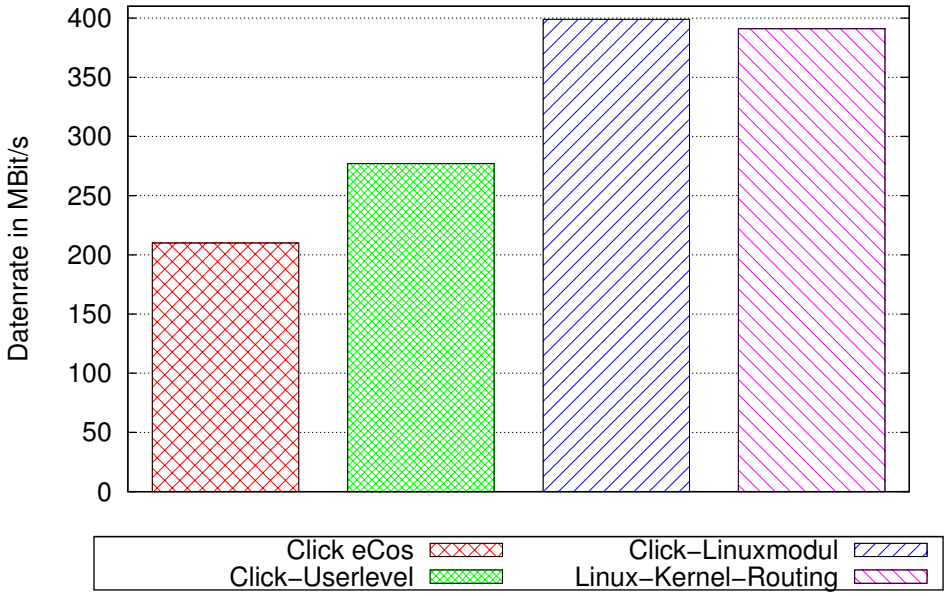


Abbildung 5.14: TCP-Leistung der verschiedenen Implementierungen mit der Switch-Konfiguration

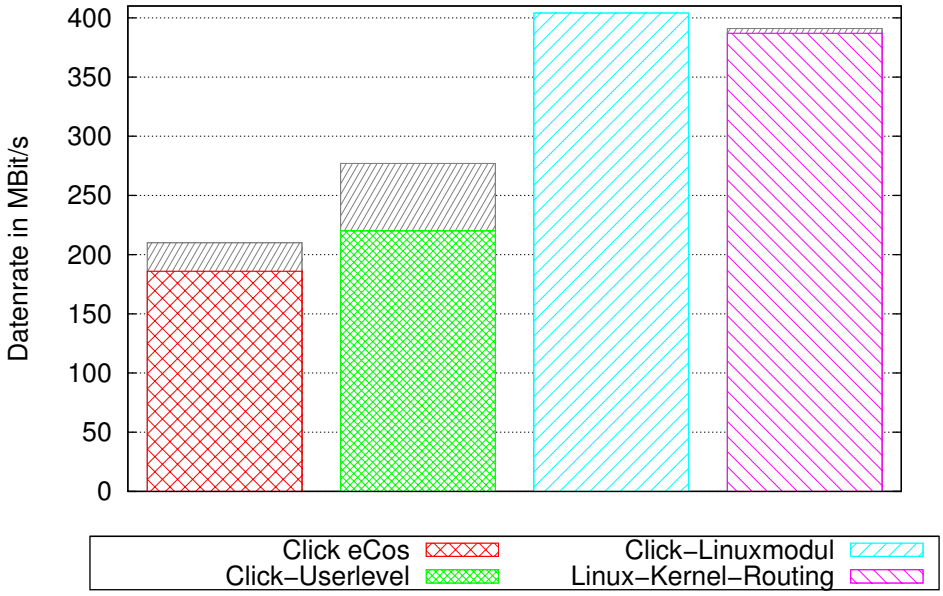


Abbildung 5.15: Switch-Konfiguration (grau) und IP-Router-Konfiguration im Vergleich

5.8 Zusammenfassung

Um die Leistungsfähigkeit des Click-Routers unter eCos beurteilen zu können, wurde ein Testaufbau genutzt, der es erlaubte die eCos-Implementierung mit Implementierungen auf Basis eines Linux-System zu vergleichen. Die angestellten Messungen ergaben, dass die eCos-Implementierung gerade im Bezug auf kleine Paketgrößen leistungstechnisch nicht an die andern Systeme heranreichen kann. Als möglicher Schwachpunkt wurde hier der Netzwerkkartentreiber für die verwendete Netzwerkkarte identifiziert. Durch die Verwendung zweier unterschiedlicher Konfigurationen lies sich neben dieser generellen Leistungsanalyse auch eine Abhängigkeit zwischen der Router-Konfiguration und der Leistung des Systems feststellen. So senken umfangreiche Konfigurationen aufgrund längerer Elementpfade den Paketdurchsatz.

6 Fazit

Ziel dieser Arbeit war die Portierung des modularen Click-Routers auf das Echtzeitbetriebssystem eCos. Es wurde geklärt, warum eCos sich im Vergleich mit anderen Projekten mit ähnlich Zielsetzung besser für die gestellte Aufgabe eignet. Ausschlaggebend war hier v.a. die Vielseitigkeit des Projekts gepaart mit dem überzeugenden Konzept der Elemente. Im weiteren Verlauf wurde die Anbindung des Routers an den Netzwerkstack von eCos beschrieben, sowie die notwendigen Änderungen, die am System vorgenommen werden mussten. Als Zusatz wurde ein Kontrollelement portiert, um dem Nutzer die Interaktion mit einem laufenden Click-Router zu ermöglichen. Die Evaluation der eCos-Implementierung und der Vergleich mit anderen Implementierungen auf Basis eines Linux-Systemen ergaben, dass die eCos-Implementierung leistungstechnisch nicht auf dem gleichen Niveau wie die anderen Systeme operieren kann. Als eine mögliche Schwachstelle wurde der Netzwerkkartentreiber in eCos identifiziert. Da sich das Gesamtsystem aber schon durch die Limitationen der Hardware nicht für den Einsatz in Gigabit-Netzen eignet, wäre ein Einsatz der eCos-basierten Lösung in Fast-Ethernet-Netzen aber durchaus denkbar. In Zukunft könnte die Überarbeitung des Treibers eine Leistungssteigerung mit sich bringen, die das System im Bezug auf die Anwendung in kleinen Netzwerken konkurrenzfähig macht.

Abkürzungsverzeichnis

AS	Autonomes System
BGP	Border Gateway Protocol
BIRD	BIRD Internet Routing Daemon
DSR	Deferred Service Routine
eCos	embedded Configurable operating system
HAL	Hardware Abstraction Layer
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ISP	Internet Service Provider
ISR	Interrupt Service Routine
LAN	Local Area Network
LGPL	GNU Lesser General Public License
NAT	Network Address Translation
OSPF	Open Shortest Path First
POSIX	Portable Operating System Interface
SAT	Source Address Table
SoHo	Small Office Home Office
TTL	Time To Live
URL	Uniform Resource Locator

XORP	eXtensible Open Router Platform
XRL	XORP Resource Locator

Literaturverzeichnis

[1] Andrew S. Tanenbaum. *Computernetzwerke*. Pearson Studium, 4., überarb. a. edition, 2003.

[2] Mark Handley, Orion Hodson, and Eddie Kohler. Xorp: an open platform for network research. *SIGCOMM Comput. Commun. Rev.*, 33:53–57, January 2003.

[3] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18:263–297, August 2000.

[4] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[5] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 3., aktualisierte auflage. edition, 2009.

[6] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[7] Jörg Rech. *Ethernet*. Heise Zeitschriften Verlag, GmbH & Co, KG, 2008.

[8] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

Abbildungsverzeichnis