technische universität
dortmund

Diplomarbeit

Entwicklung eines aspektorientierten TCP/IP-Stacks für eingebettete Systeme

Christoph Borchert
15. November 2010

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dr.-Ing. Daniel Lohmann

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 15. November 2010

Christoph Borchert

Zusammenfassung

Diese Diplomarbeit befasst sich mit der Entwicklung eines konfigurierbaren TCP/IP-Stacks für eingebettete Systeme. Da eingebettete Systeme in der Regel nur über sehr wenig Speicher verfügen, muss dieser von einem TCP/IP-Stack sehr effizient verwaltet werden. Je nach Anwendungsfall werden unterschiedliche Anforderungen an einen TCP/IP-Stack gestellt, so dass die Software hochgradig konfigurierbar sein sollte, um statisch optimal angepasst werden zu können. Die zurzeit existierenden Implementierungen wie *lwIP* und *uIP* realisieren dies durch den Einsatz des C-Präprozessors. Dadurch entstehen Probleme bzgl. der Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit des Quelltextes [1]. Im Rahmen der vorliegenden Diplomarbeit werden diese Probleme durch *aspektorientierte Programmierung* vermieden. Dazu wird ein TCP/IP-Stack unter Anwendung von aspektorientierten Entwurfsprinzipien, wie sie im Betriebssystem *CiAO* vorkommen, entworfen und implementiert. Die anschließende Integration des TCP/IP-Stacks in das CiAO-System ermöglicht die Evaluation der Performanz und des Speicherbedarfs in der Praxis.

Inhaltsverzeichnis

1 Einleitung

> „From Web browsers in cellular telephones to cafes with public wireless Internet access, from home networks with high-speed broadband access to traditional workplace IT infrastructure with a networked PC on every desk, to networked cars, to networked environmental sensors, to the interplanetary Internet—just as it seems that computer networks are essentially ubiquitous“ [2, S. 1].

Das Internet ist mittlerweile im alltäglichen Leben vieler Menschen präsent. Was vor 20 Jahren einigen forschenden Universitäten vorbehalten war, ist im Jahre 2010 für mehr als zwei Milliarden Menschen weltweit zugänglich [3, S. 4]. Die Vielfalt der Dienste des Internets ist in diesem Zeitraum enorm angestiegen. Das tägliche Abrufen und Verschicken von E-Mails, das gelegentliche Lesen der Wettervorhersage im World Wide Web und die jährliche elektronische Steuererklärung sind lediglich einige Beispiele. Der globale Informationsaustausch wurde durch das Internet in einer extrem kurzen Zeit revolutioniert. Doch wie geht es weiter?
Weiser und Brown postulieren, dass das Internet einen Übergang in eine neue Ära der Informationstechnik, dem *Ubiquitous Computing*[1] *(UC)*, einleitet:

> „UC will see the creation of *thin servers*[2], costing only tens of dollars or less, that put a full Internet server into every household appliance and piece of office equipment.“ [4, S. 77]

Alltägliche Vorgänge und Gegenstände des Menschen stehen beim Ubiquitous Computing im Vordergrund. Der Mensch wird in seinen täglichen Aufgaben durch vernetzte, „intelligente“ Mikrosysteme unterstützt, die in seiner Umgebung integriert sind. Beispielsweise könnten elektrische Haushaltsgeräte selbstständig ihren monatlichen Stromverbrauch auf einer Internetseite protokollieren, welche die monatlichen Stromkosten übersichtlich darstellt und von dem Eigentümer der Geräte einsehbar ist. Die Anzahl möglicher Einsatzgebiete für Ubiquitous Computing ist kaum überschaubar. Somit ist das Internet nicht nur auf Desktop-Rechner (PCs) und umfangreiche Serversysteme beschränkt, sondern findet ebenfalls in alltäglichen Gegenständen Einzug, in denen kleine „versteckte“, *eingebettete Systeme* vorhanden sind.

> „**Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres Produkt integriert sind**[2], und die normalerweise nicht direkt vom Benutzer wahrgenommen werden.“ [5, S. 1]

[1] Fachbegriffe werden in dieser Diplomarbeit bei ihrer ersten Verwendung *kursiv* hervorgehoben.
[2] Hervorhebungen im Original

Die technische Basis für Ubiquitous Computing stellen eingebettete Systeme dar, die sich deutlich von einem herkömmlichen PC unterscheiden. Auf der einen Seite müssen diese Systeme sehr preiswert sein, da sie lediglich ein Bestandteil eines Produktes sind und nicht, wie der PC, das Produkt selbst darstellen. Auf der anderen Seite besteht eine hohe Anforderung an einen minimalen Energieverbrauch von eingebetteten Systemen, welche häufig mit Batterien betrieben werden und aufgrund ihrer Miniaturisierung über keine aufwändige Kühlung verfügen. Ein typisches eingebettetes System ist ein *Mikrocontroller*, der zwei Euro kostet, 0,01 Watt bei vollständiger Auslastung verbraucht und 64 KByte Programmspeicher enthält. Die Software für ein derartiges System muss folglich andere Anforderungen erfüllen als die Software für einen PC. Wichtige Kriterien sind die Effizienz bzgl. der Laufzeiteigenschaften und ein minimaler Speicherbedarf.

1.1 Eingrenzende Zielsetzung und Relevanz

In der vorliegenden Diplomarbeit wird die Entwicklung der Netzwerksoftware für eingebettete Systeme beschrieben, die für eine Datenübertragung über das Internet notwendig ist. Mit dieser Software, die als *TCP/IP-Stack* bezeichnet wird, können unterschiedliche Systeme miteinander vernetzt und Informationen untereinander ausgetauscht werden. Ein TCP/IP-Stack für eingebettete Systeme stellt eine technische Grundlage des Ubiquitous Computings dar, das ohne vernetzte Mikrosysteme nicht existieren kann. Zur Erfüllung der o. g. Anforderungen an die Effizienz der Netzwerksoftware wird im Folgenden die Programmiersprache *C++* verwendet. Somit ist zusätzlich eine nahtlose Integration des entwickelten TCP/IP-Stacks in Betriebssysteme möglich, welche größtenteils ebenfalls in C++ oder *C* implementiert sind. Um flexibel in unterschiedlichen Anwendungsfällen des Ubiquitous Computings eingesetzt werden zu können, muss die Netzwerksoftware variabel sein und umfassende Konfigurationsmöglichkeiten bieten. Die Realisierung von Variabilität mit herkömmlichen Methoden der Programmiersprachen C und C++ führt in der Regel zu einer erhöhten Komplexität der Software. In dieser Diplomarbeit wird erstmalig ein hochgradig konfigurierbarer TCP/IP-Stack mit einer modernen softwaretechnischen Methode, der *aspektorientierten Programmierung* [6], entworfen und implementiert. Die Zielsetzung ist es, zu zeigen, dass durch die aspektorientierte Programmierung die Komplexität der konfigurierbaren Netzwerksoftware reduziert werden kann. Weiterhin wird überprüft, ob die entwickelte Software effizient bzgl. der Laufzeit und des Speicherbedarfs ist und ob diese Effizienz durch die aspektorientierte Programmierung beeinflusst wird.

1.2 Struktureller Aufbau der Arbeit

Nach der Einleitung erfolgt im zweiten Kapitel die Beschreibung des Konzeptes der Software-Produktlinie, das maßgebend für die weitere Gliederung der vorliegenden Diplomarbeit ist. Darin wird ein Referenzprozess der Softwareentwicklung vorgestellt, der in drei zentrale Phasen unterteilt ist. Diese Unterteilung entspricht der Struktur der folgenden drei Kapitel. Zunächst wird in Kapitel 3 die Analyse der Anforderungen an einen

konfigurierbaren TCP/IP-Stack für eingebettete Systeme betrachtet. Währenddessen wird existierende Netzwerksoftware untersucht, um die Analyse durch bereits vorhandenes Wissen zu ergänzen. Kapitel 4 baut darauf auf und erläutert den Entwurf des in dieser Diplomarbeit entwickelten TCP/IP-Stacks. Zuerst wird die Architektur der Software festgelegt, um anschließend deren Komponenten unter Anwendung der aspektorientierten Programmierung detailliert zu modellieren. Die Softwareentwicklung wird durch die Implementierung abgeschlossen, die in Kapitel 5 thematisiert wird. Wichtige Elemente und Algorithmen der Implementierung sowie die Integration des TCP/IP-Stacks in ein Betriebssystem werden behandelt.
Im nächsten Kapitel folgen die Diskussion und die Bewertung der entwickelten Netzwerksoftware. Sowohl die softwaretechnischen Eigenschaften des Quelltextes als auch die Performanz und der Speicherbedarf werden evaluiert. Diese Diplomarbeit schließt in Kapitel 7 mit einem Fazit und einem Ausblick über weiterführende Forschungsmöglichkeiten und Anwendungsgebiete ab.

2 Software-Produktlinien

Diese Diplomarbeit befasst sich mit der Entwicklung einer Software-Produktlinie. Das Prinzip der Produktlinie stellt ein zentrales Konzept der Diplomarbeit dar und wird aus diesem Grund zunächst in Abschnitt 2.1 allgemein erläutert. Anschließend wird das Konzept auf die Softwareentwicklung für eingebettete Systeme übertragen (siehe 2.2), bevor die Software-Produktlinienentwicklung durch einen Referenzprozess detailliert beschrieben wird (siehe 2.3). Die Implementierung einer Software-Produktlinie erfordert spezialisierte softwaretechnische Methoden, welche in Abschnitt 2.4 exemplarisch vorgestellt werden. Abschließend wird eine Software-Produktlinie für eingebettete Systeme anhand der CiAO Betriebssystemfamilie veranschaulicht (siehe 2.5), die im weiteren Verlauf dieser Diplomarbeit verwendet wird.

2.1 Erläuterung des Begriffs Produktlinie

Eine Produktlinie ist eine Menge von verschiedenen Produkten, die auf einer gemeinsamen Grundlage basieren. Neben dieser Gemeinsamkeit zeichnen sich die einzelnen Produkte einer Produktlinie durch zusätzliche Eigenschaften aus, die nicht jedes Produkt besitzt. Ein Beispiel für eine Produktlinie bietet die Automobilindustrie. Dort wird für verschiedene Fahrzeuge eine gemeinsame *Plattform* verwendet, die dann mit weiteren Komponenten ausgestattet wird. „Schon mit Einführung des Golf IV wurde die so genannte Plattform PQ 34 auch im Škoda Octavia, Audi TT, Audi A3, Seat Leon, Seat Toledo und im VW New Beetle verbaut“ [7, S. 109]. Eine solche Vorgehensweise hat den Vorteil, dass individuelle Kundenwünsche günstiger realisiert werden können, da bei der Entwicklung eines neuen Produktes auf die Plattform und bereits vorhandene Komponenten zurückgegriffen werden kann. Es müssen ausschließlich diejenigen Teile neu entworfen und produziert werden, die bisher in keinem anderen Produkt der Produktlinie existieren.

2.2 Software-Produktlinien für eingebettete Systeme

Das Prinzip der Produktlinie wird bei der herkömmlichen Softwareentwicklung für eingebettete Systeme in der Regel nicht genutzt. Stattdessen wird die Software für ein eingebettetes System eigenständig entwickelt, um optimal auf genau dieses System angepasst zu sein. Somit wird sichergestellt, dass keine unnötigen Laufzeit- und Speicherkosten von der Software verursacht werden. Ein Mikrocontroller mit extrem wenig Speicher kann z. B. für die Ausführung der Software eingesetzt werden und damit die monetären Kosten der Hardware reduzieren. Die Einsparung von nur einem Euro pro Mikrocontroller,

die durch den geringen Speicherverbrauch der entwickelten Software erreicht werden kann, führt bei einer Massenproduktion zu enormer Kostensenkung. Der vollständige Entwurf und die Implementierung von neuen Softwareprodukten führt allerdings zu langen Entwicklungszeiten und damit verbunden ebenfalls zu hohem Personalaufwand. Eine Softwareentwicklung für eingebettete Systeme nach dem Prinzip der Produktlinie kann diese Probleme lösen. Anstatt der Entwicklung von separaten Produkten mit ähnlicher Funktionalität wird eine gemeinsame Plattform entworfen, aus der unterschiedliche Softwareprodukte abgeleitet werden können. Die Entwicklung von redundanten Systemen wird durch eine *Software-Produktlinie* vermieden. Mit softwaretechnischen Methoden (siehe 2.4) kann sichergestellt werden, dass die abgeleiteten Produkte der Software-Produktline effizient bzgl. des Ressourcenbedarfs sind und minimale Speicher- und Laufzeitkosten aufweisen. Im folgenden Abschnitt wird auf die Vorgehensweise bei der Entwicklung einer Software-Produktline detailliert eingegangen.

2.3 Referenzprozess der Software-Produktlinienentwicklung

Das Ziel einer herkömmlichen Softwareentwicklung bezieht sich im Gegensatz zur Entwicklung einer Software-Produktlinie lediglich auf ein einzelnes Produkt. Die durchgeführten Prozesse dieser Softwareentwicklung eignen sich aufgrund dessen nicht für die Entwicklung einer Software-Produktlinie mit einer Fülle von resultierenden Produkten. Eine Produktlinienentwicklung erfordert spezielle Methoden, die in dem *Referenzprozess der Software-Produktlinienentwicklung* (siehe Abbildung 2.1) beschrieben sind. Der wichtigste Unterschied gegenüber der herkömmlichen Softwareentwicklung liegt in einer „Trennung von Domain und Application Engineering“ [8, S. 4]. Ergebnis des *Domain Engineerings* ist die Plattform der Produktlinie, die aus Referenzanforderungen, einer Referenzarchitektur und wiederverwendbaren Komponenten besteht. Clements und Northrop definieren eine Domäne wie folgt: „A domain is a specialized body of knowledge, an area of expertise, or a collection of related functionality“ [9, S. 14]. Beim Domain Engineering wird Wissen in der Form aufgearbeitet, dass es in jedem Produkt der Produktlinie genutzt werden kann. Das *Application Engineering* hingegen bezieht sich auf die Erstellung eines konkreten Produktes. Dabei wird soweit wie möglich auf die Plattform zurückgegriffen, indem die darin enthaltenen Komponenten verwendet werden. Die Programmierung beim Application Engineering kann auf ein Minimum reduziert werden, indem nur die Funktionalität neu entwickelt wird, die nicht in der Plattform enthalten ist.
In dieser Diplomarbeit liegt der Schwerpunkt auf der Entwicklung einer Plattform für eine Produktlinie, so dass im weiteren Verlauf ausschließlich das Domain Engineering betrachtet wird. Dabei werden die Domänenanalyse, der Domänenentwurf und die Domänenimplementierung durchlaufen. Die *Domänenanalyse* legt zuerst fest, welchen Bereich die Produktlinie abdecken soll, d. h. welche konkreten Produkte unterstützt werden sollen. Dieser Vorgang wird als *Scoping* bezeichnet. „Scoping identifies the commonality that members share and the ways in which they vary“ [9, S. 183]. Das Scoping wird

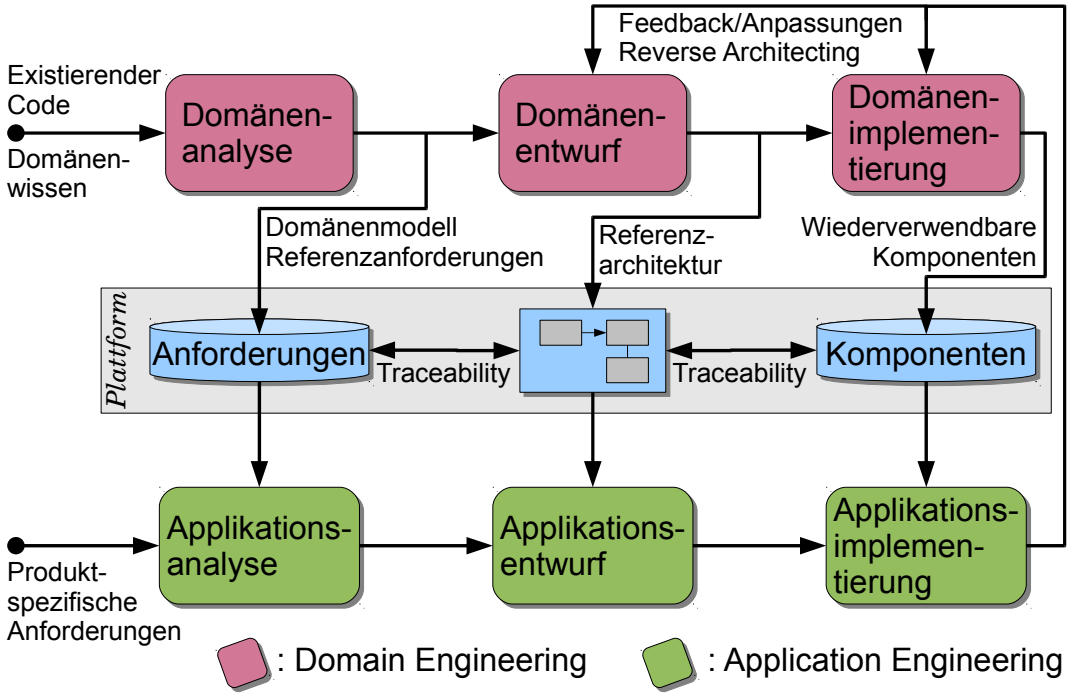

Abbildung 2.1: Referenzprozess der Software-Produktlinienentwicklung [8, S. 5]

durch *Konzept- und Merkmalmodelle* dokumentiert. Konzeptmodelle dienen der Erfassung wichtiger Eigenschaften der Domäne und können z. B. als *UML*[1]-Diagramme dargestellt werden. Im Gegensatz dazu beschreiben Merkmalmodelle die gemeinsamen und variablen Funktionalitäten der Produktlinie anhand von *Merkmaldiagrammen* [10, S. 87ff] und ergänzenden Erläuterungen. Die in der Domänenanalyse verwendeten Fachbegriffe der Domäne können in einem Domänenlexikon erklärt werden. Ein *Domänenmodell* [10, S. 23f] umfasst die Referenzanforderungen an die Produktlinie, die durch die o. g. Elemente definiert werden und stellt das Ergebnis der Domänenanalyse dar.
Auf der Basis des Domänenmodells beginnt der *Domänenentwurf*, dessen Ziel die Referenzarchitektur der Produktlinie ist. Dabei müssen zuerst die Anforderungen an diese Architektur analysiert werden, um im Anschluss dessen Modellierung vorzunehmen. Die Berücksichtigung von bekannten Entwurfs- und Architekturmustern (*„patterns“*) kann die Domänenmodellierung enorm vereinfachen und dessen Dokumentation veranschaulichen. „In a domain, patterns support developers in the selection of suitable software architectures [...] without wasting time and effort implementing solutions that are known to be inefficient, error-prone, unmaintainable, or simply broken“ [11, S. 12].
In der *Domänenimplementierung* werden letztendlich die wiederverwendbaren Softwarekomponenten der Produktlinie programmiert. Die Realisierung der Variabilität dieser Komponenten kann durch unterschiedliche Methoden erreicht werden, die in dem nachfolgenden Abschnitt erläutert werden.

[1] Unified Modelling Language

2.4 Implementierung von Konfigurierbarkeit

Variable und wiederverwendbare Software kann durch vielfältige Methoden erstellt werden. Diese Methoden lassen sich in zwei Kategorien einteilen. Die *statische Konfigurierung* bezeichnet den Ansatz, durch den der Funktionsumfang der Software spätestens zum Zeitpunkt der Kompilierung festgelegt wird. Nicht benötigte Komponenten werden zu diesem Zeitpunkt entfernt, so dass nach der Kompilierung im Idealfall keine Kosten dafür anfallen. *Dynamische Konfigurierung* stellt die konträre Vorgehensweise dar. Erst zur Laufzeit der Software wird entschieden, welche Funktionalität genutzt werden soll. Damit wird zum einen eine größere Flexibilität gegenüber der statischen Konfigurierung erreicht, zum anderen entstehen aber höhere Laufzeit- und Speicherkosten. In den nachfolgenden Abschnitten 2.4.1 und 2.4.2 werden verschiedene Varianten der statischen und dynamischen Konfigurierung vorgestellt.

2.4.1 Varianten der statischen Konfigurierung

Für die Implementierung von statisch konfigurierbarer Software gibt es unterschiedliche Prinzipien, die individuelle Vor- und Nachteile aufweisen. Nicht jedes dieser Prinzipien lässt sich in allen Situationen sinnvoll einsetzen. Manchmal ermöglicht erst eine Kombination von verschiedenen Varianten eine optimale statische Konfigurierung. Als Varianten werden der C-Präprozessor, Compileroptimierungen, Templates und aspektorientierte Programmierung vorgestellt.

C-Präprozessor

Die wahrscheinlich am weitesten verbreitete Methode der statischen Konfigurierung wird durch den *C-Präprozessor* [12] ermöglicht. Dabei handelt es sich um ein Werkzeug zur Textersetzung, das dazu benutzt wird, Programmcode an geeigneten Stellen einzufügen, zu ersetzen und zu manipulieren. Damit lässt sich die *bedingte Übersetzung* (`#ifdef`) realisieren, d. h. es können Softwarekomponenten vor der Kompilierung an- und abgewählt werden. Eine weitere Möglichkeit der Benutzung des C-Präprozessors stellen *Makros* (`#define`) dar, die vordefinierten Programmcode an spezifizierten Stellen einfügen.
Der Einsatz des C-Präprozessors zur statischen Konfigurierung bringt jedoch erhebliche Nachteile mit sich, u. a. Probleme bzgl. der Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit des Quelltextes [1]. Verursacht wird dies zum größten Teil dadurch, dass der C-Präprozessor unabhängig von der verwendeten Programmiersprache arbeitet und somit weder Syntax noch Semantik einer solchen beachtet.

Compileroptimierungen

Optimierende Compiler erlauben eine teilweise statische Konfigurierung bereits mit den Mitteln der eingesetzten Programmiersprache, ohne auf zusätzliche Werkzeuge wie einen Präprozessor zurückzugreifen. Mit *Constant Propagation* und *Unreachable-Code Elimination* [13, S. 379ff] ist es möglich, den Kontrollfluss eines Programms effizient

zu beeinflussen. Eine Fallunterscheidung über einen konstanten Ausdruck wird vom Compiler entfernt und durch den Teil davon ersetzt, der gemäß des Wahrheitswerts der Fallunterscheidung ausgeführt werden soll. Außerdem können alle Programmfunktionen, die nicht referenziert werden, problemlos durch *Function-Level Linking* entfernt werden. Die statische Konfigurierung mit dem Compiler bietet weniger Möglichkeiten als z. B. ein Präprozessor und eignet sich daher nicht als alleinstehende Methode zur Produktlinienentwicklung, sondern muss mit weiteren Methoden kombiniert werden.

Templates

Die Programmiersprache C++ bietet mit *Templates* ein mächtiges Mittel zur statischen Konfigurierung. Auf der einen Seite werden Templates für die *generische Programmierung* [14] benötigt, wie z. B. in der *STL*[2]. So lassen sich wiederverwendbare Datenstrukturen und Algorithmen implementieren und redundante Quelltexte vermeiden. Auf der anderen Seite gibt es mit der *statischen Template Metaprogrammierung* [10, S. 397ff] eine erweiterte Anwendung von Templates zur gezielten Codegenerierung durch den Compiler. Dabei werden Templates durch Spezialisierung und Rekursion wie eine funktionale Programmiersprache benutzt, die vom Compiler ausgeführt wird, um beliebige Berechnungen zu machen. „Surprisingly, templates together with a number of other C++ features constitute a Turing-complete, compile-time sublanguage of C++“ [10, S. 406]. Theoretisch eignen sich daher Templates hervorragend für die Implementierung von Produktlinien. Allerdings sind Metaprogramme in der Praxis deutlich komplexer als herkömmliche Programme, so dass die Lesbarkeit und Wartbarkeit erhöhten Aufwand erfordert.

Aspektorientierte Programmierung

Mit der *aspektorientierten Programmierung* (*AOP*) existiert eine konzeptionelle Erweiterung gängiger Programmierparadigmen. Ein Problem prozeduraler und objektorientierter Programmiersprachen ist eine mangelnde „separation of concerns[3]“ [15, S. 211]. Wenn ein Merkmal viele Komponenten eines Systems betrifft, so kann es als *querschneidender Belang* (*„crosscutting concern“*) bezeichnet werden. Ein querschneidender Belang lässt sich in der Regel nicht separat implementieren. Vielmehr ist die Implementierung solcher Belange über alle betroffenen Komponenten des Systems verstreut. „AOP is focused on mechanisms for simplifying the realization of such crosscutting concerns“ [6, S. 30]. Querschneidende Belange können mittels AOP durch sogenannte *Aspekte* als eigenständige Komponenten entworfen und implementiert werden. Dadurch wird ein enorm hohes Potential für Wiederverwendbarkeit und Konfigurierbarkeit geschaffen.
Die Realisierung von AOP kann auf unterschiedliche Weise erfolgen, wobei eine Erweiterung vorhandener Programmiersprachen um AOP-Elemente vielversprechend erscheint. Eine Spracherweiterung erfordert allerdings die Unterstützung durch einen Compiler.

[2] Standard Template Library
[3] *Trennung der Belange*

Mit *AspectC++* [16] gibt es eine aspektorientierte Erweiterung der Programmiersprache C++ inklusive einem frei verfügbaren Compiler. Ein Aspekt besteht dabei aus einem oder mehreren *Advices*. Dies sind Anweisungen, die den Kontrollfluss eines Programms an einem definierten Punkt, einem sogenannten *Join-Point*, beeinflussen. Ein Join-Point stellt in der Regel den Aufruf („`call`“) oder die Ausführung („`execution`“) einer Funktion bzw. Methode in einer prozeduralen oder objektorientierten Programmiersprache dar. Eine weitere Differenzierung kann durch den Zeitpunkt, zu welchem ein Advice aktiv werden soll, geschehen. Mögliche Zeitpunkte sind vor („`before`“), während („`around`“) und nach („`after`“) einem Aufruf oder Ausführung einer Funktion. Die Spezifikation, welche Funktionen bzw. Methoden von einem Advice beeinflusst werden sollen, wird durch einen *Pointcut*-Ausdruck festgelegt. Dieser entspricht im Wesentlichen der Signatur der Funktion bzw. Methode und kann die Platzhalter „`%`“ und „`...`“ für beliebige Zeichenketten enthalten. Die Bestimmung des Durchschnitts, der Vereinigung und der Inversen von Pointcut-Ausdrücken ist ebenfalls möglich. Neben der Beeinflussung des Kontrollflusses durch einen Advice kann mit einem Klassenfragment („`slice`“) die Struktur einer Klasse um beispielsweise neue Attribute und Methoden erweitert werden. Die Festlegung, welche Klassen genau erweitert werden, erfolgt ebenfalls über einen Pointcut-Ausdruck. Anschauliche Beispiele für Aspekte befinden sich in den Abschnitten 4.4, 4.5 und 5.2. Für eine vollständige Beschreibung der Sprache AspectC++ wird an dieser Stelle aufgrund des beschränkten Umfangs dieser Diplomarbeit auf Spinczyk und Lohmann [16, S. 6ff] verwiesen. Der o. g. AspectC++-Compiler verarbeitet sowohl C++-Code als auch separaten Aspekt-Code und erzeugt daraus wiederum standardkonformen C++-Code, der von gängigen Compilern in Maschinencode übersetzt werden kann. Dieser Vorgang wird als *Aspektweben* bezeichnet, d. h. Aspekte werden in vorhandenen Quelltext „eingewebt“. Nach dem Aspektweben können weiterhin die drei bisher vorgestellten Varianten der statischen Konfigurierung ergänzend aktiv werden. Mit diesem Werkzeug lässt sich AOP zur statischen Konfigurierung von Software problemlos einsetzen, die Anwendung zur dynamischen Konfigurierung wird im nächsten Abschnitt erläutert.

2.4.2 Varianten der dynamischen Konfigurierung

Die dynamische Konfigurierung von Software erfordert andere Methoden und Werkzeuge als die statische Konfigurierung und wird häufig durch eine Indirektion umgesetzt. Das kann im einfachsten Fall ein Funktionsaufruf über einen Funktionszeiger sein, der zur Programmlaufzeit verändert wird und somit die gewünschte Funktion referenziert. Nachfolgend wird der Einsatz der objektorientierten Programmierung zur Implementierung von dynamischer Konfigurierbarkeit beschrieben. Anschließend wird das dynamische Aspektweben vorgestellt.

Objektorientierte Programmierung

Objektorientierte Programmiersprachen ermöglichen mit dem Mechanismus der *Polymorphie* eine teilweise dynamische Konfigurierung. Dabei wird zur Programmlaufzeit aufgrund des Objekttyps entschieden, welcher Programmcode ausgeführt werden soll.

Anwendung findet dieser Mechanismus in vielen bekannten Entwurfsmustern, wie z. B. dem der *Strategie* [17, S. 373]. Technisch wird die Polymorphie durch eine Indirektion umgesetzt, indem eine *virtuelle Funktionstabelle (vtable)* angelegt wird, anhand der zur Laufzeit die auszuführenden Funktionen ermittelt werden. Die Variabilität der Software bei dieser Methode bringt dadurch nicht vernachlässigbare Speicher- und Laufzeitkosten mit sich, so dass ihr Einsatz in eingebetteten Systemen stets genau überprüft werden sollte.

Dynamisches Aspektweben

Die aspektorientierte Programmierung (siehe 2.4.1) lässt sich auch zur dynamischen Konfigurierung verwenden. Dabei wird die Software in der Regel vor und während der Kompilierung geeignet präpariert, um zur Laufzeit durch Aspekte erweitert werden zu können. Diesen Vorgang bezeichnet man als *dynamisches Aspektweben*. Dabei treten neue Herausforderungen gegenüber dem statischen Aspektweben auf, wie z. B. die Wirkung von Aspekten auf andere Aspekte, die erst zu einem späteren Zeitpunkt aktiviert werden. *TOSKANA* [18] und *Dynamic AspectC++* [19] sind zwei Systeme, die dynamisches Aspektweben unterstützen. Beide Systeme setzen für das Laden der Aspekte Betriebssystemmittel voraus und verursachen zusätzliche Kosten für Speicher und Laufzeit. Aus diesen Gründen wird in dieser Diplomarbeit auf deren Einsatz verzichtet.

2.5 Die CiAO Betriebssystemfamilie

Die konsequente Anwendung von AOP zur statischen Konfigurierung einer Produktlinie demonstriert *CiAO*[4]. Dabei handelt es sich um eine hochgradig konfigurierbare Betriebssystemfamilie für eingebettete Systeme gemäß dem *AUTOSAR OS Standard* [20]. Ursprünglich für die TriCore Plattform von Infineon entworfen, befinden sich mittlerweile Portierungen für weitere Architekturen, u. a. IA-32, AMD64, ARM und MicroBlaze, in der Entwicklung. Lohmann u.a. [1] zeigen, dass die nachträgliche Anwendung von AOP in bereits existierenden Systemen nicht zu optimalen Ergebnissen führt, wenn diese nicht dafür entworfen sind. Aus diesem Grund wird bei CiAO eine völlig neue Philosophie verfolgt: „The idea was to build an operating system in an aspect-oriented way from scratch, considering AOP and its mechanisms *from the very beginning*[5] of the development process“ [21, S. 217]. Dadurch wird eine hervorragende Trennung der Belange geschaffen und der Ressourcenbedarf kann minimal gehalten werden. Bei dem Entwurf von CiAO wurden drei grundlegende *aspektorientierte Entwurfsprinzipien* genutzt, die auch in dieser Diplomarbeit angewendet werden und deshalb im Folgenden erläutert werden. Abschließend wird auf die Rollen der Aspekte beim Entwurf eingegangen und eine Klassifizierung nach Aspekttypen vorgenommen.

[4] CiAO is Aspect-Oriented
[5] Hervorhebungen im Original

2.5.1 Aspektorientierte Entwurfsprinzipien

Der Entwurf und die Implementierung von hochgradig konfigurierbarer Software mit AOP erfordern eigene Methoden und Prinzipien. Die nachfolgenden Entwurfsprinzipien können dabei helfen, den größtmöglichen Nutzen der Aspektorientierung zu erreichen, indem AOP bereits beim Systementwurf berücksichtigt wird.

Lose Kopplung

Durch eine *lose Kopplung* von Komponenten lassen sich Abhängigkeiten zwischen diesen auflösen. Ein anschauliches Beispiel dafür ist die Initialisierung, die in CiAO dezentral realisiert wird (siehe Abbildung 2.2). Jede Komponente enthält einen Aspekt, der sich an eine Initialisierungsfunktion (hier: `hw::hal::init()`) bindet, die vom System bereitgestellt wird. Zu einem geeigneten Zeitpunkt ruft das System diese Initialisierungsfunktion auf und die o. g. Aspekte stellen sicher, dass jede Komponente richtig initialisiert wird. Das System benötigt keine Kenntnis über die tatsächlich integrierten Komponenten. Somit können diese unabhängig verändert werden, ohne dass eine Modifikation des Systems erforderlich ist.

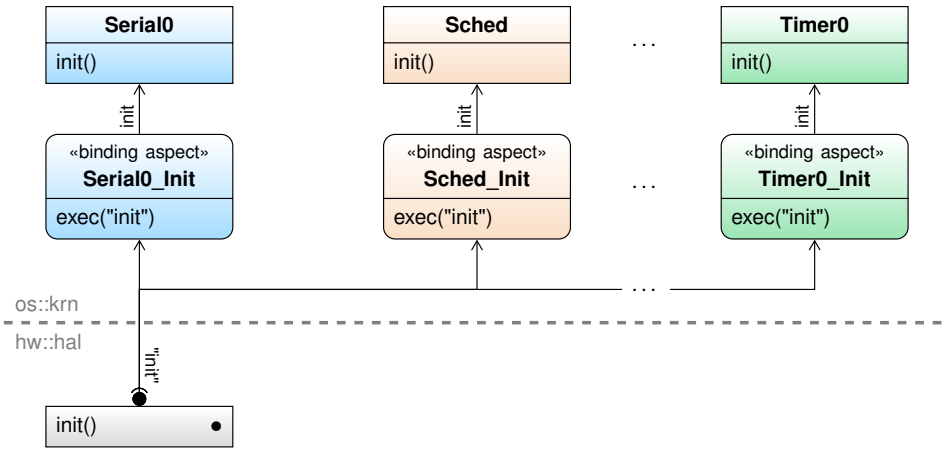

Abbildung 2.2: Initialisierung durch lose Kopplung [22, S. 130]

Sichtbare Übergänge

Damit ein System optimal durch Aspekte konfiguriert werden kann, muss der Kontrollfluss im System zu jedem Zeitpunkt statisch durch Aspekte beeinflussbar sein. In CiAO sind beispielsweise die Betriebssystemkomponenten in eigenen C++-Namensräumen gekapselt, z. B. `os::krn`. Alle Kontrollflüsse hinein und aus diesem Namensraum heraus lassen sich durch Aspekte beeinflussen, wodurch z. B. eine konfigurierbare *Unterbrechungssynchronisation* [23] ermöglicht wird. Weiterhin muss ein Funktionsaufruf genau eine Funktionalität realisieren, damit statisch erkannt werden kann, zu welchem Zweck diese

Funktion aufgerufen wird. Dieses Entwurfsprinzip wird deswegen *sichtbare Übergänge* genannt.

Minimale Erweiterungen

Das Prinzip der *minimalen Erweiterungen* wird zuerst von Parnas [24] erwähnt. Dabei handelt es sich um ein allgemeines Entwurfsprinzip, nach dem konfigurierbare Software zunächst nur die absolut notwendige Funktionalität enthält. Durch minimale Erweiterungen werden anschließend neue Merkmale hinzugefügt, so dass eine hochgradig konfigurierbare Produktlinie entsteht. Mit AOP lässt sich dieses Entwurfsprinzip sehr gut realisieren, indem vorhandene Klassen durch Klassenfragmente ergänzt werden, die neue Attribute und Methoden einfügen. Abbildung 2.3 illustriert dieses Entwurfsprinzip anhand von CiAO, dessen Komponenten `AS`, `Task` und `Sched` nur absolut notwendige Funktionalität enthalten. Optionale Konzepte, wie der querschneidende `ResourceSupport`, der diese drei Komponenten beeinflusst, werden durch minimale Erweiterungen in das bestehende System integriert.

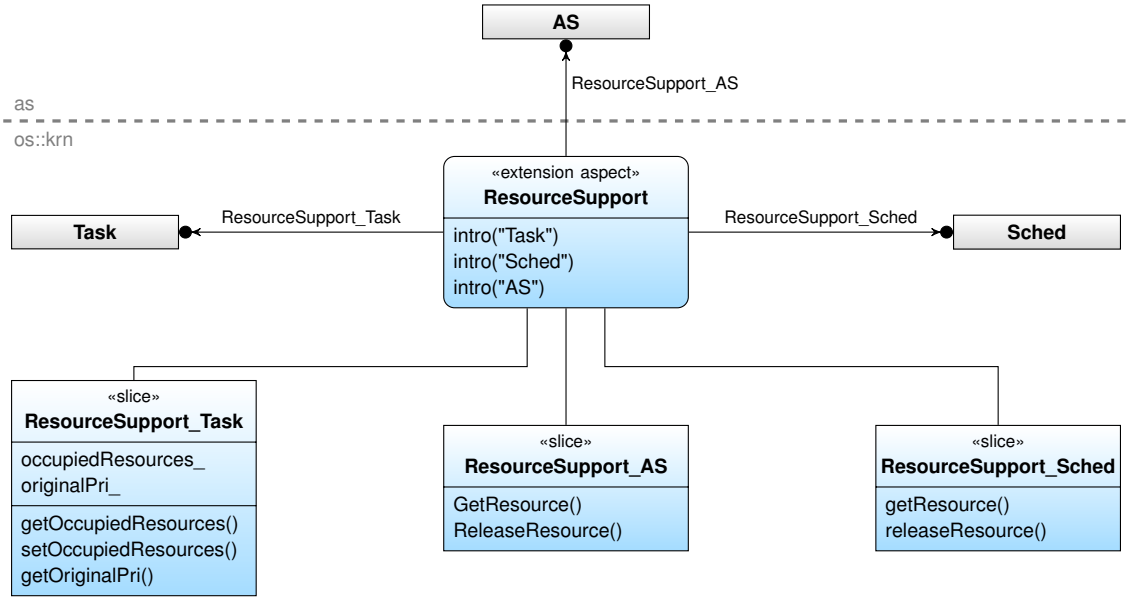

Abbildung 2.3: Minimale Erweiterungen durch Klassenfragmente [22, S. 136]

2.5.2 Aspekttypen

Neben aspektorientierten Entwurfsprinzipien (siehe 2.5.1) gibt es eine Kategorisierung von Aspekten gemäß ihrem Verwendungszweck. Die folgenden drei Aspekttypen lassen sich grob den o. g. Entwurfsprinzipien der Reihe nach zuordnen.

Binding-Aspekte

Aspekte, welche die Integration von Komponenten in ein System übernehmen, zählen allgemein zu der Klasse der *Binding-Aspekte*. Charakteristisch für diesen Aspekttyp ist die obligatorische Zugehörigkeit zu einer Komponente. Effektiv vertauscht ein Binding-Aspekt die *„kennt"-Relation* zwischen dem System und der zu integrierenden Komponente: Nur der Binding-Aspekt der Komponente „kennt" das System – das System hingegen hat keine Informationen über die Komponente, sondern stellt lediglich einen Join-Point für die Integration bereit. Demgegenüber steht die klassische Komponentenintegration ohne AOP, in der das System explizit die vorhandenen Komponenten „kennt" und somit kaum konfigurierbar ist. Im Folgenden wird auf einen Spezialfall der Binding-Aspekte eingegangen, welche die o. g. Vertauschung der „kennt"-Relation ausnutzen, um Ereignisse entlang dieser Relation zu propagieren.

Wird ein System gemäß einer *funktionalen Hierarchie* [25, S. 267] entworfen, so wird es in Schichten $S_0, S_1, \ldots, S_n$ eingeteilt. Eine Schicht S_i „kennt" die Schichten $S_{i-1}, S_{i-2}, \ldots, S_0$ und kann deren Funktionalität benutzen, darf jedoch keine Schicht S_j mit $j > i$ direkt verwenden. Eine solche *Schichtenarchitektur* ermöglicht Konfigurierbarkeit durch Weglassen von „oben liegenden" Schichten. Werden z. B. die Schichten $S_i, S_{i+1}, \ldots, S_n$ weggelassen, so hat das keine Auswirkung auf die Arbeitsweise der „unten liegenden" Schichten $S_{i-1}, S_{i-2}, \ldots, S_0$, die unabhängig davon weiter funktionieren. Ein *Upcall* entspricht nun einem Funktionsaufruf entgegen dieser Hierarchie, z. B. von Schicht S_1 nach S_2, den es laut dieser Definition nicht geben darf. Durch Binding-Aspekte lässt sich so etwas dennoch realisieren, ohne dabei die o. g. Konfigurierbarkeit zu verletzen. Solche Aspekte werden als *Upcall-Aspekte* bezeichnet und stellen eine Unterklasse der Binding-Aspekte dar.

Policy-Aspekte

Aspekte, die Beziehungen zwischen unabhängigen Komponenten herstellen, werden *Policy-Aspekte* genannt. Voraussetzung für den Einsatz dieses Aspekttyps ist ein System, das mit dem Entwurfsprinzip der sichtbaren Übergängen entworfen ist. Durch Policy-Aspekte lassen sich in einem solchen System verschiedene Strategien realisieren, indem der Kontrollfluss des Systems beeinflusst wird, so dass z. B. eine neue Komponente für die gewünschte Strategie benutzt wird. Prinzipiell ist die Strategie, die ein Policy-Aspekt umsetzt, unabhängig von den Komponenten des Systems.

Erweiterungsaspekte

Das Werkzeug zur Implementierung des Entwurfsprinzips der minimalen Erweiterungen mit AOP ist ein *Erweiterungsaspekt*. Dieser fügt neue Funktionalität in bestehende Komponenten ein, indem der Kontrollfluss durch zusätzliche Instruktionen ergänzt wird und neue Attribute, Methoden und Vererbungsbeziehungen angelegt werden.

3 Domänenanalyse

Nachdem in Kapitel 2 die theoretischen Grundlagen für eine Domänenanalyse erklärt wurden, wird in diesem Kapitel deren praktische Durchführung erläutert. Dabei wird die Domäne der *Internetprotokollfamilie in eingebetteten Systemen* analysiert. In einem ersten Schritt (siehe 3.1) wird die Architektur der im Internet verwendeten Protokolle anhand eines Referenzmodells beschrieben. Anschließend erfolgt eine Vorstellung und Bewertung von bereits existierender Software für die o. g. Domäne. In Abschnitt 3.3 wird abschließend ein Domänenmodell erstellt, welches die Anforderungen an eine Software-Produktlinie für diese Domäne definiert.

3.1 Referenzmodell

Netzwerkprotokolle werden in der Regel als Schichtenarchitektur entworfen, um deren Komplexität möglichst gering zu halten und um Konfigurierbarkeit zu gewährleisten. Jede Ebene einer Schichtenarchitektur realisiert eine in sich abgeschlossene Funktionalität, wodurch eine Aufteilung eines vollständigen Systems in kleinere Komponenten mit überschaubaren Eigenschaften erreicht wird. Das *OSI*[1]*-Modell* [26] ist ein standardisiertes *Referenzmodell* einer Schichtenarchitektur, die den Ablauf einer Kommunikation konzeptionell beschreibt. Der Kommunikationsvorgang wird dabei in sieben Schichten eingeteilt, die in Abbildung 3.1 (linke Seite) dargestellt sind. In der Praxis hat sich jedoch ein vereinfachtes Modell, das *TCP/IP-Referenzmodell* [27], etabliert, welches in Abbildung 3.1 (rechte Seite) gezeigt wird. Im Gegensatz zu dem OSI-Modell sind einige Schichten zusammengefasst, so dass eine Einteilung in vier Schichten erfolgt. Das TCP/IP-Referenzmodell verwenden alle Computersysteme, die mit dem Internet verbunden sind. Aus diesem Grund werden nachfolgend die vier Schichten dieses Referenzmodells kurz erläutert.

Anwendungsschicht

Die Anwendungsschicht ist die oberste der vier Schichten. Auf ihr findet die Interaktion mit dem Benutzer eines Computersystems statt. Die Daten, die während der Kommunikation ausgetauscht werden, entstammen dieser Schicht. Es gibt unzählige Protokolle, die dort angesiedelt sind, wie z. B. *HTTP*[2] (World Wide Web), *SMTP*[3] (E-Mail) und *FTP*[4]

[1] Open Systems Interconnection
[2] Hypertext Transfer Protocol
[3] Simple Mail Transfer Protocol
[4] File Transfer Protocol

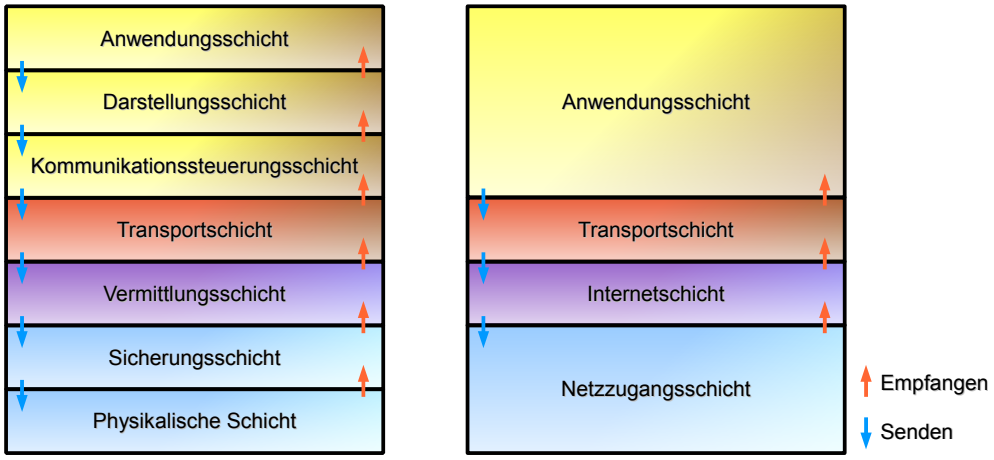

Abbildung 3.1: OSI- und TCP/IP-Referenzmodell

(Datentransfer). Damit wird für jeden Anwendungszweck festgelegt, welchem Format die übertragenen Daten entsprechen müssen.

Transportschicht

Die *Transportschicht*, die unter der Anwendungsschicht liegt, stellt eine Beziehung zwischen den beteiligten Kommunikationspartnern her. Dabei wird in der Regel eines der Protokolle *TCP*[5] [28] oder *UDP*[6] [29] verwendet. TCP ist ein verbindungsorientiertes Protokoll, das für eine zuverlässige Kommunikation sorgt, so dass alle Daten vollständig und in der richtigen Reihenfolge übertragen werden. UDP hingegen ist wesentlich einfacher aufgebaut. Es ist ein verbindungsloses Protokoll und liefert keine Garantie, ob die übertragenen Daten ankommen und in welcher Reihenfolge. Beide Protokolle haben unterschiedliche Anwendungsfälle. Während TCP für E-Mail und Datentransfer genutzt wird, findet UDP Anwendung im Multimedia-Bereich.

Internetschicht

An die Transportschicht schließt die Internetschicht an, welche die Adressierung der zu übertragenden Daten übernimmt. Dazu werden alle Daten in Form von kleinen *Paketen* verschickt, die mit einer Absender- und Zieladresse versehen sind. Das dabei eingesetzte Protokoll heißt *IP*[7] [30] und ist dafür zuständig, dass jedes Paket an den richtigen Ort geschickt wird. Zurzeit wird fast ausschließlich die Version 4 benutzt (IPv4), mit der etwas über vier Milliarden (2^{32}) verschiedene Teilnehmer adressiert werden können. Mit der Version 6 (IPv6) [31] steht ein Nachfolger für das Internet der heutigen Form bereit, der einen deutlich vergrößerten Adressraum von 2^{128} Teilnehmern umfasst.

[5] Transmission Control Protocol
[6] User Datagram Protocol
[7] Internet Protocol

Netzzugangsschicht

Die Netzwerkhardware, die für das Übertragen der Pakete auf einem physikalischen Medium zuständig ist, sowie die dazu passende Softwareanbindung (*Treiber*) befinden sich auf der *Netzzugangsschicht*. Diese Schicht ist die unterste der vier Schichten und regelt, wann und wie lange die Netzwerkhardware senden darf. Darüber hinaus findet in vielen Fällen eine Fehlererkennung statt, d. h. die empfangenen Pakete werden auf Korrektheit überprüft. Es gibt eine große Menge von verschiedenen Netzwerktechnologien, die auf dieser Schicht eingesetzt werden, doch die meisten Computersysteme verwenden das *Ethernet* [32], für welches die entsprechende Hardware extrem günstig erhältlich ist.

3.2 Implementierungen

Die Software-Implementierung einer der beiden in Abbildung 3.1 gezeigten Referenzmodelle wird aufgrund der Schichtenarchitektur als *Stack* bezeichnet. Ein *TCP/IP-Stack* ist somit eine konkrete Implementierung des TCP/IP-Referenzmodells und realisiert die praktische Anwendung des theoretischen Modells. Im Folgenden werden verschiedene TCP/IP-Stacks vorgestellt, deren Quelltexte frei verfügbar sind und damit Grundlagen für diese Diplomarbeit darstellen können. Die Betrachtung vorhandener TCP/IP-Stacks erfolgt an dieser Stelle, um von bereits existierendem Wissen zu profitieren. Weiterhin wird untersucht, ob sich diese Implementierungen für eingebettete Systeme eignen, d. h. ob sie konfigurierbar und somit effizient bzgl. des Speicherbedarfs sind.

3.2.1 Berkeley Software Distribution (BSD)

„The de facto standard for TCP/IP implementations is the one from the Computer Systems Research Group at the University of California at Berkeley“ [33, S. 16]. Bereits 1982 wurde die erste Version dieser Software veröffentlicht [34, S. 4] und seitdem kontinuierlich verbessert. Dabei handelt es sich primär um ein UNIX-artiges Betriebssystem, das mit einem TCP/IP-Stack ausgestattet ist. Die Funktionalität dieser Software ist ausgesprochen groß, z. B. gibt es mit dem *KAME-Projekt* [35] bereits seit dem Jahr 2000 eine Unterstützung für IPv6. Zielplattform dieser Software ist eine Workstation, so dass es nur geringfügige Konfigurierungsmöglichkeiten gibt. Quelltext 3.1 zeigt einen Auszug aus einer Konfigurationsdatei des *OpenBSD*[8]*-Kernels*. Die Option `INET` kombiniert die Protokolle IP, ICMP[9], TCP und UDP und ermöglicht keine separate Auswahl. Lediglich optionale Merkmale wie IPv6 und einige TCP-Erweiterungen sind konfigurierbar. Neben OpenBSD gibt es noch vergleichbare Weiterentwicklungen. Dabei ist z. B. *FreeBSD* zu nennen, das sich in den Konfigurationsmöglichkeiten jedoch kaum unterscheidet. Laut Dunkels [36, S. 20] beträgt allein die Codegröße der Implementierung von TCP in FreeBSD 4.1 etwa 27 KByte. Dieser TCP/IP-Stack eignet sich somit ausschließlich für eingebettete Systeme mit einem ausreichend großen Speicher.

[8] OpenBSD ist eine Weiterentwicklung der ursprünglichen BSD
[9] Internet Control Message Protocol

```
option     TCP_SACK  # Selective Acknowledgements for TCP
option     TCP_ECN   # Explicit Congestion Notification for TCP
option     TCP_SIGNATURE # TCP MD5 Signatures, for BGP routing sessions
#option    TCP_FACK  # Forward Acknowledgements for TCP

[...]

option     INET      # IP + ICMP + TCP + UDP
option     ALTQ      # ALTQ base
option     INET6     # IPv6 (needs INET)
option     IPSEC     # IPsec
```

Quelltext 3.1: /usr/src/sys/conf/GENERIC aus OpenBSD 4.7

3.2.2 Linux

Das Betriebssystem *Linux* enthält einen TCP/IP-Stack, der die gleiche Programmierschnittstelle (*API*[10]) wie die BSD-Systeme aufweist, jedoch unabhängig davon entwickelt wurde [37, S. 20]. Entwurfsziele dieses Stacks sind hohe Performanz und reichhaltige Funktionalität, die mittlerweile sogar über die der BSD-Systeme hinausgeht. Der dadurch verursachte Speicherbedarf spielt eine untergeordnete Rolle, da Linux ebenfalls für den Einsatz auf einer Workstation oder einem Desktop-Rechner entwickelt wurde. Folglich fallen die Konfigurierungsmöglichkeiten dieses TCP/IP-Stacks gering aus. Es gibt zwar durchaus viele Konfigurationspunkte, jedoch nur für optionale Erweiterungen. Die Grundfunktionen, wie TCP und UDP, lassen sich genau wie bei den BSD-Systemen nicht einstellen. Abbildung 3.2 zeigt die unkomprimierten Codegrößen der wichtigsten Konfigurationspunkte des TCP/IP-Stacks eines Linux-Kernels[11] für die i386-Architektur. Auffällig ist, dass mindestens 240 KByte benötigt werden. Dieser TCP/IP-Stack ist für eingebettete Systeme mit wenig Speicher absolut ungeeignet.

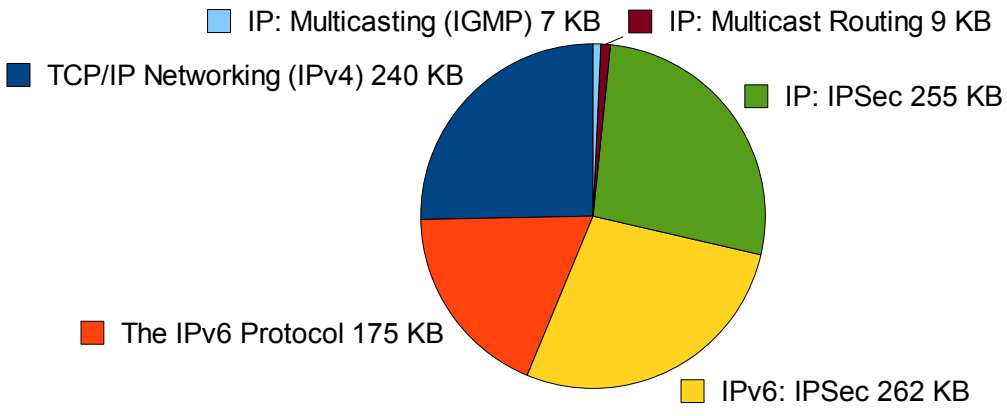

Abbildung 3.2: Codegröße des TCP/IP-Stacks von Linux-2.6.33.2 (i386)

[10] Application Programming Interface
[11] Version 2.6.33.2

3.2.3 PICmicro Stack

PICmicro ist eine Produktlinie von 8, 16 und 32 Bit Mikrocontrollern, für die Bentham [38] einen TCP/IP-Stack entwickelt hat, der speziell auf den Einsatz als *Webserver* (HTTP) zugeschnitten ist. Die gesamte Software passt in einen 8 Bit Mikrocontroller mit etwas weniger als 16 KByte Programmspeicher. Allerdings werden an den implementierten Protokollen, d. h. TCP und IP, deutliche Vereinfachungen vorgenommen. Beispielsweise wird davon ausgegangen, dass die übertragenen Daten pro Verbindung niemals die Größe eines einzelnen *TCP-Segmentes* überschreiten [38, S. 283], welches in der Regel 1460 Byte umfasst. Damit ist der Einsatz dieses TCP/IP-Stacks nur für eine sehr eingeschränkte Gruppe von Anwendungsfällen geeignet, wie z. B. als Webserver für kleine Webseiten.

3.2.4 Lightweight IP

Speziell an die Anforderungen von eingebetteten Systemen ist *Lightweight IP* (lwIP) [36] angepasst. Dabei handelt es sich um einen hochgradig konfigurierbaren TCP/IP-Stack, der z. B. in dem Betriebssystem *eCos*[12] verwendet wird. Die Konfigurierungsmöglichkeiten umfassen u. a. die Speicherverwaltung, IP, ICMP, UDP, TCP und die Programmierschnittstelle. Letztere kann entweder durch *ereignisgesteuerte Callback-Funktionen* realisiert werden oder eine BSD-ähnliche API nachbilden. Damit kann lwIP durchaus als Software-

```
#if !MEM_USE_POOLS && !MEMP_MEM_MALLOC
static
#endif
const u16_t memp_sizes[MEMP_MAX] = {
#define LWIP_MEMPOOL(name,num,size,desc)  LWIP_MEM_ALIGN_SIZE(size),
#include "memp_std.h"
};

#if !MEMP_MEM_MALLOC /* don't build if not configured for use */

/** This array holds the number of elements in each pool. */
static const u16_t memp_num[MEMP_MAX] = {
#define LWIP_MEMPOOL(name,num,size,desc)  (num),
#include "memp_std.h"
};

/** This array holds a textual description of each pool. */
#ifdef LWIP_DEBUG
static const char *memp_desc[MEMP_MAX] = {
#define LWIP_MEMPOOL(name,num,size,desc)  (desc),
#include "memp_std.h"
};
#endif /* LWIP_DEBUG */
```

Quelltext 3.2: memp.c aus lwIP 1.32

[12] embedded Configurable operating system

Produktlinie angesehen werden, da hinreichende Variabilität vorhanden ist. Aufgrund der hohen Variabilität fällt ebenfalls die Codegröße variabel aus, die in der Version 1.32 von lwIP, je nach Konfiguration, etwa zwischen 5 und 40 KByte beträgt. Eine detailliertere Auflistung der Codegröße erfolgt in Abschnitt 6.3. In Quelltext 3.2, der einen typischen Ausschnitt aus einer Implementierungsdatei von lwIP zeigt, ist zu erkennen, dass sämtliche Variabilität der Software durch den Einsatz des C-Präprozessors erreicht wird. Dadurch entstehen die in Abschnitt 2.4 angesprochenen Probleme bzgl. der Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit des Quelltextes. Abgesehen von diesen Nachteilen bietet lwIP einen flexibel konfigurierbaren TCP/IP-Stack, der sich optimal für eingebettete Systeme eignet.

3.2.5 Micro IP

Mit etwa 7 KByte Codegröße realisiert *Micro IP* (uIP) [39] einen minimalen TCP/IP-Stack, der die Protokolle TCP, IP und Ethernet unterstützt sowie eine eingeschränkte Nutzung von UDP erlaubt. Abgesehen von der optionalen Unterstützung für UDP gibt es kaum Konfigurierungsmöglichkeiten, die eine Anpassung dieser Software erlauben. Beispielsweise ist TCP nicht deaktivierbar und es wird nur genau ein Netzwerkgerät unterstützt. Wie der PICmicro Stack führt uIP ebenfalls diverse Vereinfachungen an den Protokollen durch. So wird z. B. TCP nur als *Stop-and-Wait*-Verfahren [40, S. 206ff] implementiert, d. h. es kann immer nur ein TCP-Segment pro Verbindung gleichzeitig versendet bzw. empfangen werden. Falls ein TCP-Segment verloren geht, ist es nicht wie sonst üblich der TCP/IP-Stack, der sich um das erneute Versenden kümmert, sondern die Anwendung selbst muss dafür Sorge tragen. Dadurch gestaltet sich die Anwendungsentwicklung deutlich komplizierter als bei den bisher vorgestellten TCP/IP-Stacks. Weiterhin wird die Programmierschnittstelle nur durch ereignisgesteuerte Callback-Funktionen repräsentiert. Eine gleichzeitige Verwendung von uIP durch mehrere Tasks ist also problematisch. Zielplattform dieses TCP/IP-Stacks sind eingebettete Systeme mit extrem wenig Speicher und in der Regel keinem Betriebssystem, so dass uIP dort wie eine Programmbibliothek benutzt wird.

3.3 Domänenmodell

Ein Domänenmodell, dessen Eigenschaften bereits in Abschnitt 2.3 beschrieben wurden, ist der wichtigste Bestandteil einer Domänenanalyse. In diesem Kapitel erfolgt die konkrete Erstellung eines solchen Domänenmodells. Zuerst wird mit der Domänendefinition in Abschnitt 3.3.1 das Scoping durchgeführt, um den Abdeckungsbereich der Software-Produktlinie zu konkretisieren. Anschließend folgen essentielle Konzeptmodelle, die bei einem späteren Domänenentwurf zwingend berücksichtigt werden müssen. In Abschnitt 3.3.3 werden daraufhin die für eine Software-Produktlinie obligatorischen Merkmalmodelle erläutert. Auf ein explizites Domänenlexikon wird an dieser Stelle aufgrund des begrenzten Umfangs dieser Diplomarbeit verzichtet.

3.3.1 Domänendefinition

Die Domäne dieser Diplomarbeit ist ein TCP/IP-Stack für eingebettete Systeme, der mindestens die Protokolle IPv4, UDP und TCP umfasst. Aufgrund der enormen Verbreitung des Ethernets muss dieses ebenfalls unterstützt werden, wodurch zusätzlich das Protokoll *ARP*[13] [41] erforderlich ist. Eine zukünftige Erweiterung um neue Protokolle, wie z. B. IPv6, sollte problemlos möglich sein. Da die Zielplattform eingebettete Systeme sind, muss dieser TCP/IP-Stack effizient bzgl. des Speicherbedarfs und der Laufzeit sein. Sicherheit durch kryptographische Verfahren wie *IPsec*[14] und *TLS*[15] gehören hingegen nicht zum Funktionsumfang. Die Domänendefinition wird nachfolgend durch Beispiele und Gegenbeispiele vervollständigt, die veranschaulichen, welche Systeme zu dieser Domäne gehören und welche Systeme ausgeschlossen werden.

Beispielsysteme dieser Domäne

1. **Sensornetzwerk:** Geographisch verteilte Mikrocontroller messen Wetterdaten (Temperatur, Luftdruck, usw.) und senden diese periodisch über UDP in Verbindung mit IPv4. Ein herkömmlicher Rechner protokolliert die empfangenen Daten.
2. **Smartphone:** Ein modernes Mobiltelefon verbindet sich per *UMTS*[16] mit dem Internet, um mittels *VoIP*[17] preiswerte Telefongespräche zu ermöglichen. Dabei werden die Protokolle IPv4 und UDP eingesetzt.
3. **Navigationsgerät:** Durch eine UMTS-Verbindung werden aktuelle Benzinpreise von nahe gelegenen Tankstellen abgerufen und es wird von einem Navigationsgerät für Automobile eine Route zu der günstigen Tankstelle berechnet. Die Daten werden über TCP in Verbindung mit IPv4 übertragen.

Gegenbeispielsysteme

1. **Rechenzentrum:** Ein Server in einem Rechenzentrum, der tausende von Anfragen pro Sekunden bearbeitet.
2. **Desktop-Rechner:** Ein gewöhnlicher Computer, mit dem E-Mails durch TLS verschlüsselt übertragen werden.

3.3.2 Konzeptmodelle

In diesem Abschnitt werden drei wichtige Konzepte dieser Domäne vorgestellt, die für einen Domänenentwurf essentiell sind. Diese Konzepte sind nicht spezifisch für eingebettete Systeme, sondern betreffen jeden TCP/IP-Stack. Zunächst wird dabei auf die Konzepte

[13] Address Resolution Protocol
[14] Internet Protocol Security
[15] Transport Layer Security
[16] Universal Mobile Telecommunications System
[17] Voice over IP

Verkapselung und Demultiplexing eingegangen, die den Sende- und Empfangsvorgang von Datenpaketen allgemein festlegen. Anschließend wird das zustandsbasierte Konzept des Protokolls TCP erläutert.

Verkapselung

Das Versenden von Daten nach dem TCP/IP-Referenzmodell (siehe 3.1) sieht das Konzept der *Verkapselung* (*„Encapsulation“*) vor. Die zu versendenden Nutzdaten werden von der Anwendungsschicht bis zur Netzzugangsschicht weitergereicht und dabei in jeder Schicht um zusätzliche Header-Informationen erweitert. Dieser Vorgang wird in Abbildung 3.3 illustriert. Die Länge der Daten nimmt bei jedem Übergang zwischen den gezeigten Schichten kontinuierlich zu, bis schließlich die Netzzugangsschicht erreicht ist. Der dort aufgebaute *Ethernet frame* umfasst einen Ethernet-Header, IP-Header, TCP-Header und die eigentlichen Nutzdaten. Außerdem ist ein *Ethernet trailer* enthalten, der eine 32 Bit *CRC*[18]-Prüfsumme darstellt. Die Nutzdaten sind also durch die zusätzlichen Header und den Trailer wie in einer *„Kapsel“* eingeschlossen. So lassen sich die vier Schichten des TCP/IP-Referenzmodells direkt an einem versendeten Datenpaket ablesen. Ein TCP/IP-Stack muss das Konzept der Verkapselung realisieren, damit eine standardkonforme Kommunikation mit anderen Systemen erfolgen kann.

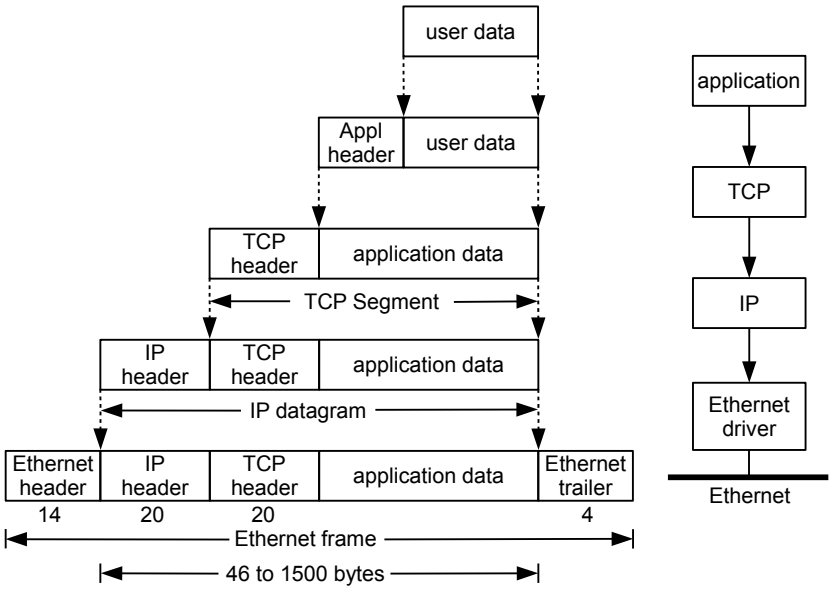

Abbildung 3.3: Verkapselung beim Sendevorgang [33, S. 10]

Demultiplexing

Das *Demultiplexing* ist das inverse Konzept zur Verkapselung. Der Empfang eines Datenpaketes beginnt nach dem TCP/IP-Referenzmodell (siehe 3.1) auf der Netzzugangsschicht.

[18]Cyclic Redundancy Check

Von dort aus wird das Paket Schicht für Schicht nach oben geleitet, bis es schließlich in der Anwendungsschicht sein Ziel gefunden hat. Bei jedem Übergang zwischen diesen Schichten werden die Protokoll-Header analysiert und aufgrund der darin enthaltenen Informationen wird das Demultiplexing durchgeführt. Dieses Konzept ist in Abbildung 3.4 dargestellt. Jeder Protokoll-Header enthält einen „Hinweis“ darauf, wie die Verkapselung aufgebaut ist, d. h. welches Protokoll als nächstes analysiert werden muss. Damit kann auf jeder Schicht entschieden werden, zu welchem Protokoll der nächst höheren Schicht das empfangene Paket weitergeleitet werden soll. Bei dieser Weiterleitung wird der analysierte Protokoll-Header entfernt und der nächsten Schicht lediglich ein kleineres Datenpaket übergeben. Dieser Vorgang geschieht für die Anwendungsschicht völlig transparent, so dass dort nur die Nutzdaten ohne die verwendeten Header-Informationen ankommen.

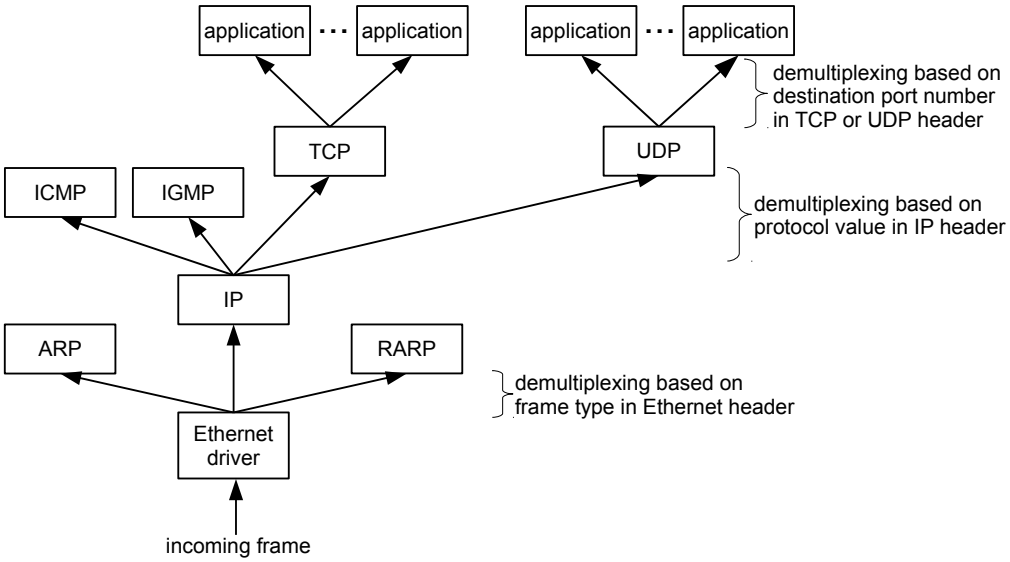

Abbildung 3.4: Demultiplexing beim Empfangsvorgang [33, S. 11]

TCP-Zustandsmaschine

TCP ist das einzige in dieser Diplomarbeit behandelte Protokoll, das zustandsbasiert funktioniert. Da TCP ein verbindungsorientiertes Protokoll ist, muss der aktuelle Zustand einer Verbindung abgespeichert werden, d. h. ob eine Verbindung gerade aufgebaut wird, bereits besteht oder beendet ist. Für die Beschreibung von zustandsbasierten Systemen gibt es mit *StateCharts* [42] einen geeigneten Formalismus. In Abbildung 3.5 ist ein StateChart der *TCP-Zustandsmaschine* gezeigt, die das Verhalten des Protokolls konzeptionell spezifiziert. Normalerweise wird nur ein Teil der in diesem Diagramm abgebildeten Zustände und Zustandsübergänge verwendet. Ein TCP-Server beispielsweise führt in der Regel nur genau die Zustandsübergänge durch, die in diesem Diagramm durch gestrichelte Pfeile gekennzeichnet sind. Die breiteren Pfeile hingegen heben die üblichen Zustandsübergänge eines TCP-Clients hervor. Ausgangspunkt der TCP-Zustandsmaschine ist der

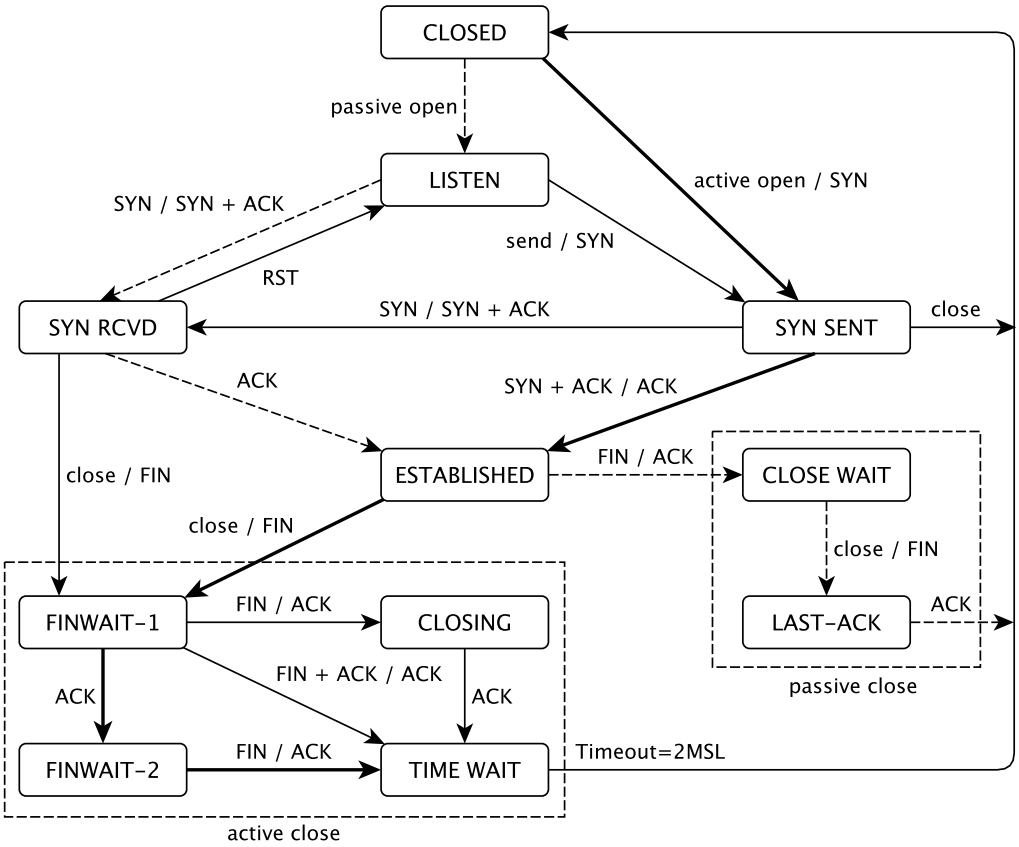

Abbildung 3.5: StateChart der TCP-Zustandsmaschine [33, S. 241]

Zustand *CLOSED*. Wenn ein TCP-Client benutzt werden soll, wird von der Anwendungsschicht das Ereignis *active open* ausgelöst und in den Zustand *SYN SENT* gewechselt. Falls hingegen ein TCP-Server verwendet werden soll, wird durch *passive open* in den Zustand *LISTEN* übergegangen und auf einen externen Verbindungsaufbau gewartet. Der darauf folgende Verbindungsaufbau wird als *„three-way handshake“* [28, S. 30] bezeichnet und durch den Austausch von Paketen mit entsprechenden *TCP-Flags* realisiert. Diese Flags sind im dem StateChart mit Abkürzungen aus drei Großbuchstaben ersichtlich. Der Nutzdatentransfer findet sowohl bei einem Server als auch bei einem Client in dem Zustand *ESTABLISHED* statt, der eine korrekt aufgebaute Verbindung symbolisiert. Die Terminierung einer Verbindung erfolgt durch die Anwendungsschicht mit dem Ereignis *close* und damit verbundenen Zustandsübergang in den Zustand *FINWAIT-1*. Dieser Vorgang wird als *active close* bezeichnet. Falls jedoch der Kommunikationspartner die Verbindung vorzeitig beendet, findet ein *passive close* statt und es wird in den Zustand *CLOSE WAIT* gewechselt. Im Anschluss erfolgt wiederum ein Austausch von Paketen mit entsprechenden TCP-Flags. Wenn dieser Vorgang abgeschlossen ist, befindet sich die TCP-Zustandsmaschine wieder im Anfangszustand *CLOSED*. Auf eine detailliertere Beschreibung dieses Diagramms sei an dieser Stelle lediglich auf Stevens [33, S. 240ff] verwiesen. Es bleibt festzuhalten, dass das Konzept der TCP-Zustandsmaschine für

jedes System, welches TCP verwendet, enorm wichtig ist. Nur wenn die Semantik dieses Konzeptes von jedem Kommunikationsteilnehmer exakt eingehalten wird, kann eine Interoperabilität von verschiedenen Systemen gewährleistet werden.

3.3.3 Merkmalmodelle

Merkmalmodelle definieren den Funktionalitätsumfang einer Software-Produktlinie. Zur grafischen Erfassung von gemeinsamen und variablen Anforderungen an die Produktlinie können Merkmaldiagramme eingesetzt werden. Im Nachfolgenden wird die Notation von Czarnecki und Eisenecker [10, S. 87ff] zur Darstellung von Merkmaldiagrammen benutzt. Abbildung 3.6 zeigt ein solches Merkmaldiagramm, das die generellen Anforderungen an einen TCP/IP-Stack in dieser Domäne definiert. Die ausgefüllten Kreise beziehen sich auf Merkmale, die immer vorhanden sein müssen. Die Speicherverwaltung, Protokoll- und Netzwerkgeräteunterstützung sowie die API gehören somit zum minimalen Funktionsumfang und können durch weitere Merkmalmodelle spezifiziert werden. Die Betriebssystemunterstützung, ein Paketfilter und das Routing sind hingegen optionale Komponenten, die lediglich in einigen Varianten des TCP/IP-Stacks vorhanden sein können. Im Folgenden werden zuerst die Merkmale beschrieben, die nicht durch ein weiteres Merkmalmodell verfeinert werden. Im Anschluss daran werden detaillierte Merkmalmodelle der verbleibenden Komponenten vorgestellt.

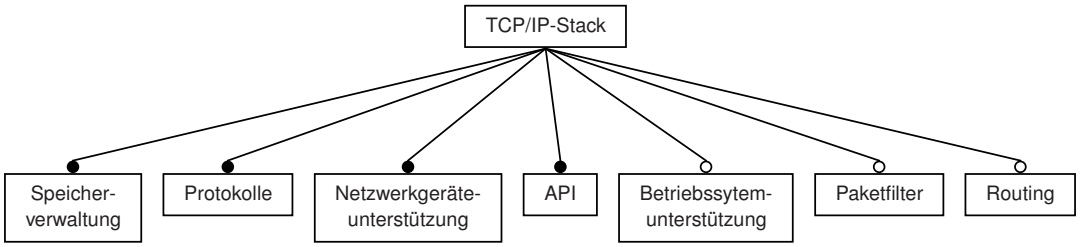

Abbildung 3.6: Merkmaldiagramm des TCP/IP-Stacks

Routing

Das Empfangen und Weiterleiten von Daten, die für ein anderes Computersystem bestimmt sind, wird als Routing bezeichnet. Im Falle von IP werden also IP-Pakete empfangen, gepuffert und weiter versendet, was dem Prinzip *Store-and-Forward* [40, S. 344] entspricht. Nur Geräte, die als *Router* eingesetzt werden, benötigen diese Funktionalität. Alle anderen, die sogenannten *Endgeräte*, machen von dieser Funktion keinen Gebrauch.

Paketfilter

Unerwünschte Pakete lassen sich mit einem Paketfilter ignorieren. So können z. B. gezielt Verbindungen mit bestimmten IP-Adressen ausgeschlossen werden. Außerdem können Pakete, durch die der Betrieb des eingesetzten Gerätes gestört werden könnte, ignoriert

werden. Ein solcher Paketfilter wird häufig auf Routern und Servern benutzt, um deren Sicherheit zu erhöhen. Viele Endgeräte benötigen hingegen keinen Paketfilter.

Betriebssystemunterstützung

Falls dieser TCP/IP-Stack auf einem System eingesetzt wird, auf dem ein Betriebssystem läuft, sollte eine Integration in das Betriebssystem erfolgen. Damit kann für optimale Funktionalität gesorgt werden, indem Betriebssystemmittel wie *Semaphore* und *Timer* genutzt werden. Es gibt jedoch auch Anwendungsfälle, die kein Betriebssystem erfordern, aber dennoch einen TCP/IP-Stack benötigen.

API

Zu den notwendigen Komponenten eines TCP/IP-Stacks gehört die Programmierschnittstelle, die den Informationsfluss zwischen der Anwendungsschicht und der Transportschicht herstellt. In dieser Domäne ist die *Sockets Networking API* [37] am weitesten verbreitet. Alternativ gibt es die Möglichkeit, mit ereignisgesteuerten Callback-Funktionen zu arbeiten (siehe 3.2).

Netzwerkgeräteunterstützung

Die Unterstützung verschiedener Netzwerktechnologien auf der Netzzugangsschicht ist enorm wichtig, um eine ausreichende Flexibilität der Software zu gewährleisten. Ein TCP/IP-Stack, der nur in einer Ethernet-Umgebung eingesetzt werden kann, ist möglicherweise in vielen eingebetteten Systemen nutzlos. In Abbildung 3.7 werden einige Netzwerktechnologien als optionale Merkmale dargestellt, die je nach Anwendungsfall verwendet werden können. Das Ethernet wird exemplarisch um zusätzliche Merkmale erweitert. Manche Ethernet-Karten bieten z. B. eine hardwarebeschleunigte Prüfsummenberechnung von IP-, TCP- und UDP-Prüfsummen an, die zur Steigerung der Effizienz benutzt werden sollte. Das *TCP Offloading* geht noch einen Schritt weiter und lagert Teile des TCP/IP-Stacks auf der Netzwerkkarte aus. Dieses Merkmal ist jedoch stark abhängig

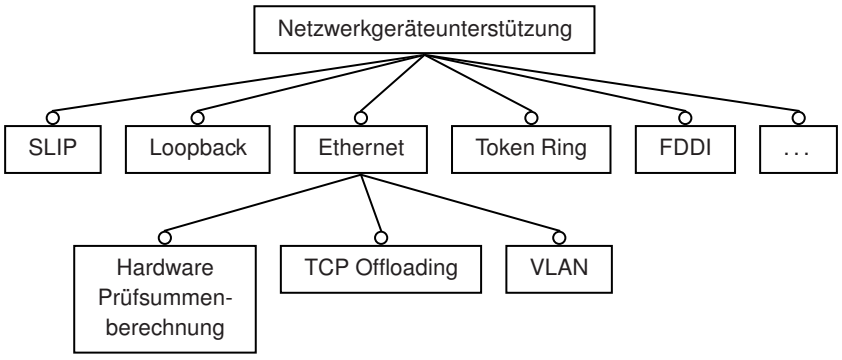

Abbildung 3.7: Optionale Merkmale der Netzwerkgeräteunterstützung

von der verwendeten Hardware, so dass es dafür keine einheitliche Schnittstelle gibt. Mittels *VLAN*[19] lässt sich außerdem das Ethernet virtualisieren, d. h. ein physikalisches Netz wird in mehrere logische Netze aufgeteilt.

Protokolle

Das Kernstück eines TCP/IP-Stacks ist die Unterstützung von Protokollen. Da je nach Anwendungsfall unterschiedliche Protokolle eingesetzt werden, ist jedes Protokoll ein optionales Merkmal. Wichtige Protokolle sind IP, TCP, UDP und ARP, die im Folgenden vorgestellt werden. Neben den benannten gibt es noch viele weitere Protokolle, die sich nach Möglichkeit einfach in den TCP/IP-Stack integrieren lassen sollten. Weist ein Protokoll Variabilität auf, kann es durch ein weiteres Merkmalmodell beschrieben werden.

IP Das Internetprotokoll gibt es in zwei verschiedenen Versionen: IPv4 und IPv6 (siehe 3.1). Das Merkmal IP muss, wie in Abbildung 3.8 illustriert, durch IPv4, IPv6 oder durch beide realisiert werden. Die Unterstützung für IP enthält also mindestens eine der beiden Versionen, was durch den ausgefüllten Kegel unterhalb von IP in dem Merkmaldiagramm veranschaulicht wird. IPv4 und IPv6 sind wiederum in Senden und Empfangen aufgeteilt. Mindestens eines dieser beiden Merkmale muss vorhanden sein, damit eine Unterstützung für die entsprechende Version von IP sinnvoll ist. Jede Version lässt sich durch ein optionales Merkmal, die *Optionen* erweitern. Die *Fragmentierung* ist bei IPv4 direkt in das Protokoll integriert und lässt sich weiterhin in das Senden und Empfangen von Fragmenten aufteilen. Diese Aufteilung hängt davon ab, ob die entsprechende Sende- bzw. Empfangsfunktionalität von IPv4 auch vorhanden ist. Bei IPv6 wird die Fragmentierung durch die *IPv6 Optionen* realisiert.

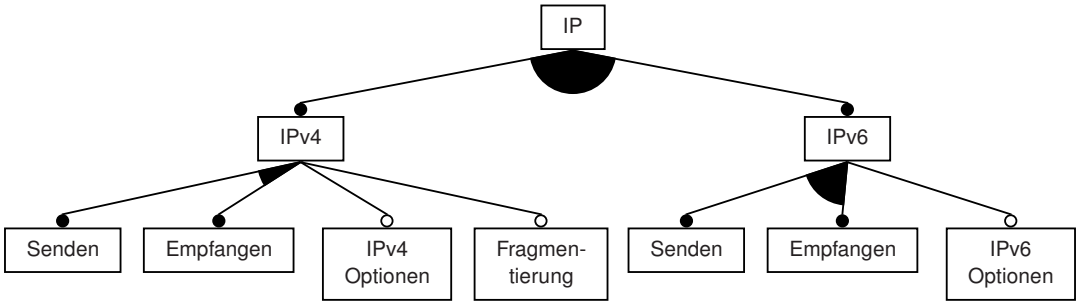

Abbildung 3.8: Variabilität des Internetprotokolls

ARP Das *Address Resolution Protocol* (ARP) ist abhängig von den verwendeten Netzwerkgeräten und von der IP-Version. In der Regel wird ARP für Ethernet und IPv4 verwendet, indem eine Abbildung von IPv4-Adressen auf Ethernet-Adressen stattfindet. Diese Abbildung kann dynamisch oder statisch implementiert werden. Im ersten Fall

[19] Virtual Local Area Network

werden Anfragen versendet, welche die gesuchte IPv4-Adresse enthalten (*ARP-Request*). Das Computersystem, das mit dieser IPv4-Adresse identifiziert wird, antwortet mit der zugehörigen Ethernet-Adresse (*ARP-Reply*). Nach einer gewissen Zeit, dem *Cache Timeout*, kann dieser Vorgang wiederholt werden, um zu prüfen, ob sich der Aufbau des Netzes verändert hat. Ist die Zuordnung von IPv4- zu Ethernet-Adressen bereits bekannt, so kann ein Benutzer diese statisch in das System eintragen, so dass keine Anfragen und Antworten dafür generiert werden müssen. Das Merkmaldiagramm in Abbildung 3.9 veranschaulicht diesen Zusammenhang.

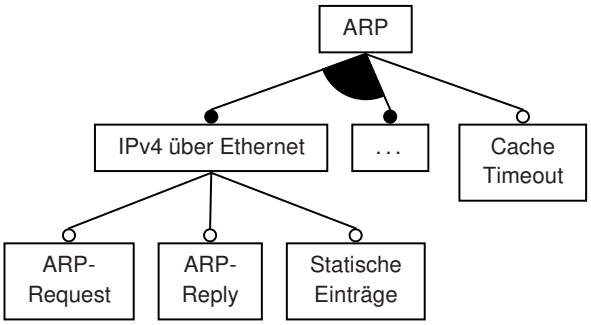

Abbildung 3.9: ARP Merkmaldiagramm

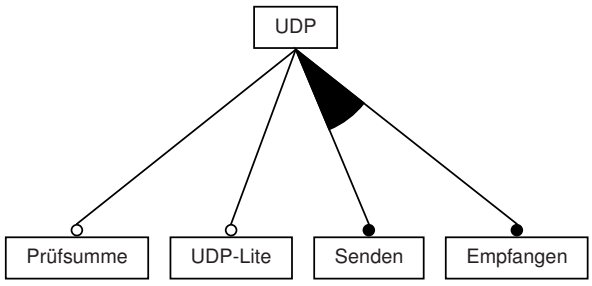

Abbildung 3.10: UDP Merkmaldiagramm

UDP Da das User Datagram Protocol (UDP) selbst nur minimale Funktionalität bietet, gibt es wenige Variationsmöglichkeiten. Abbildung 3.10 stellt alle Merkmale dieses Protokolls dar. UDP lässt sich zunächst nach Senden und Empfangen differenzieren, wobei mindestens eines dieser Merkmale vorhanden sein muss. Weiterhin können alle UDP-Pakete mit einer optionalen Prüfsumme beim Senden versehen werden und, ebenfalls optional, beim Empfangen danach überprüft werden. Mit *UDP-Lite* [43] kann darüber hinaus festgelegt werden, welche Teile eines UDP-Paketes mit dieser Prüfsumme geschützt werden sollen.

TCP „TCP is the most complex of all protocols in the suite of Internet protocols“ [44, S. 165]. Aus dieser Komplexität resultiert eine enorme Variabilität innerhalb dieses Protokolls. In Abbildung 3.11 ist ein Merkmaldiagramm gezeigt, das die wichtigsten Merkmale von TCP enthält. In der Mitte dieses Diagramms ist zu erkennen, dass TCP entweder als Client eingesetzt, d. h. eine Verbindung wird aktiv aufgebaut, oder als Server verwendet werden kann und somit passiv auf den Aufbau einer Verbindung gewartet wird. Die Kombination dieser beiden Merkmale in einem System ist ebenfalls möglich.
TCP ist für die Übertragung von großen Datenmengen ausgelegt und unterstützt deswegen eine Flusskontrolle mit *Sliding Window* [33, S. 275ff]. Der Empfänger gibt dabei die Größe des *„Fensters“* vor, d. h. wie viele Bytes empfangen werden können. Dieses Fenster wird als *Advertised Window* bezeichnet. Der Sender kann so viele Datenpakete versenden, bis die Größe des Fensters erreicht ist und muss dann auf Empfangsbestätigungen für diese Pakete warten. In TCP ist diese Technik weiterhin mit *kumulativen Empfangsbestä-*

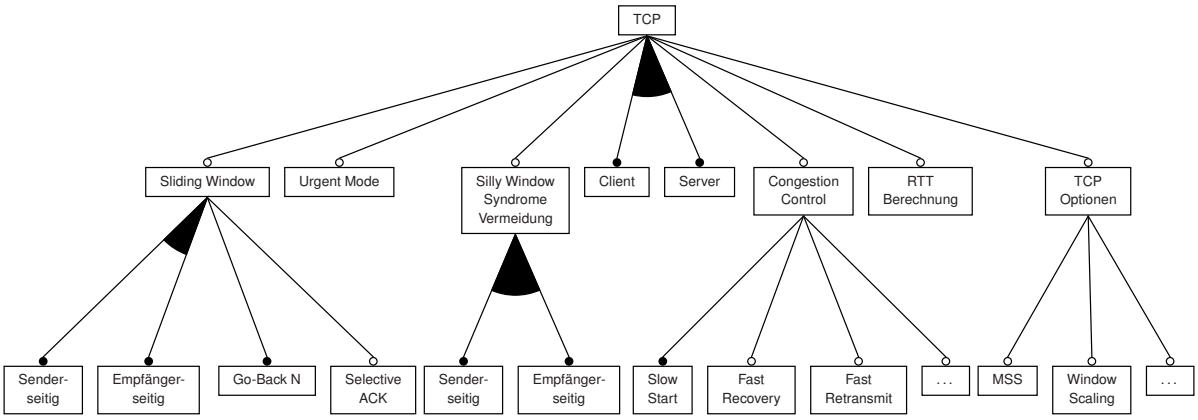

Abbildung 3.11: Konfigurierungsmöglichkeiten innerhalb von TCP

tigungen kombiniert. Treffen beim Sender mehrere Pakete, z. B. mit den Nummern *1, . . . , N*, ein, genügt eine Empfangsbestätigung für das Paket mit der Nummer *N*. Damit werden alle Pakete bis zur Nummer *N* gleichzeitig bestätigt. Geht beispielsweise das *1.* Paket verloren, wird überhaupt keine Empfangsbestätigung versendet und der Sender beginnt nach einer gewissen Zeit alle *N* Pakete erneut zu versenden. Aus diesem Grund wird dieses Vorgehen mit *Go-Back-N* bezeichnet. Das Merkmal des Sliding Windows lässt sich weiterhin in ein Sender- und Empfängermerkmal unterteilen. Das senderseitige Merkmal ermöglicht das Versenden von mehreren Paketen hintereinander, ohne dabei auf eine Empfangsbestätigung zu warten. Auf der Empfängerseite bedeutet das Merkmal, mehrere Pakete zwischenzuspeichern und kumulative Empfangsbestätigungen zu erzeugen. Ein System, das große Datenmengen empfängt, jedoch selbst keine Daten versendet, benötigt folglich nur eine empfängerseitige Unterstützung für das Sliding Window. Mit *Selective Acknowledgements (Selective ACK)* [45] gibt es eine Erweiterung des Go-Back-N-Prinzips, so dass gezielt einzelne Pakete bestätigt werden können. Im Falle eines Paketverlustes müssen dann nur genau die Pakete erneut versendet werden, die tatsächlich verlorenen gegangen sind. Das Sliding Window selbst stellt bei TCP ein optionales Merkmal dar, denn die Festlegung der Fenstergröße auf genau ein Paket entspricht der Funktionalität eines Stop-and-Wait-Verfahrens.
Neben den Nutzdaten, die über eine TCP-Verbindung übertragen werden, können zusätzlich dringende Daten im *Urgent Mode* mit einer höheren Priorität übermittelt werden, die der Empfänger bevorzugt behandelt. Viele Anwendungen machen von diesem Merkmal allerdings keinen Gebrauch, so dass es in dieser Domäne als optionales Merkmal behandelt wird.
Das *Silly Window Syndrome* (SWS) [46] tritt auf, wenn der Empfänger eines Datentransfers langsamer als der Sender ist und die Freigabe von geringem Pufferspeicher sofort dem Sender mitteilt. Das führt dazu, dass nur kleine anstatt große Datenpakete übertragen werden, wodurch der Durchsatz enorm beeinträchtigt wird. Für dieses Problem gibt es zwei Lösungen, die auch zusammen benutzt werden können. Zum einen kann der Empfänger das SWS vermeiden, indem er die Freigabe von Pufferspeicher erst ab einer definierten Größe, z. B. einem vollen TCP-Segment, dem Sender mitteilt. Zum anderen

kann der Sender das SWS verhindern, indem er keine kleinen Pakete versendet sondern wartet, bis das Advertised Window eine Mindestgröße erreicht, wie z. B. die Größe eines vollen TCP-Segmentes.
Zur Vermeidung einer Überlast in einem Netzwerk verfügt TCP über eine *Congestion Control*. Dabei handelt es sich um verschiedene Algorithmen, die das Senden von TCP-Segmenten unter bestimmten Bedingungen begrenzen können. Mit *Slow Start* [47] steht ein grundlegendes Verfahren bereit, das diese Funktionalität realisiert. „It operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgements are returned by the other end“ [33, S. 285]. Dazu wird ein sogenanntes *Congestion Window* (CW) benutzt, das anfangs der Größe eines einzelnen TCP-Segmentes entspricht. Wenn eine Empfangsbestätigung eintrifft, so wird das CW um die Größe eines weiteren Segmentes erhöht. Wird der Verlust eines Segmentes festgestellt, d. h. die zugehörige Empfangsbestätigung bleibt aus, wird das CW zurück auf den Anfangswert gesetzt. Die Datenmenge, die der Sender versenden darf, wird zu jeder Zeit durch das CW limitiert. Weiterhin wird ein *Threshold* verwaltet, der das Wachstum des CWs begrenzt, indem beim Eintreffen einer Empfangsbestätigung das CW nur noch um einen kleinen konstanten Wert erhöht wird, falls das CW größer als der Threshold ist. Der Wert dieses Thresholds wird bei dem Verlust eines Segmentes auf die Hälfte des aktuellen CWs gesetzt. Das häufige Auftreten von verlorenen Segmenten führt bei Slow Start also dazu, dass weniger Pakete versendet werden. Mit *Fast Retransmit* und *Fast Recovery* [33, S. 312f] gibt es zusätzliche Optimierungen, die den Durchsatz bei sporadischen Paketverlusten steigern. Das Merkmal der Congestion Control ist abhängig von dem Merkmal des senderseitigen Sliding Windows, da erst das aufeinander folgende Versenden von mehreren Paketen reguliert wird. Weiterhin benötigt ein System, das nur Daten empfängt, keine Congestion Control. Deswegen ist dieses Merkmal in dieser Domäne optional.
Ein weiteres optionales Merkmal ist die Berechnung der *Round-Trip-Time* (RTT). Falls ein Paket verloren geht, muss es erneut versendet werden. Dazu wartet der Sender eine gewisse Zeit nach dem Senden auf eine Empfangsbestätigung. Trifft diese nicht ein, wird das betroffene Paket noch einmal gesendet. Für einen optimalen Durchsatz sollte diese Wartezeit weder zu groß noch zu klein sein, sondern etwa der durchschnittlichen Zeit entsprechen, die bis zum Eintreffen einer Empfangsbestätigung vergeht. Wie diese Round-Trip-Time genau berechnet werden kann, wird in Abschnitt 5.2.4 erläutert.
Die Funktionalität von TCP kann durch *TCP Optionen* nochmals ergänzt werden. Die wichtigste Option dabei ist die der *Maximum Segment Size* (MSS). Damit lässt sich die Größe der TCP-Segmente optimal an die *Maximum Transmission Unit* (MTU) der verwendeten Netzwerktechnologie anpassen. Weitere Optionen, wie z. B. das *Window Scaling* [48], dienen der Durchsatzsteigerung von Hochgeschwindigkeitsnetzen.

Speicherverwaltung

Einer der wichtigsten Faktoren für die Performanz eines TCP/IP-Stacks ist dessen Speicherverwaltung, die festlegt, wie Daten im Speicher organisiert sind. Pakete, die empfangen werden, müssen in sogenannten *Puffern* abgelegt werden, bis sie von der

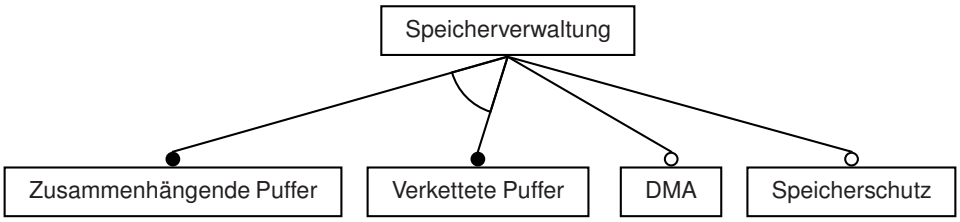

Abbildung 3.12: Varianten der Speicherverwaltung

Anwendungsschicht ausgelesen werden. Das Versenden geschieht ebenfalls über Puffer, die mit Daten befüllt und anschließend einem Netzwerkkartentreiber übergeben werden. Da eingebettete Systeme in der Regel nur über sehr begrenzten Speicherplatz verfügen, muss in dieser Domäne sehr effizient damit gearbeitet werden. Abbildung 3.12 zeigt einige Merkmale einer solchen Speicherverwaltung. Es gibt grundsätzlich zwei alternative Strategien, die in der Abbildung durch den nicht-ausgefüllten Bogen gekennzeichnet sind. Entweder werden *zusammenhängende* oder *verkettete Puffer* eingesetzt. Letztere bestehen aus einer *verketteten Liste*, die in jedem Listenelement eine feste Menge von Nutzdaten oder Header-Informationen speichert. Diese Art von Puffern wird in den BSD-Systemen eingesetzt, in denen, je nach BSD-Variante, ein Listenelement 128 oder 256 Byte an Daten umfasst und als *mbuf*[20] [34, S. 31ff] bezeichnet wird. Ein komplettes Datenpaket besteht somit häufig aus mehreren dieser Listenelemente. Der Vorteil an dieser Art von Speicherverwaltung liegt in der enormen Flexibilität. Einerseits kann ein Datenpaket beliebig erweitert werden, indem z. B. zusätzliche Header-Informationen während des Sendevorgangs vorne angehängt werden. Andererseits wird durch diese verketteten Listen die Komplexität der Netzzugangsschicht erhöht, da ein Netzwerkkartentreiber ebenfalls damit umgehen muss. Weiterhin ist ein komplettes Paket möglicherweise durch die einzelnen Listenelemente über den gesamten Speicher verteilt, so dass bei dem Einsatz von *Caches* mehrere *Cache-Misses* bei einem Zugriff auf ein einzelnes Paket auftreten können. Die alternative Strategie zu den verketteten Puffern ist die Verwendung von zusammenhängenden Puffern, die linear im Speicher angeordnet sind. Dabei wird für jedes Paket ein Speicherbereich reserviert, in den es vollständig hinein passt. Falls beim Sendevorgang Header-Informationen vorne hinzugefügt werden, muss bereits vorher genügend Platz dafür reserviert sein. Diese Vorgehensweise verwendet Linux, dessen Netzwerkpuffer als *sk_buf*[21] [49] bezeichnet werden. Durch die örtliche Lokalität eines Paketes gestaltet sich ein Netzwerkkartentreiber einfacher und die Verwendung von Caches wird effizienter. Neben diesen beiden Strategien gibt es noch weitere optionale Merkmale. Mit *DMA*[22] liest eine Netzwerkkarte selbstständig den Inhalt der verwendeten Puffer aus und schreibt ebenfalls selbstständig die Daten eines empfangenen Paketes hinein, ohne dabei den Hauptprozessor zu belasten. Wird ein Betriebssystem mit *Speicherschutz* verwendet, muss der TCP/IP-Stack diesen ebenfalls unterstützen, da sich ein Netzwerkkartentreiber, im Gegensatz zur Anwendungsschicht, in der Regel in einem privilegierten Modus befindet.

[20] Memory Buffer
[21] Socket Buffer
[22] Direct Memory Access

4 Domänenentwurf

In diesem Kapitel wird auf der Grundlage des erstellten Domänenmodells aus dem vorigen Kapitel eine Referenzarchitektur für die Software-Produktlinie eines TCP/IP-Stacks für eingebettete Systeme entwickelt. Zuerst wird die Grundidee der Referenzarchitektur vorgestellt (siehe 4.1). Daran schließt eine Betrachtung dieser Architektur unter dem Gesichtspunkt der Hierarchie an, indem die Referenzarchitektur in Schichten mit abgegrenzten Funktionen eingeteilt wird und einzelne Komponenten der Software-Produktlinie detailliert modelliert werden. Abschnitt 4.3 diskutiert eine konfigurierbare Speicherverwaltung, während die Vorstellung eines Entwurfs der Netzzugangsschicht in Abschnitt 4.4 erfolgt. Dabei wird besonders auf die Vorteile eines aspektorientierten Entwurfs eingegangen. Aufbauend darauf wird eine Erweiterung um die Internetschicht in Abschnitt 4.5 vorgenommen. Der Entwurf einer Schichtenarchitektur kann sowohl *Top-Down* als auch *Bottom-Up* durchgeführt werden. Beide Vorgehensweisen werden in diesem Domänenentwurf verfolgt und in den beiden separaten Abschnitten 4.6 und 4.7 beschrieben.

4.1 Referenzarchitektur

An die Architektur eines TCP/IP-Stacks für eingebettete Systeme werden Anforderungen bzgl. hochgradiger Konfigurierbarkeit und größtmöglicher Effizienz gestellt. Eine klare Strukturierung der Architektur ermöglicht die Realisierung beider Anforderungen. In Abbildung 4.1 ist eine konzeptionelle Sicht auf die Referenzarchitektur dieser Diplomarbeit abgebildet. Die Netzzugangsschicht, welche die Netzwerkhardware sowie die passenden Treiber enthält, wird als *Interface* abstrahiert. Für jedes verwendete Netzwerkgerät gibt es ein zugehöriges Interface. Die Schnittstelle eines solchen Interfaces ist dabei unabhängig von dem entsprechenden Netzwerkgerät, so dass die restlichen Komponenten des TCP/IP-Stacks geräteunabhängig sind. Durch diese Abstraktion wird die Konfigurierbarkeit der Netzzugangsschicht erreicht, da die Netzwerkhardware durch die definierte Schnittstelle des Interfaces austauschbar ist.
Trifft ein neues Datenpaket ein, wird es von dem zugehörigen Interface der Komponente *Demux*[1] übergeben. Das Interface selbst hat dabei keine Informationen über die Art des Datenpaketes und arbeitet völlig protokollunabhängig. Erst in der Demux-Komponente wird dieses Datenpaket analysiert, d. h. es wird untersucht, welche Protokolle benutzt werden. Darüber hinaus wird das Paket auf Integrität überprüft. Die Hauptaufgabe des Demux' besteht darin, das empfangene Paket einem Task zuzuordnen. Gibt es einen Task, der auf dieses Paket wartet, wird es in einen Ringpuffer kopiert, aus dem der Task das

[1] Demultiplexer

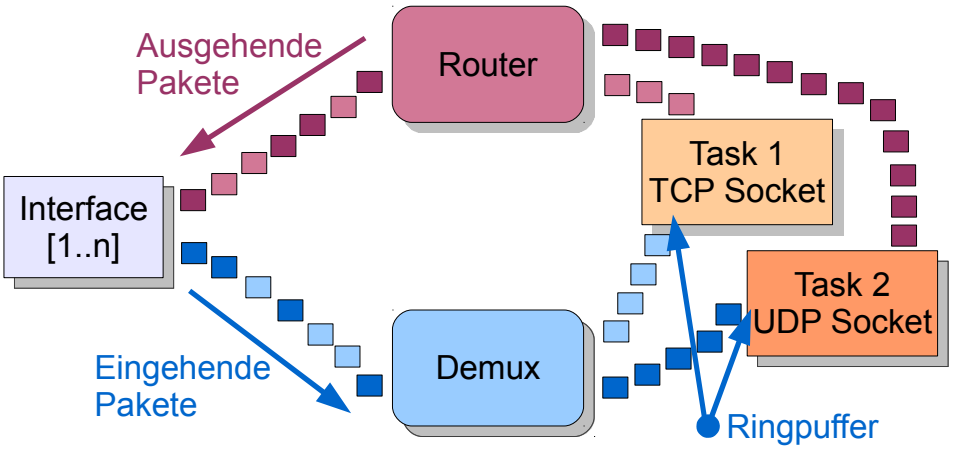

Abbildung 4.1: Referenzarchitektur des TCP/IP-Stacks

Paket zu einem späteren Zeitpunkt auslesen kann. Wird kein passender Task gefunden, so wird das Paket im Demux verworfen.
Versendet ein Task ein neues Datenpaket, wird dieses zuerst vollständig und in Abhängigkeit von dem verwendeten Protokoll erstellt. Danach wird der *Router* benutzt, um das passende Interface zu finden, welches das zu versendende Paket zu seinem Ziel führt. Der daraus resultierende Sendevorgang verläuft vollständig protokollunabhängig, d. h. das Interface versendet Datenpakete, ohne genaue Informationen über deren Inhalt zu besitzen.
Durch die beschriebene Architektur wird die Konfigurierbarkeit der Protokolle erreicht. Die Internetschicht und die Transportschicht sind somit ebenfalls austauschbar, so dass es prinzipiell keine Rolle spielt, welche Kombinationen von IPv4, IPv6, UDP und TCP verwendet werden. Die Komponenten Interface, Demux und Router können einem Betriebssystemkern zugeordnet werden, während der Hauptteil der Protokollverarbeitung direkt in den Tasks stattfindet. Der Anteil eines Betriebssystemkerns an dieser Referenzarchitektur beschränkt sich auf das Versenden und Empfangen von Datenpaketen, die prinzipiell protokollunabhängig sind. Dieser Ansatz bietet einige Vorteile gegenüber einem TCP/IP-Stack, der komplett im Betriebssystemkern enthalten ist. Der wichtigste Vorteil ist, dass die Prioritäten von Tasks berücksichtigt werden. Die Protokollverarbeitung findet in dem Laufzeitkontext eines Tasks statt, so dass diejenigen Tasks mit einer höheren Priorität auch bei der Verwendung des TCP/IP-Stacks bevorzugt werden. Eingehende Pakete werden vom Betriebssystemkern lediglich in entsprechende Ringpuffer kopiert, die von den Tasks gemäß der Taskprioritäten ausgelesen werden. Die Einhaltung dieser Prioritäten ist besonders bei eingebetteten Systemen wichtig, die als *Echtzeitsysteme* [50] verwendet werden. Weiterhin kann eine Isolation der einzelnen Tasks stattfinden, indem jeder Task über einen eigenen Ringpuffer und eine eigene Speicherverwaltung verfügt.
Diese Referenzarchitektur basiert auf dem Grundprinzip eines *Network Channels* [51]. Dabei handelt es sich um eine Warteschlange für Datenpakete, die nach dem *Erzeuger-Verbraucher*-Prinzip benutzt wird. Implementiert ein Betriebssystemkern ausschließlich

solche Network Channels und überlässt den Hauptteil der Protokollverarbeitung den Tasks, kann sich dies positiv auf die Performanz des Systems auswirken. In einem Mehrkern- bzw. Mehrprozessorsystem wird eine maximale Parallelisierung erreicht, da die Tasks völlig unabhängig voneinander arbeiten können. Durch die Verwendung von Ringpuffern vereinfacht sich die Unterbrechungssynchronisation, so dass im Idealfall keine Sperrung von Unterbrechungen oder sonstige *Locking*-Mechanismen benötigt werden. Bezüglich der Effizienz und der damit erreichbaren Performanz ist die in Abbildung 4.1 gezeigte Referenzarchitektur für eingebettete Systeme geeignet.

4.2 Funktionale Hierarchie

Die bisher entworfene Referenzarchitektur zeigt lediglich eine konzeptionelle Sicht auf den TCP/IP-Stack. Eine genaue Verbindung zum TCP/IP-Referenzmodell (siehe 3.1) ist noch nicht zu erkennen. Aufgrund der Schichtenarchitektur des TCP/IP-Referenzmodells bietet sich für die Referenzarchitektur des TCP/IP-Stacks ebenfalls eine Schichtenarchitektur an. Als Grundlage dafür dient die Partitionierung in die vier Schichten des TCP/IP-Referenzmodells. Um hochgradige Konfigurierbarkeit zu gewährleisten, ist eine weitere Verfeinerung dieser Partitionierung notwendig. Beispielsweise muss bereits beim Architekturentwurf eine Aufteilung in Sende- und Empfangskomponenten stattfinden, um Systeme zu unterstützen, die lediglich eine dieser beiden Komponenten verwenden. Eine erprobte Methode für den Entwurf einer konfigurierbaren Schichtenarchitektur ist die Erstellung einer funktionalen Hierarchie (siehe 2.5.2). In Abbildung 4.2 ist eine solche Hierarchie der Referenzarchitektur des TCP/IP-Stacks dargestellt, die in sieben Schichten unterteilt ist. Auf der ersten Schicht befinden sich Treiber für Netzwerkgeräte. Diese

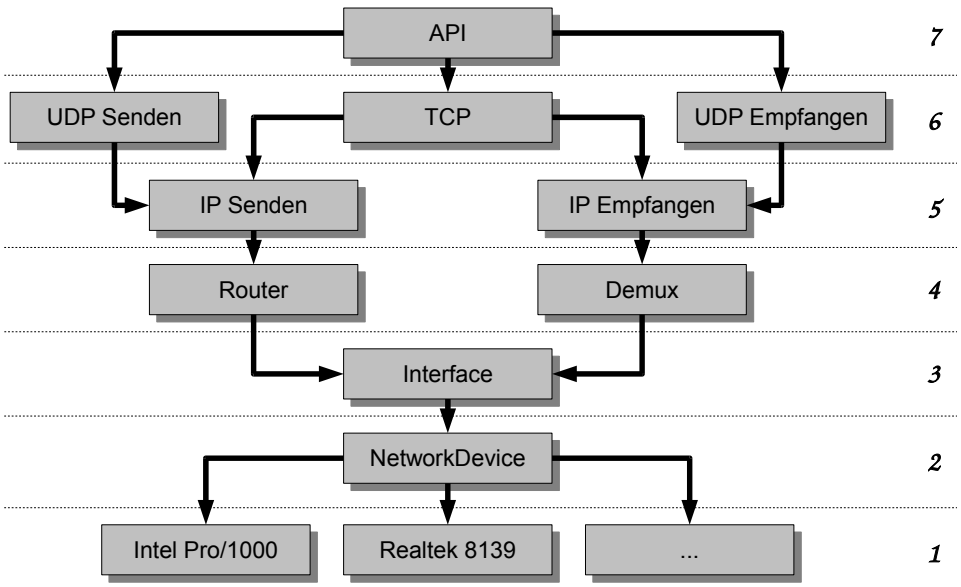

Abbildung 4.2: Funktionale Hierarchie des TCP/IP-Stacks

sind spezifisch für das verwendete Gerät, wie z. B. für die Ethernetgeräte *Intel Pro/1000* oder *Realtek 8139*. Auf der zweiten Ebene wird von diesen Geräten abstrahiert und die Funktion *NetworkDevice* bereitgestellt. Diese Schicht stellt für alle Netzwerkgeräte eine einheitliche Schnittstelle zur Verfügung, indem die gerätespezifischen Funktionen aus der ersten Schicht benutzt werden. Die beiden untersten Ebenen entsprechen der Netzzugangsschicht des TCP/IP-Referenzmodells, da durch das NetworkDevice lediglich Zugang zur Netzwerkhardware ermöglicht wird. In der dritten Schicht wird neue Funktionalität durch das *Interface* hinzugefügt, wie z. B. IP-Adressen. Jedes Interface enthält zusätzlich zu den Funktionen des NetworkDevices eine IP-Adresse und eine IP-Subnetzmaske, sofern IP verwendet wird. Die folgenden Schichten verwenden ausschließlich die durch das Interface bereitgestellten Funktionen. Die dritte Ebene ist somit die erste Schicht, die der Internetschicht des TCP/IP-Referenzmodells zugeordnet werden kann. Auf der vierten Ebene sind die in Abschnitt 4.1 beschriebenen Komponenten Router und Demux angesiedelt, die für das protokollabhängige Senden und Empfangen von Datenpaketen zuständig sind. Diese Komponenten werden für die fünfte Schicht benötigt, in der das Senden und Empfangen von IP-Paketen schließlich realisiert wird. Diese Ebene stellt die letzte Schicht dar, in der Funktionen der Internetschicht des TCP/IP-Referenzmodells enthalten sind, und ermöglicht dadurch eine weitere Abstraktionsebene. Auf die fünfte Ebene bauen die Funktionen TCP und UDP auf. Während UDP in Sende- und Empfangsfunktionen aufgeteilt ist, benötigt TCP aufgrund der Empfangsbestätigungen beide Funktionen. Diese sechste Schicht entspricht somit der Transportschicht des TCP/IP-Referenzmodells. Auf der obersten Ebene ist die *API* zu finden, die einer Anwendung Zugang zum TCP/IP-Stack erlaubt. Die Anwendungsschicht des TCP/IP-Referenzmodells ist in dieser funktionalen Hierarchie nicht direkt repräsentiert, sondern mit der API wird lediglich eine Schnittstelle definiert, die eine Abstraktion des gesamten TCP/IP-Stacks für eine Anwendung darstellt.

4.3 Speicherverwaltung

In allen gerade beschriebenen Schichten des TCP/IP-Stacks werden Datenpakete verarbeitet, die im Speicher repräsentiert werden müssen. Damit verwendet jede Schicht direkt oder indirekt die Speicherverwaltung des TCP/IP-Stacks, die somit ein querschneidender Belang ist. In Abschnitt 3.3.3 befindet sich bereits die Analyse verschiedener Merkmale einer Speicherverwaltung, aus der hervorgeht, dass zusammenhängende Puffer die effizienteste Realisierung darstellen. Ein Datenpaket, das sich in einem zusammenhängenden Puffer befindet, kann durch einen Zeiger auf den Anfang des Paketes und durch eine Längenangabe repräsentiert werden. Daraus ergibt sich für die Netzzugangsschicht die Möglichkeit, eine besonders einfache Schnittstelle bereitzustellen. Eine Sendefunktion kann beispielsweise die Signatur `send(const void* data, unsigned datasize)` besitzen und benötigt keine weiteren Informationen über die Speicherobjekte. Somit werden Treiber für Netzwerkgeräte deutlich vereinfacht und portabler.
Da eingebettete Systeme in der Regel über keine dynamische Speicherverwaltung verfügen, ist es nicht möglich, Puffer für Datenpakete einem *Heap* zu entnehmen. Häufig werden in eingebetteten Systemen sogenannte *Pools* verwendet. Das sind statisch ange-

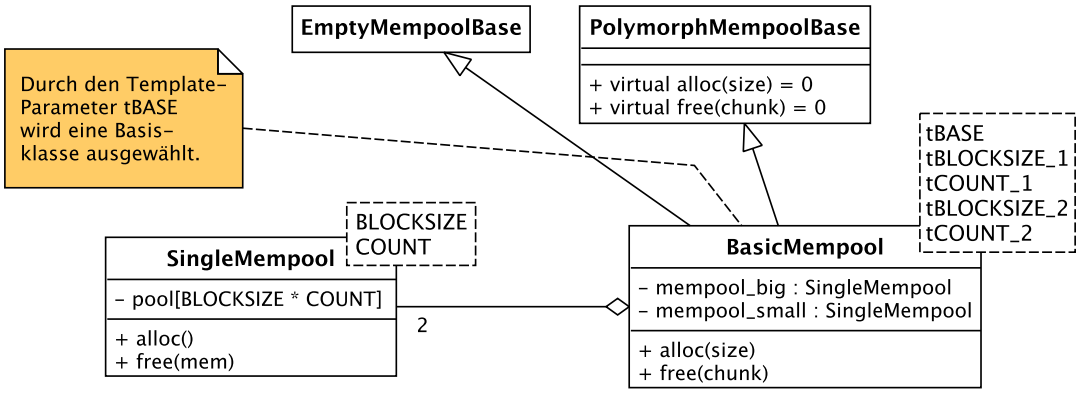

Abbildung 4.3: Entwurf der Speicherverwaltung

legte Speicherbereiche fester Größe, die eine bestimmte Anzahl von Speicherelementen enthalten. Durch die feste Größe der Speicherelemente wird eine Fragmentierung des Speichers vermieden. Aus diesem Grund eignet sich der Einsatz von Pools ausgezeichnet für die Speicherverwaltung dieses TCP/IP-Stacks. Es gibt jedoch zwei wichtige Kriterien: einerseits hängt die Performanz eines Datentransfers maßgeblich von der Gesamtgröße des verfügbaren Speicherbereichs ab – andererseits wird die maximale Paketgröße durch die Größe eines Speicherelementes limitiert. Ist ein Speicherelement zu klein, lässt sich die MTU der Netzwerkhardware nicht erreichen und die Effizienz sinkt. Ist ein Speicherelement zu groß, wird beim Transfer von kleinen Paketen der „übrige“ Speicherplatz verschwendet. Eine optimale Größe gibt es nicht, denn häufig werden große und kleine Pakete gleichzeitig verschickt. „Packet size distributions tend to be bimodal“ [52, S. 5]. Dies lässt sich anhand eines TCP-Datentransfers veranschaulichen. Der Nutzdatenstrom wird über Pakete maximaler Größe transportiert, während die Empfangsbestätigungen in kleinen Paketen übermittelt werden. Deswegen wird eine Speicherverwaltung mit Pools benötigt, die große und kleine Pakete effizient unterstützt und deren Größe und Anzahl am besten konfigurierbar sind.
Die Referenzarchitektur aus Abschnitt 4.1 stellt eine weitere Anforderung an die Speicherverwaltung, die demnach die Isolation von Tasks unterstützen muss. Daraus folgt, dass jeder Task einen eigenen, konfigurierbaren Pool benötigt, sofern Isolation erforderlich ist. Den Entwurf einer solchen Speicherverwaltung zeigt Abbildung 4.3. Die Konfigurierbarkeit wird in diesem Entwurf durch Templates realisiert, um gleichzeitig verschiedene Instanzen der Speicherverwaltung mit unterschiedlichen Konfigurationen zu ermöglichen. Ausgangspunkt ist dabei die Klasse `BasicMempool`, die über fünf Template-Parameter konfiguriert werden kann. Der erste Parameter `tBASE` definiert eine Basisklasse, von der `BasicMempool` erbt. Die weiteren vier Parameter werden zur Konfiguration der Größe der Speicherelemente (`tBLOCKSIZE_*`) sowie deren Anzahl (`tCOUNT_*`) benutzt. Intern werden dadurch zwei Objekte vom Typ `SingleMempool` angelegt, dessen Template-Parameter durch die instanziierende Klasse bestimmt werden. Methodenaufrufe von `alloc(size)` und `free(chunk)` in `BasicMempool` werden zu den entsprechenden `SingleMempool`-Objekten delegiert. Durch einen Klassenalias wird konfiguriert, ob ein System gleichzeitig

Mempools unterschiedlicher Größen unterstützt oder ob es einen statisch festgelegten Mempool gibt, indem ein Typ `Mempool` definiert wird, der von dem Rest des TCP/IP-Stacks benutzt werden kann. Werden unterschiedliche Mempools benötigt, wird der Klassenalias „`typedef PolymorphMempoolBase Mempool`“ angelegt und der Zugriff erfolgt somit über virtuelle Methoden. Die Instanziierung eines Mempools erfolgt mit `BasicMempool<PolymorphMempoolBase, ...>`. Wird hingegen nur eine Variante verwendet, so wird diese durch „`typedef BasicMempool<EmptyMempoolBase, ...> Mempool`“ definiert. Letztere ist durch die Vermeidung der virtuellen Methoden effizienter, während die erste Variante über deutlich mehr Flexibilität verfügt.

4.4 Netzzugangsschicht

Aus der funktionalen Hierarchie (siehe 4.2) lässt sich für die Netzzugangsschicht direkt ein Klassendiagramm ableiten, welches in Abbildung 4.4 gezeigt ist. Die Schichten eins bis drei der funktionalen Hierarchie werden durch die C++-Namensräume `hw::dev`[2], `hw::hal`[3] und `ipstack` explizit hervorgehoben, um sichtbare Übergänge (siehe 2.5.1) zu ermöglichen. Jede Schicht enthält Klassen, die den Funktionen der funktionalen Hierarchie entsprechen. Die Methoden dieser Klassen sind zur besseren Übersicht in diesem Klassendiagramm auf `getType()`, `send(data, datasize)` und `hasBeenSent(addr)` reduziert. Die erste Methode klassifiziert ein Gerät, indem sie anzeigt, ob es sich um ein Gerät vom Typ Ethernet, Loopback oder Ähnliches handelt. Die Methode `send(data, datasize)`

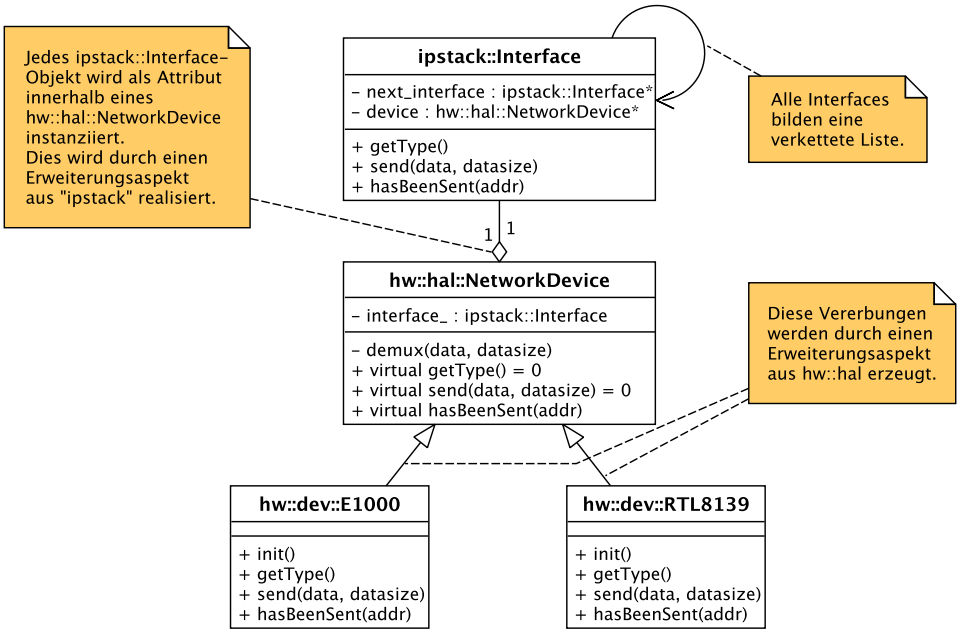

Abbildung 4.4: Vereinfachtes Klassendiagramm der Netzzugangsschicht

[2]hardware devices
[3]hardware abstraction layer

realisiert die Sendefunktionalität, die je nach Gerät synchron oder asynchron ausfallen kann. Wird ein asynchroner Sendevorgang durchgeführt, kehrt diese Methode sofort zum Aufrufer zurück, der zu einem geeigneten Zeitpunkt mit `hasBeenSent(addr)` feststellen kann, ob der asynchrone Sendevorgang abgeschlossen ist. Die Gerätetreiberklassen, hier exemplarisch `E1000` und `RTL8139`, implementieren jeweils diese Methoden. In der darüber liegenden Schicht wird durch die Basisklasse `NetworkDevice` eine einheitliche Schnittstelle für alle Gerätetreiber bereitgestellt. Diese Schnittstelle umfasst ebenfalls die drei o. g. Methoden, die in dieser Klasse jedoch virtuell sind und damit von den erbenden Gerätetreiberklassen überschrieben werden. Diese Verebungsbeziehung wird von einem Erweiterungsaspekt der Schicht `hw::hal` erzeugt, der in Quelltext 4.1 gezeigt ist.

```
aspect NetworkDevice_Inheritance_Aspect {
  pointcut devices() = "hw::dev::E1000" || "hw::dev::RTL8139";

  //advice all above mentioned devices to derive from 'NetworkDevice'
  advice devices() : slice class : public hw::hal::NetworkDevice;
};
```

Quelltext 4.1: Erweiterungsaspekt der schichtenübergreifenden Vererbung

Dieser Aspekt stellt sicher, dass die Klassenstruktur der funktionalen Hierarchie entspricht, da die Gerätetreiberklassen dadurch unabhängig von der darüber liegenden Schicht sind. Die Klasse `NetworkDevice` enthält weiterhin die Funktion `demux(data, datasize)`, die nicht von den Gerätetreiberklassen überschrieben wird. Wird ein neues Paket von einem Gerät empfangen, so wird mit einem Upcall-Aspekt der Kontrollfluss von der Unterbrechungsbehandlung des Gerätes zu dieser Funktion übergeben. Dieser Aspekt ist ebenfalls Bestandteil der Schicht `hw::hal` und wird in Quelltext 4.2 vereinfacht dargestellt.

```
aspect NetworkDevice_IRQ_Binding_E1000 {
  advice execution("% hw::irq::IRQ_E1000::Handler()") : before() {
    const void* data = hw::dev::E1000::Inst().nextData();
    unsigned datasize = hw::dev::E1000::Inst().nextDataLength();
    hw::dev::E1000::Inst().demux(data, datasize);
  }
};
```

Quelltext 4.2: Upcall-Aspekt der Unterbrechungsbehandlung

Der Empfang eines Paketes ist somit stets an dem Aufruf der Funktion `demux(...)` innerhalb des zugehörigen Objektes der Klasse `NetworkDevice` zu erkennen. Auf der dritten Schicht befindet sich die Klasse `Interface`, deren Objekte als Attribute innerhalb der Klasse `NetworkDevice` instanziiert werden. Diese Instanziierung wird ebenfalls durch einen Erweiterungsaspekt realisiert, der ein `Interface`-Objekt als Klassenfragment der Klasse `NetworkDevice` hinzufügt. Durch diesen Erweiterungsaspekt wird erneut die Einhaltung der funktionalen Hierarchie sichergestellt, indem dieser Aspekt zur Schicht

`ipstack` gehört. Jedes `Interface`-Objekt enthält einen Zeiger auf das beherbergende `NetworkDevice`-Objekt und leitet alle Methodenaufrufe per Delegation dahin weiter. Existieren mehrere Objekte der Klasse `Interface`, sind sie als verkettete Liste gemäß der Initialisierung der zugehörigen Gerätetreiber angeordnet.
Dieser Entwurf der Netzzugangsschicht basiert auf der Verwendung von virtuellen Methoden innerhalb der Klasse `NetworkDevice` und dem damit verbundenen objektorientierten Mechanismus der Polymorphie (siehe 2.4.2). In einem System mit verschiedenen Netzwerkgeräten wird dadurch die unvermeidbare Fallunterscheidung beim Zugriff auf die unterschiedlichen Geräte realisiert. In einem System mit nur einem Netzwerkgerät werden hingegen unnötige Speicher- und Laufzeitkosten verursacht. In Abschnitt 5.1 wird gezeigt, wie durch aspektorientierte Programmierung dieser Entwurf dennoch kostenneutral implementiert werden kann, selbst wenn nur ein Netzwerkgerät vorhanden ist.

4.5 Internetschicht – Minimale Erweiterung

Die Klasse `Interface` (siehe Abbildung 4.4) bildet eine Abstraktion für alle Netzwerkgeräte, auf die der TCP/IP-Stack zugreift. Die Verwaltung der `Interface`-Objekte wird von der Klasse `Router` durchgeführt. Diese Klasse entspricht der gleichnamigen Funktion der funktionalen Hierarchie (siehe 4.2) und realisiert die in Abschnitt 4.1 beschriebene Funktionalität. Durch die Verkettung aller `Interface`-Objekte genügt es, einen Zeiger auf den Anfang dieser Liste abzuspeichern, um über alle `Interface`-Objekte iterieren zu können. In Abbildung 4.5 ist ein vereinfachtes Klassendiagramm gezeigt, das die Klasse `Router` und dessen Relationen zu anderen Klassen und Aspekten veranschaulicht. Die Notation der Aspekte in diesem und den folgenden Klassendiagrammen ist an Lohmann [22, S. 129] angelehnt. Der ausgefüllte Kreis an einem Ende einer Assoziation symbolisiert einen Join-Point, der von einem Aspekt beeinflusst wird. Durch das Entwurfsmuster *Singleton* [17, S. 157ff] wird erreicht, dass es genau ein Objekt der Klasse `Router` gibt,

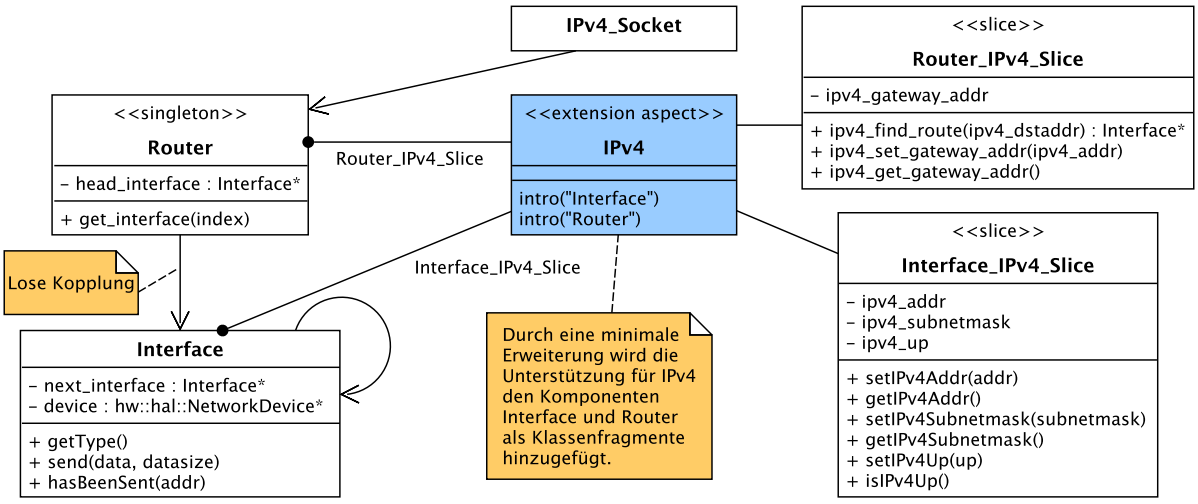

Abbildung 4.5: Minimale Erweiterung des Routers um IPv4-Funktionalität

das wie ein globales Objekt verwendbar ist. Die Funktionalität dieser Klasse beschränkt sich auf die öffentliche Methode `get_interface(index)`, welche die Iteration über alle `Interface`-Objekte ermöglicht. Die Initialisierung dieser Klasse sowie die Verkettung der `Interface`-Objekte untereinander erfolgt nach dem aspektorientierten Entwurfsprinzip der losen Kopplung (siehe 2.5). Dazu wird eine Methode mit der Signatur `init()` benutzt. Wenn in einer von `hw::hal::NetworkDevice` erbenden Klasse (siehe Abbildung 4.4) diese Methode ausgeführt wird, wird das entsprechende Gerät in die verkettete Liste aufgenommen und der `head_interface`-Zeiger des Routers aktualisiert. In Quelltext 4.3 ist der Binding-Aspekt zu sehen, der für dieses Entwurfsprinzip eingesetzt wird. Somit lässt sich die Anzahl der Netzwerkgeräte sehr einfach variieren, ohne dass Änderungen am TCP/IP-Stack oder den Geräten selbst durchgeführt werden müssen. Darüber hinaus ist die Anzahl von gleichzeitig verwendbaren Netzwerkgeräten prinzipiell unbegrenzt.

```
aspect Router_Aspect {
  advice execution("% ...::init()") &&
         within(derived("hw::hal::NetworkDevice")) &&
         that(device) : after(hw::hal::NetworkDevice& device){

    Router& router = Router::Inst(); //get singleton instance

    //initialize interface
    Interface& interface = device.interface_;
    interface.setNetworkDevice(&device); //initialize for delegation

    //insert at front of linked list
    interface.next_interface = router.head_interface;
    router.head_interface = &interface;
  }
};
```

Quelltext 4.3: Lose Kopplung der Interfaces durch einen Binding-Aspekt

Die vierte Version des Internetprotokolls (IPv4) wird in diesem TCP/IP-Stack als minimale Erweiterung hinzugefügt. Die Klasse `Interface` wird durch das Klassenfragment `Interface_IPv4_Slice` um drei private Attribute erweitert, die aus einer IPv4-Adresse, einer IPv4-Subnetzmaske und einer Statusvariablen bestehen. Weiterhin werden öffentliche Zugriffsmethoden auf diese Attribute hinzugefügt. Aufgrund dessen kann jedem Netzwerkgerät eine IPv4-Adresse zugewiesen werden, indem der Router benutzt wird, um über alle `Interface`-Objekte zu iterieren und die passenden Attribute festzulegen. Die Klasse `Router` wird durch das Klassenfragment `Router_IPv4_Slice` um eine *Gateway-Adresse* erweitert, die zum Senden benutzt wird, falls die Zieladresse nicht in einem lokalen Netz liegt. Die Methode `ipv4_find_route(ipv4_dstaddr)` ermöglicht die Suche nach einem passenden Netzwerkgerät, das zu einer gewünschten IPv4-Zieladresse führt. Durch den Erweiterungsaspekt `IPv4` wird eine Abhängigkeit der Klassen `Interface` und `Router` von dem Internetprotokoll vermieden und das Prinzip der funktionalen Hierarchie umgesetzt. Eine Erweiterung um ein anderes Protokoll, wie z. B. IPv6, ist problemlos möglich und kann analog entworfen werden. Die Unterstützung für IPv4 in

den bisher beschriebenen Komponenten wird für die darauf aufbauende Schicht benötigt, die durch die Klasse `IPv4_Socket` repräsentiert wird und das Senden und Empfangen von IPv4-Paketen ermöglicht. Zu einer besseren Übersicht ist diese Klasse in Abbildung 4.5 als leere Klasse dargestellt. Sie wird im nächsten Abschnitt detailliert erläutert.

4.6 Top-Down

Die Schichten fünf und sechs der funktionalen Hierarchie (siehe 4.2) stellen die Funktionen der Protokolle IP, UDP und TCP dar. Die Version des Internetprotokolls ist dabei nicht festgelegt. UDP und TCP bauen auf dem Internetprotokoll auf und erfordern einen davon getrennten Entwurf, damit verschiedene Versionen des Internetprotokolls genutzt werden können. Des Weiteren ist eine Trennung zwischen der Internetschicht und der Transportschicht beim Entwurf sinnvoll, um eine Duplizierung der Implementierung der Internetschicht in den Protokollimplementierungen von UDP und TCP zu vermeiden. Der Entwurf sollte vielmehr eine gemeinsame Nutzung der Implementierung des Internetprotokolls durch UDP und TCP ermöglichen. Eine Aufteilung dieser Funktionen in unterschiedliche Klassen realisiert diese Entwurfsanforderung durch eine strikte Trennung der einzelnen Protokolle. Somit wird je eine Klasse für IPv4, IPv6, UDP und TCP angelegt. Die Schichtenrelation zwischen diesen Klassen könnte durch Vererbung umgesetzt werden, wenn die Klassen UDP und TCP direkt von IPv4 oder IPv6 erben. Damit wird allerdings die Version des Internetprotokolls ebenfalls direkt festgelegt und ist nicht mehr ohne weiteres konfigurierbar. Eine konfigurierbare Schichtenrelation mittels Vererbung ließe sich durch Templates über *Mixin-Layer* [53] realisieren. Dabei verfügen die erbenden Klassen, hier UDP und TCP, über einen Template-Parameter, welcher deren Basisklasse definiert. Die Parametrisierung von Klassen durch Templates führt allerdings in der aktuellen[4] Version von AspectC++ dazu, dass diese Klassen nicht mehr durch Aspekte beeinflusst werden können. Aus diesem Grund wird der Ansatz mit Mixin-Layer in dieser Diplomarbeit nicht weiter verfolgt, da der Architekturentwurf eine optimale Verwendung von AOP garantieren soll. Ein alternativer Entwurf ist in Abbildung 4.6 dargestellt. Die Klassen `IPv4_Socket`, `UDP_Socket` und `TCP_Socket` entsprechen der o. g. Aufteilung gemäß der Protokolle. Das Protokoll IPv6 ist in dieser Abbildung nicht vorhanden, kann aber genau wie IPv4 behandelt und durch eine Klasse `IPv6_Socket` repräsentiert werden. Die Relationen zwischen diesen Klassen werden durch weitere Klassen hergestellt. Die Klasse `IPv4_UDP_Socket` erbt somit die Funktionalitäten von `IPv4_Socket` und `UDP_Socket` und erweitert diese selbst nur geringfügig. Im Wesentlichen werden die Methoden dieser Klasse zu der passenden Basisklasse delegiert. Die meisten Methoden sowie der Konstruktor der Basisklassen sind mit einer eingeschränkten Sichtbarkeit (`#: protected`) versehen und können somit lediglich aus einer erbenden Klassen aufgerufen werden. Die eingeschränkte Sichtbarkeit stellt sicher, dass nur die spezialisierten Klassen, die eine Relation zwischen dem Internetprotokoll und einem Transportprotokoll herstellen, von einem Benutzer verwendet werden können.

[4] 15. November 2010

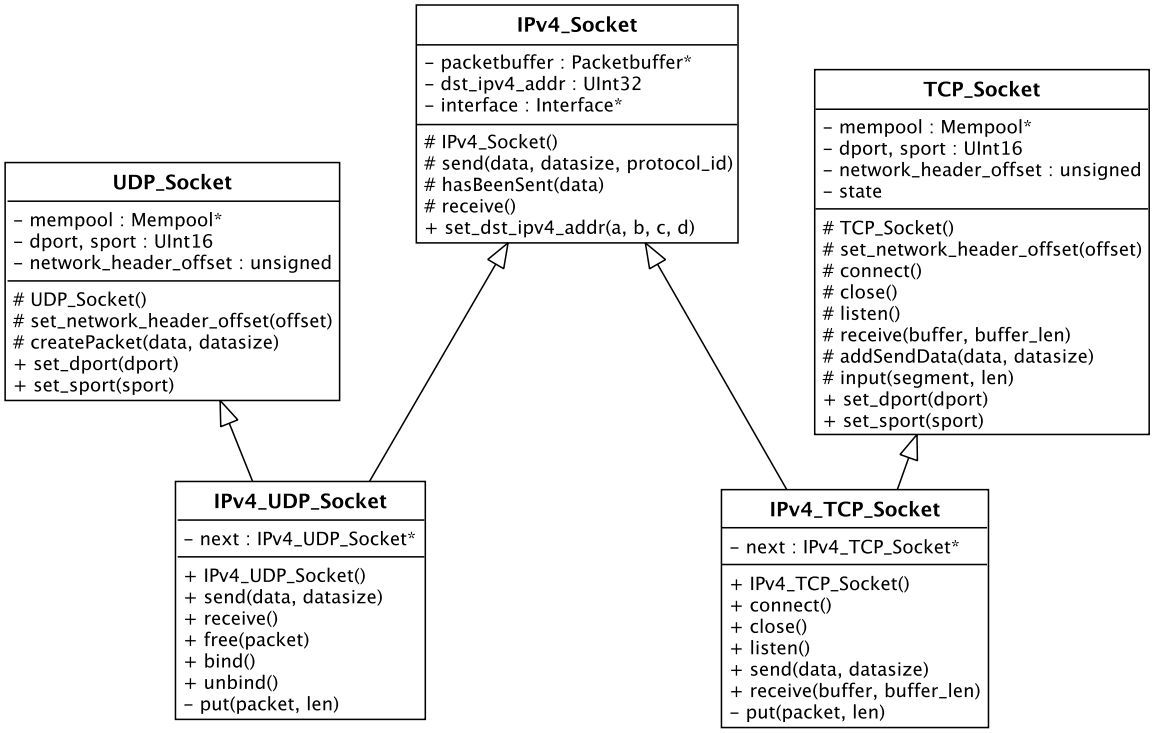

Abbildung 4.6: Entwurf der Protokollklassen

Die Klasse `IPv4_Socket` bildet die Grundlage für darauf aufbauende Komponenten. Durch einen Zeiger auf ein Objekt der Klasse `Interface` wird der Zugriff auf die Netzzugangsschicht durchgeführt, um IPv4-Pakete zu versenden. Die Initialisierung dieses Zeigers erfolgt mit Hilfe des Routers (siehe 4.5), sobald eine IPv4-Adresse mit der Methode `set_dst_ipv4_addr(...)` spezifiziert wird. Für das Empfangen von IPv4-Paketen enthält diese Klasse einen Zeiger auf einen Ringpuffer, der aufgrund des Verwendungszwecks mit `packetbuffer` bezeichnet wird. Dort werden von der Komponente Demux (siehe 4.1) eintreffende Paket abgelegt, die dann mit der Methode `receive` der Klasse `IPv4_Socket` ausgelesen werden können.
Die Speicherverwaltung für jede Verbindung übernimmt die Transportschicht. Dazu wird ein Zeiger vom Typ `Mempool` (siehe 4.3) benutzt, der auf einen beliebigen Pool von Speicherelementen zeigt. Dieser Zeiger wird von den Klassen `UDP_Socket` und `TCP_Socket` verwendet, um Speicherplatz anzufordern, der für das Senden von neuen Paketen notwendig ist. Der Entwurf ermöglicht sowohl das Benutzen eines Mempools durch mehrere Verbindungen (Sockets), als auch eine Isolation von Verbindungen durch den Einsatz von exklusiven Mempools pro Socket. Eine flexible Kombination dieser beiden Varianten ist ebenfalls vorgesehen. Des Weiteren ist in beiden Klassen das Attribut `network_header_offset` enthalten. Damit wird die Größe des Speicherbereichs vorgegeben, der vor den Nutzdaten eines zu versendenden Paketes reserviert wird. Der Wert dieses Attributes wird von einer erbenden Klasse über eine *set-Methode* festgelegt, die

z. B. der Konstruktor von `IPv4_UDP_Socket` mit dem Wert von 20 Byte aufruft. Dieser Methodenaufruf bildet einen nützlichen Join-Point und kann durch optionale Aspekte erweitert werden, um einen größeren Speicherbereich für weitere Header-Informationen, wie z. B. einem Ethernet-Header, zu reservieren. Abgesehen von den Port-Nummern und zugehörigen Zugriffsmethoden haben die Klassen `UDP_Socket` und `TCP_Socket` keine weiteren Gemeinsamkeiten. Die UDP-Klasse stellt ausschließlich eine Methode zur Verfügung, mit der ein UDP-Paket inklusive UDP-Header erstellt werden kann. Die Klasse `TCP_Socket` enthält hingegen eine deutlich umfangreichere Funktionalität, welche durch die Merkmale von TCP vorgegeben ist. Dazu zählen u. a. der Auf- und Abbau von Verbindungen sowie eine zuverlässige Datenübertragung. Weiterführende Informationen zu dieser Klasse befinden sich in Abschnitt 5.2.4. Im Folgenden wird der Entwurf der Klassen `IPv4_UDP_Socket` und `IPv4_TCP_Socket` verfeinert, um optimale Konfigurierbarkeit zu ermöglichen. Abschließend wird auf den Zusammenhang von IPv4 und Ethernet eingegangen, der durch das Protokoll ARP hergestellt wird.

4.6.1 UDP über IPv4

Die Klasse `IPv4_UDP_Socket`, die in Abbildung 4.6 dargestellt ist, lässt sich in Sende- und Empfangsfunktionalität aufteilen. Diese Aufteilung veranschaulicht Abbildung 4.7. Die Klasse `IPv4_UDP_Socket`, die von den beiden in dieser Abbildung nicht gezeigten Basisklassen `IPv4_Socket` und `UDP_Socket` erbt, stellt anfangs keine Funktionalität dar, sondern enthält lediglich einen öffentlichen Konstruktor. Die für das Senden und Empfangen zuständigen Methoden und Attribute werden mit den in diesem Diagramm gezeigten Erweiterungsaspekten als minimale Erweiterungen hinzugefügt. Somit wird die in der funktionalen Hierarchie (siehe 4.2) festgelegte Trennung zwischen dem Senden und Empfangen von UDP-Paketen erreicht. Der Ablauf eines Sendevorgangs ist in dem Sequenzdiagramm in Abbildung 4.8 exemplarisch gezeigt. Dort ist zu erkennen, dass hauptsächlich eine Delegation zu den Basisklassen `UDP_Socket` und `IPv4_Socket` stattfindet. Die zu versendenden Nutzdaten werden um Header-Informationen der jeweiligen Protokolle erweitert und das vollständige Paket wird anschließend über ein `Interface`-Objekt versendet.

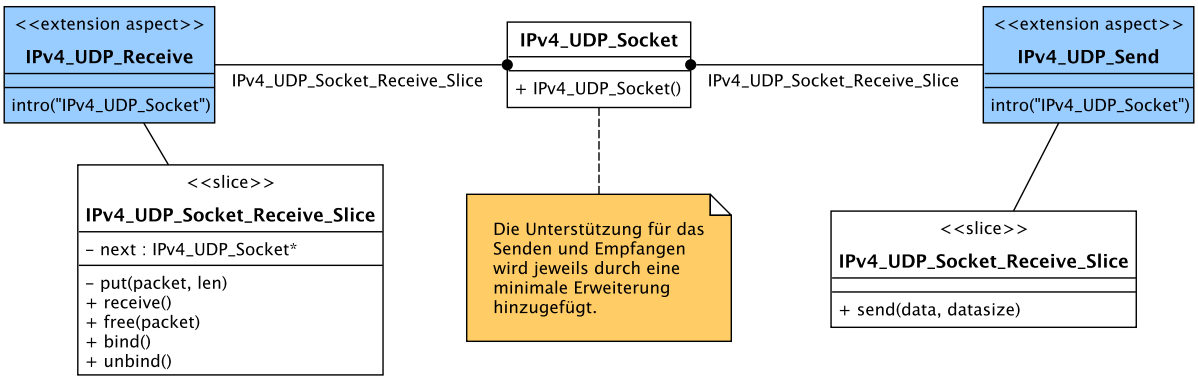

Abbildung 4.7: Entwurf von `IPv4_UDP_Socket` durch minimale Erweiterungen

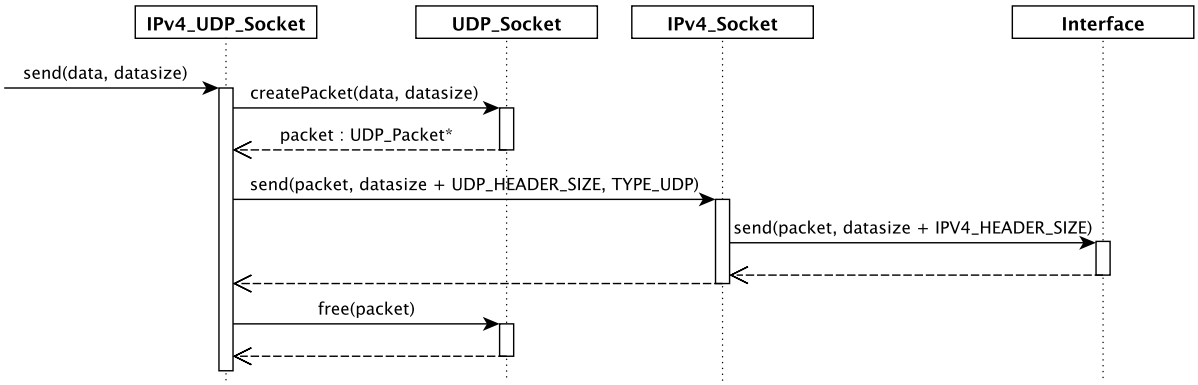

Abbildung 4.8: Senden eines UDP-Paketes über IPv4

Für das Empfangen werden hingegen die Methoden `bind()` und `unbind()` benötigt, die ein Objekt der Klasse `IPv4_UDP_Socket` an der Komponente Demux (siehe 4.1) registrieren bzw. abmelden. An dieser Stelle wird eine verkettete Liste aller Objekte von dieser Klasse verwaltet, die für das Empfangen von UDP Paketen über IPv4 registriert sind. Aus diesem Grund enthält das Klassenfragment `IPv4_UDP_Socket_Receive_Slice` einen Zeiger auf ein folgendes Objekt der Klasse `IPv4_UDP_Socket`. Stellt die Demux-Komponente fest, dass ein eingetroffenes Paket für ein registriertes `IPv4_UDP_Socket`-Objekt bestimmt ist, wird dieses Paket über die Methode `put(packet, len)` übergeben. Diese Methode benutzt den Mempool und Ringpuffer der Basisklassen `UDP_Socket` und `IPv4_Socket`, um das Paket in einen freien Speicherbereich zu kopieren und einen Zeiger darauf in dem Ringpuffer abzulegen. Es entsteht somit ein weiterer nützlicher Join-Point, der dazu benutzt werden kann, durch einen Aspekt einem Task den Empfang eines neuen Paketes zu signalisieren. Dieser Empfangsvorgang findet nach dem gleichen Prinzip ebenfalls in der Klasse `IPv4_TCP_Socket` statt, die im nächsten Abschnitt erläutert wird.

4.6.2 TCP über IPv4

Die Variabilität der Klasse `IPv4_TCP_Socket` fällt deutlich geringer aus als bei der UDP-Variante, da sowohl das Senden als auch das Empfangen protokollbedingt immer benötigt werden. Lediglich eine Unterscheidung in Client- und Serverfunktionalität wird durchgeführt. Die Unterstützung für einen TCP-Client erfordert gegenüber einem TCP-Server in dieser Klasse nur eine zusätzliche Methode, die durch `connect()` repräsentiert wird. Wird diese Methode nicht verwendet, kann sie durch einen optimierenden Compiler (siehe 2.4.1) automatisch entfernt werden. Aus diesem Grund ist die Client-Unterstützung direkt in dieser Klasse vorhanden. Die Funktionalität eines TCP-Servers ist hingegen in dieser Klasse etwas umfangreicher und daher als minimale Erweiterung entworfen. Deshalb wird die Methode `listen()` der Klasse `IPv4_TCP_Socket` in ein Klassenfragment ausgelagert und kann durch einen Erweiterungsaspekt konfiguriert werden. Die Registrierung an der Demux-Komponente ist in diesem Fall ebenfalls etwas aufwändiger. Einzelheiten dazu werden in Abschnitt 4.7.2 beschrieben. Aus der Trennung der Protokolle IP und TCP

in unterschiedliche Klassen resultiert eine erhöhte Komplexität beim Empfangen von Datenpaketen. Dieser Vorgang wird exemplarisch in dem Sequenzdiagramm in Abbildung 4.9 illustriert. Ausgangspunkt ist der Methodenaufruf `receive(buffer, buffer_len)` der Klasse `IPv4_TCP_Socket`. Die dabei verwendeten Parameter sind ein Zeiger auf einen Datenpuffer der Anwendungsschicht sowie die maximale Länge dieses Puffers. Zunächst werden die beiden Parameter an die Basisklasse `TCP_Socket` delegiert und dort abgespeichert. Danach erfolgt das Auslesen eines neuen Paketes aus dem Ringpuffer der anderen Basisklasse `IPv4_Socket` mittels `receive()`. Der IPv4-Header dieses Paketes wird in `IPv4_TCP_Socket` entfernt, so dass ein TCP-Segment übrig bleibt. Dieses Segment wird wiederum der Klasse `TCP_Socket` über eine Methode mit der Signatur `input(segment, len)` übergeben und dort verarbeitet. Zum Schluss werden mittels `copyData(buffer, len)` die empfangenen Nutzdaten in den Datenpuffer kopiert, der zu Beginn dort abgespeichert wurde, um sie dem Aufrufer zur Verfügung zu stellen.

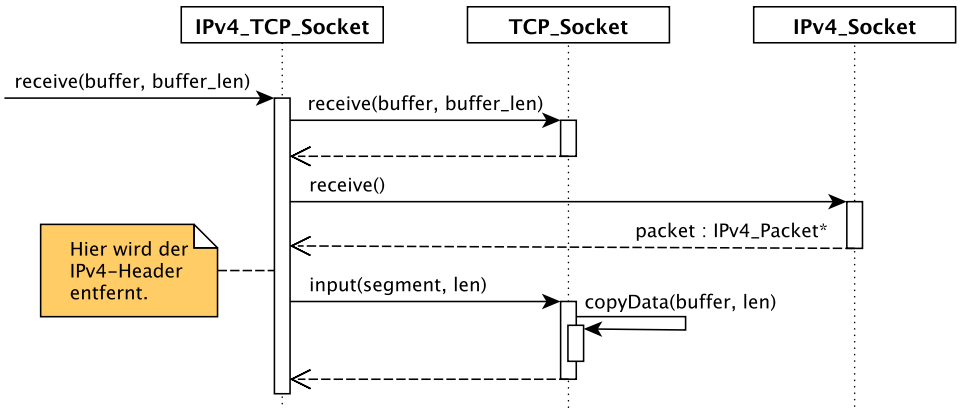

Abbildung 4.9: Empfangen von Nutzdaten mit TCP über IPv4

4.6.3 IPv4 über Ethernet – ARP

Die Kommunikation von mehreren Netzwerkgeräten über ein gemeinsames Medium ist die Grundlage für das Ethernet. Zur Feststellung, für welches Netzwerkgerät des gemeinsamen Mediums ein Datenpaket bestimmt ist, wird jedem Gerät eine eindeutige Ethernet-Adresse zugewiesen. Diese Adressierung ist unabhängig von Adressierungsmechanismen durch andere Protokolle, wie z. B. IPv4. Wird ein Datenpaket über das Ethernet übertragen, so wird zuvor ein 14 Byte umfassender Ethernet-Header versendet, der eine Ziel- und Absenderadresse sowie ein Typ-Feld für den Inhalt des Datenpaketes enthält [32, S. 24f]. Erstellt ein TCP/IP-Stack ein Paket, das über Ethernet versendet werden soll, müssen in dem Speicherbereich vor diesem Paket 14 Byte reserviert werden, damit der Ethernet-Header dort eingefügt werden kann. Da die Paketerstellung durch ein Transportprotokoll wie UDP oder TCP gestartet wird, muss die Transportschicht dafür entsprechend modifiziert werden. Das Typ-Feld des Ethernet-Headers bezieht sich hingegen auf die Internetschicht, d. h. dort wird eine Identifikationsnummer für die Version des

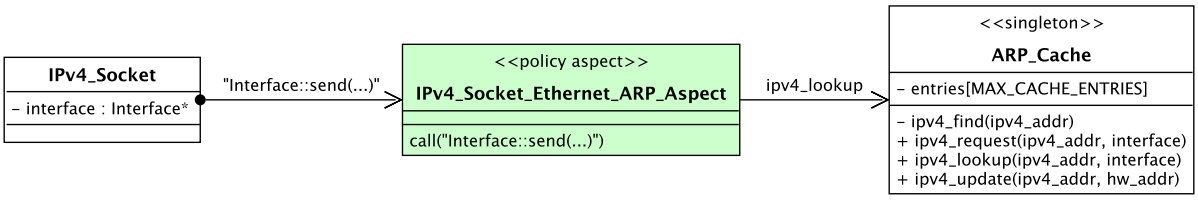

Abbildung 4.10: Entwurf von ARP mit einem Policy-Aspekt

Internetprotokolls eingetragen. Somit existiert eine Abhängigkeit der Netzzugangsschicht, zu der Ethernet gehört, bzgl. der Internet- und Transportschicht und es entsteht ein querschneidender Belang. Die Unterstützung für Ethernet lässt sich dennoch als eigenständige Komponente durch einen Aspekt entwerfen. Ein Advice, der den Parameter der Methode `set_network_header_offset(unsigned)` der Transportschicht-Klassen um 14 Byte erhöht, ermöglicht die Reservierung des Speicherbereichs für den Ethernet-Header. Das Einfügen des Ethernet-Headers vor einem IP-Paket kann ebenfalls durch einen weiteren Advice realisiert werden, indem alle Aufrufe von `Interface::send(...)`, die zusätzlich aus der Klasse `IPv4_Socket` stammen, beeinflusst werden.
Die Bestimmung der Absenderadresse für den Ethernet-Header ist trivial. Für die Zieladresse wird hingegen das Protokoll ARP (siehe 3.3.3) benötigt, das IPv4-Adressen auf Ethernet-Adressen abbildet. Ein Policy-Aspekt ermöglicht die Umsetzung dieses Belangs, der ebenfalls querschneidend ist. In Abbildung 4.10 ist der Policy-Aspekt sowie die Interaktion mit den betroffenen Komponenten abgebildet. Es wird mit `Interface::send(...)` der gleiche Join-Point wie bei der o. g. Einfügung des Ethernet-Headers benutzt, denn an diesem Punkt des Kontrollflusses ist sowohl die IPv4-Adresse als auch die Information, ob überhaupt ein Ethernet-Gerät zum Versenden benutzt wird, bekannt. Findet ein solcher Methodenaufruf statt, übergibt der o. g. Policy-Aspekt die IPv4-Zieladresse dem `ARP_Cache`. Dieser enthält eine Tabelle von Paaren bestehend aus einer IPv4-Adresse und einer zugehörigen Ethernetadresse. In dieser Tabelle wird nach einem passenden Eintrag gesucht. Ist bereits ein Eintrag für die übergebene IPv4-Adresse vorhanden, wird die zugeordnete Ethernet-Adresse als Zieladresse des Ethernet-Headers für das zu versendende IPv4-Paket verwendet. Gibt es keinen derartigen Eintrag, wird ein ARP-Request generiert und auf eine Antwort (ARP-Reply) gewartet. Trifft innerhalb einer Sekunde keine Antwort ein, wird dieser Vorgang drei Mal wiederholt und dann ggf. abgebrochen. Wenn eine Antwort rechtzeitig ankommt, wird die entdeckte Ethernet-Adresse mit der entsprechenden IPv4-Adresse als Paar in der Tabelle gespeichert, um eine erneute Suche danach zu vermeiden. Anschließend kann die Ethernet-Adresse wie im ersten Fall für das Versenden des IPv4-Paketes benutzt werden.
Der Aufbau des ARP-Caches ist hierbei sehr modular gestaltet, um die in Abschnitt 3.3.3 geforderte Variabilität zu unterstützen. Es gibt stets genau einen ARP-Cache, der aus diesem Grund als Klasse mit dem Entwurfsmuster *Singleton* entworfen ist. Diese Klasse enthält anfangs keine Funktionalität, sondern lediglich eine leere Methode `clear()`, die einen expliziten Join-Point zum Löschen des gesamten Caches darstellt. Abbildung 4.11 zeigt einen aspektorientierten Entwurf, der dem Entwurfsprinzip der

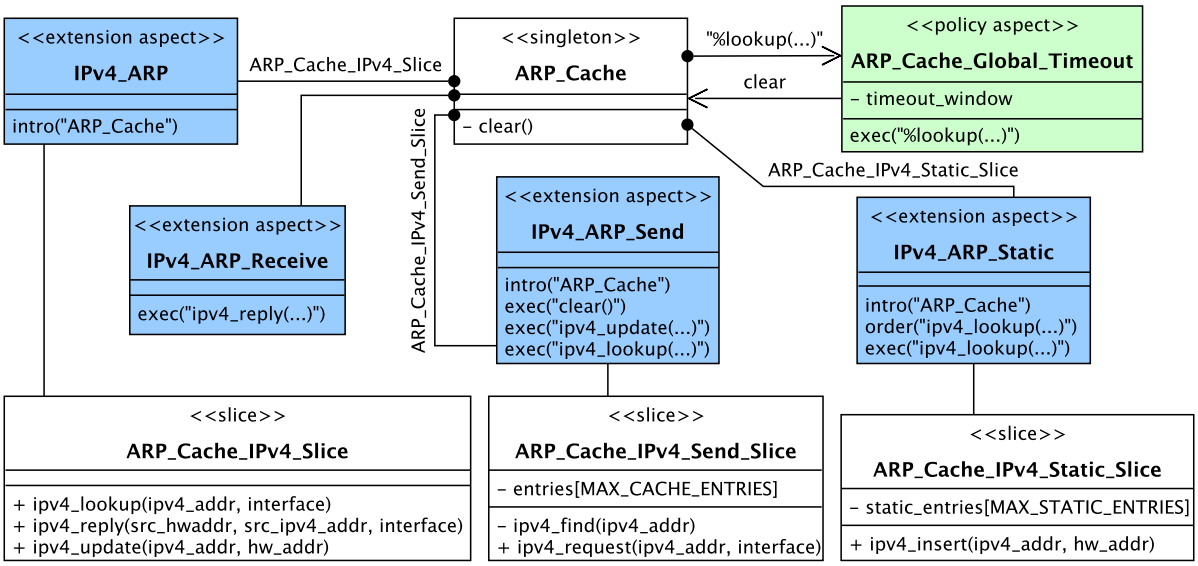

Abbildung 4.11: Entwurf des ARP-Caches

minimalen Erweiterungen entspricht. Die Unterstützung für ARP zum Auflösen von IPv4-Adressen wird durch den Erweiterungsaspekt `IPv4_ARP` bereitgestellt. Dieser fügt der Klasse `ARP_Cache` weitere leere Methoden hinzu, die wiederum explizite Join-Points darstellen. Unabhängig von dem Vorhandensein weiterer Merkmale wird eine einheitliche Schnittstelle dieses Caches für IPv4-Unterstützung erzeugt. Die Möglichkeit auf ARP-Requests, die von anderen Systemen generiert werden, mit ARP-Replies zu antworten wird durch den Aspekt `IPv4_ARP_Receive` realisiert. Soll dieser TCP/IP-Stack IPv4-Pakete über Ethernet empfangen, wird dieses Merkmal benötigt. Das Verwalten der IPv4- und Ethernet-Adressen wird von dem Erweiterungsaspekt `IPv4_ARP_Send` übernommen. Dieser fügt die o. g. Tabelle als Klassenfragment hinzu, in der die dynamische Abbildung von IPv4- auf Ethernet-Adressen abgespeichert wird. Die Methode `ipv4_lookup(ipv4_addr, interface)` wird darüber hinaus durch diesen Aspekt erweitert, so dass eine Suche in dieser Tabelle stattfindet. Damit die gespeicherte Abbildung wieder gelöscht werden kann, wird die Methode `clear()` der Klasse `ARP_Cache` um die dazu nötigen Instruktionen erweitert. Weiterhin ermöglicht dieser Erweiterungsaspekt das Versenden von ARP-Requests durch das beschriebene Klassenfragment. Das Merkmal der statischen Einträge in dem Cache wird erneut durch einen optionalen Erweiterungsaspekt hinzugefügt. Dieser verwendet ein eigenes Klassenfragment, in dem eine Tabelle exklusiv für eine statische Abbildung von IPv4 auf Ethernet-Adressen vorhanden ist, welche über die Methode `ipv4_insert(ipv4_addr, hw_addr)` befüllt werden kann. Durch einen *Order-Advice* wird die Methode `ipv4_lookup(...)` ergänzt, um zuerst nach statischen Einträgen zu suchen und erst daran anschließend einen möglicherweise vorhandenen dynamischen Cache zu benutzen. Das Leeren des dynamischen Caches in periodischen Zeitabständen wird durch einen Policy-Aspekt durchgeführt. Dieser prüft bei dem Aufruf einer Methode, deren Signatur das Wort `lookup` enthält, ob der Cache noch aktuell ist und ruft andernfalls den expliziten Join-Point `clear()` auf, um eine Löschung des dyna-

mischen Caches zu veranlassen. Damit kann auf eine sich ändernde Netzwerktopologie reagiert werden, in der z. B. IP-Adressen dynamisch vergeben werden. Durch den gerade beschriebenen aspektorientierten Entwurf ist die Unterstützung für Ethernet und ARP extrem flexibel und genau auf die Bedürfnisse eines konkreten Anwendungsfalls maßschneiderbar. Des Weiteren ist die Verwendung von ARP nicht auf IPv4 beschränkt, sondern eine Erweiterung für andere Protokolle ist bereits vorgesehen, um eine größtmögliche Wiederverwendbarkeit der entworfenen Komponenten zu erreichen.

4.7 Bottom-Up

Nachdem in Abschnitt 4.6 die Schichtenarchitektur gemäß dem Prinzip Top-Down beschrieben wurde, wird nachfolgend die konträre Vorgehensweise Bottom-Up erläutert, indem u. a. auf die Klassifizierung von Paketen (siehe 4.7.1) und die Registrierung von Verbindungen (siehe 4.7.2) eingegangen wird. Der Empfang von Datenpaketen beginnt auf der Netzzugangsschicht. Dort signalisiert ein Netzwerkgerät den vollständigen Empfang eines neuen Paketes durch eine Hardware-Unterbrechung, die eine entsprechende Unterbrechungsbehandlung auslöst. Diese Unterbrechungsbehandlung ist abhängig von den eingesetzten Netzwerkgeräten und benötigt dementsprechend verschiedene Implementierungen. Durch die Klasse `hw::hal::NetworkDevice` wird eine einheitliche Schnittstelle für den Empfang von Datenpaketen bereitgestellt, indem die Methode `demux(data, datasize)` durch einen Upcall-Aspekt von der Unterbrechungsbehandlung aufgerufen wird (siehe 4.4). Aufgrund dessen wird der Übergang von der ersten zur zweiten Schicht der funktionalen Hierarchie (siehe 4.2) erreicht. An der Signatur dieser Methode ist zu erkennen, dass keine Informationen über die Art der empfangenen Pakete, d. h. welche Protokolle benutzt werden, vorhanden sind. Eine Aufschlüsselung der Pakete anhand der Protokolle wird erst auf der vierten Schicht der funktionalen Hierarchie durch die Funktion Demux vorgenommen. Diese Funktion wird lediglich einmal in dem gesamten System benötigt und kann somit als Klasse gemäß dem Singleton-Entwurfsmuster entworfen werden. Um eine größtmögliche Wiederverwendbarkeit dieser Klasse zu erreichen, muss ihre Schnittstelle unabhängig von den konkret benutzten Protokollen sein. Aus diesem Grund enthält die Klasse `Demux` lediglich eine generische Methode `demux(data, len, interface)`, die anfangs eine leere Implementierung besitzt und somit einen expliziten Join-Point darstellt. Der Übergang von der zweiten Schicht zu dieser Klasse auf der vierten Schicht der funktionalen Hierarchie wird ebenfalls durch einen Upcall-Aspekt erreicht, der den Kontrollfluss aus der Methode `demux(...)` der Klasse `hw::hal::NetworkDevice` zu der gleichnamigen Methode der Klasse `Demux` transferiert. Das für den Empfang des Datenpaketes verantwortliche Netzwerkgerät wird dabei auf das zugehörige Objekt der Klasse `Interface` abstrahiert und der Klasse `Demux` als dritten Parameter der o. g. Methode übergeben. In Abbildung 4.12 ist dieser Vorgang anschaulich dargestellt. Durch die Upcall-Aspekte wird eine lose Kopplung der beteiligten Komponenten erzeugt, denn ohne diese Aspekte gäbe es keine Beziehungen zwischen den Klassen `E1000`, `NetworkDevice` und `Demux`. Damit wird zum einen die funktionale Hierarchie nicht verletzt und zum anderen bietet dieser Entwurf eine einfache Integration von unterschiedlichen Netzwerkge-

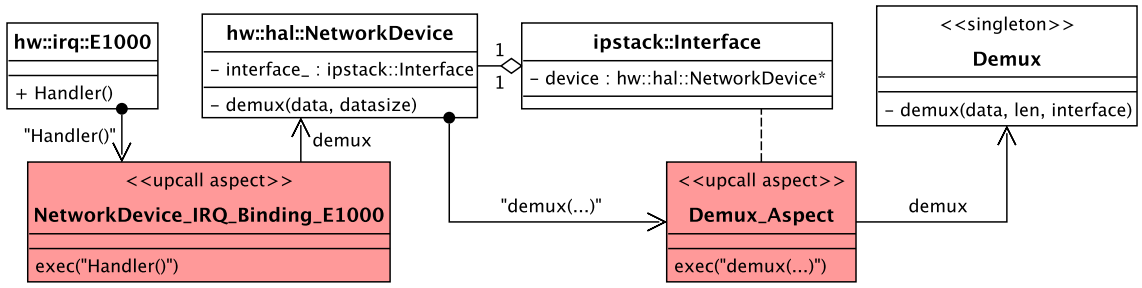

Abbildung 4.12: Upcall-Aspekte beim Paketempfang

räten. Lediglich ein gerätespezifischer Upcall-Aspekt ist notwendig, um ein empfangenes Datenpaket dem TCP/IP-Stack zu übergeben, ohne dass Modifikationen an dem Stack oder an einem Gerätetreiber erforderlich sind.

4.7.1 Klassifizierung von Paketen

Die Analyse eines empfangenen Datenpaketes wird von der Klasse `Demux` durchgeführt. Diese Klasse erhält durch den Upcall-Aspekt, der im vorherigen Abschnitt beschrieben wurde, alle von einem Netzwerkgerät empfangenen Pakete. Das Datenpaket muss zunächst überprüft werden, ob es einem bestimmten Protokollformat genügt und kann daran anschließend entsprechend weiterverarbeitet werden. Konzeptionell entspricht dieses Vorgehen einer Fallunterscheidung. So kann z. B. anhand des Typ-Feldes eines Ethernet-Headers entschieden werden, ob ein IPv4-, IPv6- oder ARP-Paket vorliegt. Ein IP-Paket enthält wiederum ein Protokoll-Feld, das zur Unterscheidung des verwendeten Transportprotokolls benutzt wird. Daran kann erkannt werden, ob es sich um ein UDP- oder TCP-Paket handelt.
Klassisch wird dieses Verhalten durch eine einfache `if`-Anweisung erreicht. Ein Beispiel ist in Quelltext 4.4 zu sehen, der die Fallunterscheidung zwischen TCP- und UDP-Paketen beim Empfang innerhalb des uIP-Stacks (siehe 3.2) zeigt. Die Unterstützung für UDP ist in diesem Stack optional und wird aus diesem Grund durch den C-Präprozessor mittels

```
  if(BUF->proto == UIP_PROTO_TCP) { /* Check for TCP packet. If so,
950                                    proceed with TCP input
                                       processing. */
    goto tcp_input;
  }

955 #if UIP_UDP
  if(BUF->proto == UIP_PROTO_UDP) {
    goto udp_input;
  }
#endif /* UIP_UDP */
```

Quelltext 4.4: Klassiche Protokollunterscheidung in uip.c aus uIP 1.0

`#if` und `#endif` konfigurierbar gemacht, während das Empfangen von TCP-Paketen obligatorisch ist. Diese Vorgehensweise birgt einige Nachteile. Zum einen ist die Wartbarkeit nicht optimal, denn die Erweiterung um zusätzliche Protokolle erfordert eine direkte Modifikation innerhalb des oben gezeigten Quelltextes. Die jeweilige Protokollunterstützung ist direkt in der Empfangsfunktion des uIP-Stacks verankert. Somit kommt es zu einer fehlenden Trennung der Belange, die eine Ursache für die unzureichende Wartbarkeit ist. Zum anderen eignet sich dieser Entwurf nicht zur Umsetzung der funktionalen Hierarchie, die in Abschnitt 4.2 beschrieben ist. Die Netzzugangsschicht, auf der der Datenempfang beginnt, benutzt Informationen aus höheren Schichten, um den Aufbau eines empfangenen Paketes zu analysieren. Dadurch entsteht eine zyklische Abhängigkeit zwischen den Schichten und Konfigurierbarkeit kann lediglich durch externe Werkzeuge wie dem Präprozessor erreicht werden.
Die genannten Probleme lassen sich durch einen aspektorientierten Entwurf vermeiden. Dazu wird zuerst ein neuer Aspekttyp eingeführt, der ein neues aspektorientiertes Entwurfsprinzip ermöglicht.

Upcall-Dispatcher-Aspekte

Ein *Upcall-Dispatcher-Aspekt* (engl. *„Upcall Dispatcher Aspect“*) kann als eine gezielte Kombination von einen Upcall- und einem Erweiterungsaspekt (siehe 2.5.1) verstanden werden. Ähnlich zum Erweiterungsaspekt wird eine bestehende Klasse um ein Klassenfragment erweitert und wie beim Upcall-Aspekt wird ein Ereignis auf einer höheren Schicht ausgelöst. Der Verwendungszweck eines Upcall-Dispatcher-Aspektes ist stets durch ein Datenobjekt begründet, welches die Ursache für alle auftretenden Ereignisse darstellt. In Abbildung 4.13 wird dieser Zusammenhang veranschaulicht. Ein expliziter Join-Point auf einer Schicht N wird von einem Upcall-Dispatcher-Aspekt einer höheren

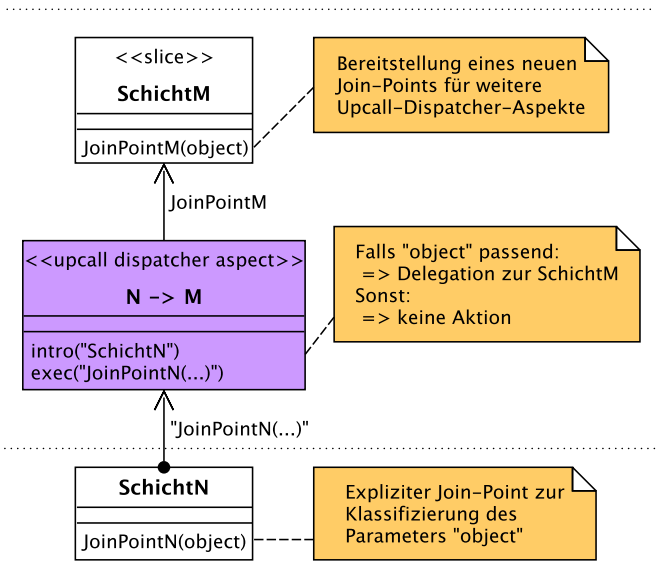

Abbildung 4.13: Upcall-Dispatcher-Aspekt

Schicht *M* beeinflusst. In diesem Aspekt wird das betreffende Datenobjekt daraufhin untersucht, ob es in der Schicht *M* weiterverarbeitet werden soll. Trifft diese Bedingung zu, wird das Datenobjekt zwecks Weiterverarbeitung erneut zu einem expliziten Join-Point delegiert, der sich nun auf der Schicht *M* befindet. Dieser Join-Point wird ebenfalls von dem Upcall-Dispatcher-Aspekt bereitgestellt, indem ein entsprechendes Klassenfragment benutzt wird. Eine mögliche Implementierung dieses Aspektes ist in Quelltext 4.5 gezeigt. Dieser Aspekttyp zeichnet sich durch eine flexible Verwendung eines *Around Advices* aus. Damit wird der Kontrollfluss in den Upcall-Dispatcher-Aspekt transferiert und dort wird mit Hilfe des Datenobjektes eine Bedingung überprüft. Je nach Wahrheitswert dieser Bedingung kehrt der Kontrollfluss entweder zum Ausgangspunkt zurück oder wird in eine höhere Schicht geleitet. Durch diesen Vorgang ist es möglich, dass mehrere Upcall-Dispatcher-Aspekte den gleichen Join-Point beeinflussen, und ausschließlich derjenige Aspekt aktiv wird, dessen Bedingung bzgl. des Datenobjektes erfüllt ist. Diese Aspekte können problemlos weggelassen werden, so dass sich eine dezentrale und damit konfigurierbare Fallunterscheidung entwerfen lässt.

```
aspect Upcall_Dispatcher_Aspect {
  advice "SchichtN" : slice SchichtM; // class fragment

  advice execution("% ...::JoinPointN(Object*)") &&
         args(object) && that(schicht) :
         around(Object* object, SchichtN& schicht){

    if(condition(object)){ // does 'object' match?
      schicht.JoinPointM(object); // delegation (upcall)
    }
    else{
      tjp->proceed(); // no action: return to SchichtN
    }
  }
};
```

Quelltext 4.5: Implementierung eines Upcall-Dispatcher-Aspektes

Der grundlegende Unterschied dieses Aspekttyps gegenüber den bisher in Abschnitt 2.5.1 beschriebenen Aspekttypen ist, dass eine Beeinflussung von Aspekten durch andere Aspekte bereits eingeplant wird. Die in dem Klassenfragment definierten Methoden schaffen eine Schnittstelle, die durch andere Aspekte benutzt werden kann. Eine „Stapelung“ von Upcall-Dispatcher-Aspekten ist somit problemlos möglich, indem ein Aspekt der höheren Schicht die Methoden des Klassenfragmentes von dem Aspekt der niedrigeren Schicht beeinflusst. Diese Vorgehensweise führt zu einem neuen aspektorientierten Entwurfsprinzip, das im Folgenden erläutert wird.

Upcall-Dispatcher-Hierarchie

Eine mehrfache und aufeinander aufbauende Verwendung von Upcall-Dispatcher-Aspekten wird im Weiteren als *Upcall-Dispatcher-Hierarchie* (engl. *„Upcall Dispatcher Hierarchy“*)

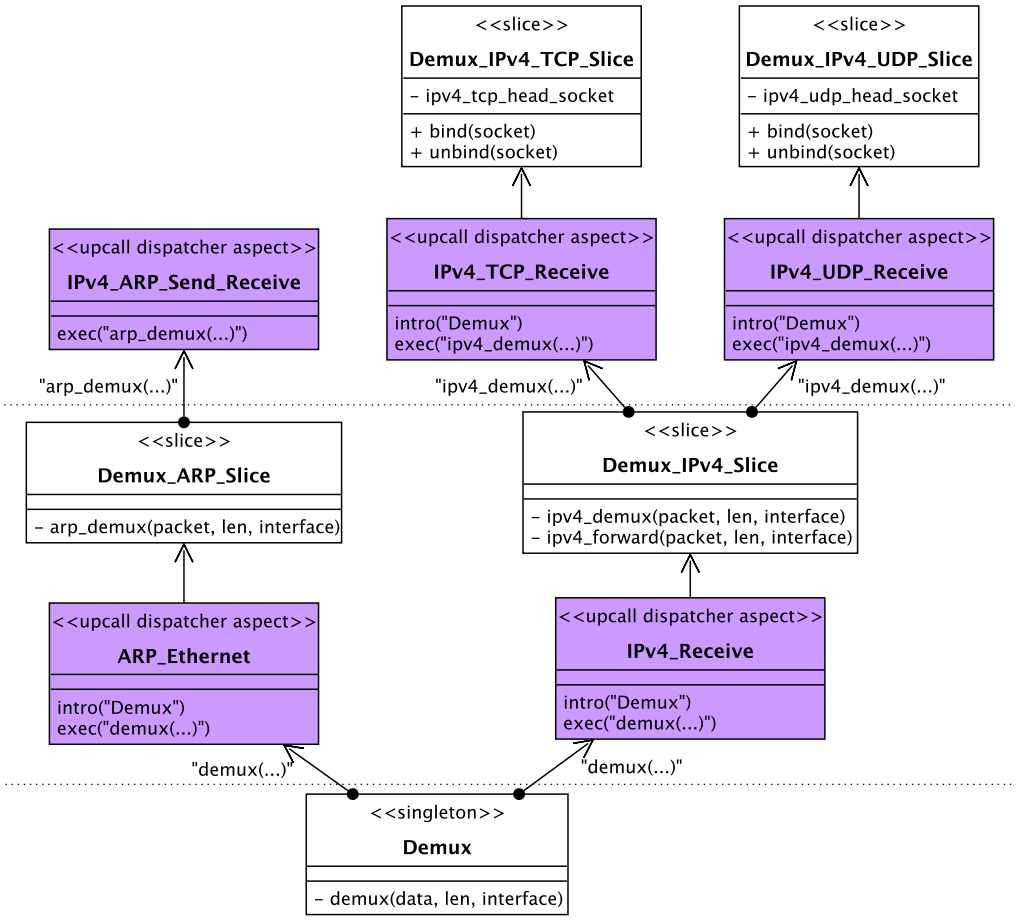

Abbildung 4.14: Upcall-Dispatcher-Hierarchie

bezeichnet. Eingesetzt werden kann dieses Entwurfsprinzip überall dort, wo eine konfigurierbare Fallunterscheidung in Verbindung mit Upcalls benötigt wird. Erläutert wird dieses Entwurfsprinzip anhand der Anwendung in dem TCP/IP-Stack dieser Diplomarbeit, welche die Klassifizierung von Datenpaketen bzgl. der verwendeten Protokolle betrifft. Abbildung 4.14 zeigt ein Klassendiagramm, das nach diesem Prinzip aufgebaut ist. Ausgangspunkt der Upcall-Dispatcher-Hierarchie ist dabei die Klasse `Demux`. Der explizite Join-Point dieser Klasse wird von zwei Upcall-Dispatcher-Aspekten verwendet. Auf der linken Seite befindet sich der Aspekt `ARP_Ethernet`, mit dem eintreffende Pakete darauf überprüft werden, ob sie über ein Ethernet-Gerät empfangen wurden und ob es sich dabei um ein ARP-Paket handelt. Treffen diese beiden Bedingungen zu, wird zuerst der Ethernet-Header entfernt, die Paketlänge überprüft und angepasst und das Paket danach der Methode `arp_demux(packet, len, interface)` übergeben. Diese Methode wird durch einen weiteren Upcall-Dispatcher-Aspekt beeinflusst, der ARP-Pakete, die eine IPv4-Adresse enthalten, verarbeitet. Handelt es sich um genau so ein ARP-Paket, wird es von dem Aspekt `ARP_IPv4_Send_Receive` auf Korrektheit untersucht und anschließend werden die nötigen Informationen der Klasse `ARP_Cache` (siehe 4.6.3) übergeben. Die rechte Seite der Abbildung enthält den Aspekt `IPv4_Receive`, der alle empfange-

nen Pakete überprüft, ob sie einen IPv4-Header enthalten. Falls das zutrifft, wird die Prüfsumme und die Länge des Headers verifiziert und die IPv4-Adresse des Paketziels ermittelt. Entspricht diese IPv4-Adresse nicht der des Netzwerkgerätes, über welches das Paket empfangen wurde, wird das IPv4-Paket der Methode `ipv4_forward(packet, len, interface)` übergeben, um anschließend zum eigentlichen Ziel weitergeleitet zu werden. Stimmt die Zieladresse mit der des Netzwerkgerätes überein, wird die Methode `ipv4_demux(packet, len, interface)` mit dem IPv4-Paket aufgerufen, um eine lokale Weiterverarbeitung dieses Paketes auszulösen. Diese Weiterverarbeitung wird durch die Aspekte `IPv4_TCP_Receive` und `IPv4_UDP_Receive` übernommen, die auf dem Aspekt `IPv4_Receive` aufbauen, indem sie die Methode `ipv4_demux(packet, len, interface)` beeinflussen. Alle an dieser Methode eintreffenden Pakete werden auf das Vorhandensein eines TCP- oder UDP-Headers untersucht und anschließend durch den entsprechenden Aspekt verarbeitet, in dem u. a. die TCP- bzw. UDP-Prüfsumme verifiziert wird. Das Klassenfragment `Demux_IPv4_TCP_Slice` enthält einen Zeiger auf den Anfang einer verketteten Liste von `IPv4_TCP_Socket`-Objekten (siehe 4.6), die für den Empfang von Datenpaketen registriert sind. Für ein empfangenes TCP-Paket wird diese Liste durchlaufen und das passende Objekt gesucht, das in IPv4-Adressen und TCP-Portnummern mit dem Paket übereinstimmt. Wenn ein solches Objekt gefunden werden kann, wird eine Kopie des Paketes in dem Ringpuffer des Objektes abgespeichert. Andernfalls muss das Paket verworfen werden. Die Verarbeitung von UDP-Paketen findet auf die gleiche Weise statt.
Der gerade beschriebene Entwurf bietet den Vorteil, dass jeder Upcall-Dispatcher-Aspekt problemlos weggelassen werden kann. Wird beispielsweise in einem bestimmten Anwendungsfall das Protokoll TCP nicht benötigt, kann der Aspekt `IPv4_TCP_Receive` einfach ausgelassen werden und es findet automatisch keine Verarbeitung von empfangenen TCP-Paketen statt. Eine Upcall-Dispatcher-Hierarchie ist außerdem flexibel erweiterbar. So könnte z. B. ein Aspekt `IPv6_Receive` hinzugefügt werden, um IPv6-Pakete zu empfangen. Konzeptionell entspricht dieser Entwurf exakt der funktionalen Hierarchie aus Abschnitt 4.2, da zwischen den Upcall-Dispatcher-Aspekten die gleichen Relationen wie zwischen den Funktionen der funktionalen Hierarchie gelten. Obwohl der Kontrollfluss beim Paketempfang entgegen dieser Hierarchie verläuft, bleibt eine Trennung zwischen den Schichten trotzdem erhalten.
Die Effizienz dieses Entwurfsprinzips hängt von Compileroptimierungen ab. Der Compiler AspectC++ veranlasst ein *Inlining* von allen expliziten Join-Points und deren Aufrufen in der Upcall-Dispatcher-Hierarchie. Dadurch unterscheidet sich die Effizienz dieses Entwurfs nicht von der einfachen Fallunterscheidung aus Quelltext 4.4. Der generierte Maschinencode ist sogar identisch zu einer solchen Fallunterscheidung.

4.7.2 Registrierung von Verbindungen

Im vorangehenden Abschnitt wurde darauf hingewiesen, dass eine Registrierung von Objekten der Klassen `IPv4_TCP_Socket` und `IPv4_UDP_Socket` an der Klasse `Demux` stattfindet, bevor der eigentliche Datenempfang möglich ist. Durch diese Registrierung wird für jedes Protokoll eine verkettete Liste von Empfangsobjekten aufgebaut, die bei

dem Eintreffen eines zugehörigen Paketes durchsucht wird. Die Klasse `IPv4_TCP_Socket` umfasst dazu den expliziten Join-Point `bind()`, der in dieser Klasse eine leere Methode darstellt. Im Zusammenhang mit einem TCP-Verbindungsaufbau wird diese Methode aufgerufen, um den Empfang von Paketen einzuleiten. Durch den Upcall-Dispatcher-Aspekt `IPv4_TCP_Receive` wird dieser Methodenaufruf an eine gleichnamige Methode des Klassenfragmentes `Demux_IPv4_TCP_Slice` delegiert, in welchem sie schließlich implementiert wird (siehe Abbildung 4.14). Im Wesentlichen wird dabei das aufrufende Objekt in die o. g. verkettete Liste aufgenommen. Wird die Verbindung nicht mehr benötigt, kann mit dem expliziten Join-Point `unbind()`, der ebenfalls zu dem Klassenfragment `Demux_IPv4_TCP_Slice` delegiert wird, das aufrufende `IPv4_TCP_Socket`-Objekt aus der verketteten Liste wieder entfernt werden.
Die Vorgehensweise für die Klasse `IPv4_UDP_Socket` ist identisch. Da das Empfangen von UDP-Paketen optional ist, kann durch Weglassen des Upcall-Dispatcher-Aspektes `IPv4_UDP_Receive` sämtliche Empfangsfunktionalität einschließlich der Verwaltung der verketteten Liste entfernt werden.
Die Verwaltung von passiven TCP-Verbindungen, die für einen TCP-Server gebraucht werden, erfordert eine andere Vorgehensweise als aktive TCP-Verbindungen für TCP-Clients. Ein Server definiert in der Regel lediglich einen Port und nimmt alle Pakete auf diesem Port entgegen, die nicht zu einer bereits bestehenden Verbindung gehören. Ein Client hingegen definiert sowohl eine Ziel- als auch Absender-IPv4-Adresse und einen lokalen sowie entfernten Port. Daraus resultiert eine unterschiedliche Behandlung von eintreffenden TCP-Paketen, je nachdem ob sie für einen Server oder Client bestimmt sind. Aus diesem Grund wird die Unterstützung für TCP-Server durch einen separaten Upcall-Dispatcher-Aspekt `IPv4_TCP_Listen` realisiert. Dieser Aspekt enthält eine eigene verkettete Liste von `IPv4_TCP_Socket`-Objekten, die sich in dem Zustand *Listen* befinden und damit auf einen externen Verbindungsaufbau warten (siehe 3.3.2). Wird ein solcher Verbindungsaufbau durch ein eintreffendes TCP-Paket mit dem TCP-Flag `SYN` ausgelöst, so wird dieses Paket dem entsprechenden `IPv4_TCP_Socket`-Objekt übergeben. Aufgrund dessen können die Ziel- und Absender-IPv4-Adresse sowie der entfernte Port dieses Objektes festgelegt werden und somit der *Listen-Zustand* verlassen werden, wodurch eine Registrierung als vollständige Verbindung mit `bind()` möglich wird. Nach einem Verbindungsende durch ein Paket mit dem TCP-Flag `FIN` findet erneut ein Übergang in den Zustand *Listen* statt, d. h. die Verbindung wird mit `unbind()` beendet und das zugehörige Objekt in die verkettete Liste des Aspektes `IPv4_TCP_Listen` aufgenommen. Dieser Vorgang ist schematisch in Abbildung 4.15 gezeigt. Die An- und Abmeldung an der

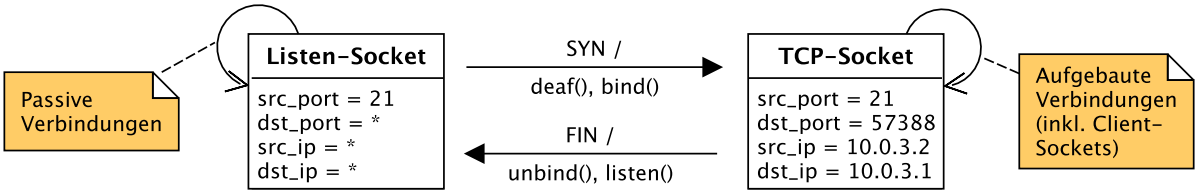

Abbildung 4.15: Listen-Socket

Liste der passiven Verbindungen erfolgt durch zwei explizite Join-Points mit den Signaturen `listen()` und `deaf()`, die in einem Klassenfragment des Upcall-Dispatcher-Aspektes `IPv4_TCP_Listen` vorhanden sind. Ein solcher *„Listen-Socket“* kann stets genau einen Verbindungsaufbau annehmen und diesen verarbeiten. Erst nach dem Verbindungsende ist es möglich, einen weiteren Verbindungsaufbau entgegenzunehmen. Ein paralleler TCP-Server, der mehrere Verbindungen gleichzeitig bedient, benötigt dementsprechend mehrere dieser *„Listen-Sockets“*, die alle den gleichen Port benutzen. So kann statisch festgelegt werden, wie viele Verbindungen gleichzeitig bearbeitet werden können, ohne dass eine dynamische Prozesserzeugung benötigt wird. Diese Vorgehensweise stellt keine Einschränkung dar und findet ebenfalls in anderen Systemen Verwendung, wie z. B. beim *Prefork* des *Apache-Webservers* [54, S. 102ff].

5 Domänenimplementierung

Die Software-Produktlinienentwicklung umfasst neben der Domänenanalyse (Kapitel 3) und dem Domänenentwurf (Kapitel 4) die Implementierung von wiederverwendbaren Softwarekomponenten, die Gegenstand dieses Kapitels ist. Im Folgenden wird auf einige charakteristische Gesichtspunkte der Domänenimplementierung eingegangen, um einen Überblick über die entwickelten Softwarekomponenten zu vermitteln. Im Vordergrund steht die Anwendung der Aspektorientierung, die ein zentraler Bestandteil der Implementierung des in dieser Diplomarbeit entworfenen TCP/IP-Stacks darstellt. Zu Beginn findet in Abschnitt 5.1 eine Diskussion über die effiziente Umsetzung des Strategie-Entwurfsmusters statt, welches in dem o. g. Domänenentwurf verwendet wird. Daraufhin werden einige wichtige Details der Implementierung von Netzwerkprotokollen vorgestellt. Abschnitt 5.3 befasst sich mit der konfigurierbaren Programmierschnittstelle des TCP/IP-Stacks, die von der Anwendungsschicht genutzt werden kann. Abschließend wird die Integration des TCP/IP-Stacks in die CiAO-Betriebssystemfamilie behandelt (siehe 5.4).

5.1 Kostenneutrales Strategie-Entwurfsmuster

Der Entwurf der Netzzugangsschicht aus Abschnitt 4.4 basiert auf der Verwendung von virtuellen Methoden zur Realisierung einer einheitlichen Schnittstelle für unterschiedliche Netzwerkgeräte. Diese Vorgehensweise entspricht dem objektorientierten Entwurfsmuster der *Strategie* [17, S. 373]. Wird genau ein Netzwerkgerät verwendet, stellt dieses Entwurfsmuster keine effiziente Lösung mehr dar, denn die virtuellen Methoden verursachen vermeidbare Speicher- und Laufzeitkosten. Mit Hilfe von AOP lässt sich dieses Entwurfsmuster dennoch effizient implementieren. Die Klasse `hw::hal::NetworkDevice`, welche die einheitliche Schnittstelle bereitstellen soll, wird als leere Klasse ohne Methoden implementiert, von der die konkreten Netzwerkgeräte (hier `hw::dev::E1000`) weiterhin erben. Sollen gleichzeitig unterschiedliche Netzwerkgeräte verwendet werden, so werden virtuelle Methoden durch ein Klassenfragment in die Basisklasse eingefügt. Findet diese Erweiterung der Basisklasse statt, ist das Resultat äquivalent zu dem ursprünglichen Entwurf aus Abschnitt 4.4. In Abbildung 5.1 ist diese Erweiterung auf der rechten Seite gezeigt. Die Initialisierung der virtuellen Funktionstabelle (vtable) kann nachträglich durch *Placement new* erfolgen, indem der Aspekt `NetworkDevice_Multi` die Konstruktoren der Gerätetreiberklassen explizit aufruft, sofern das nicht automatisch von einem *Startupcode* erledigt wird. Wird hingegen genau ein Netzwerkgerät benötigt, können nicht-virtuelle Methoden mit leerer Implementierung der Basisklasse `hw::hal::NetworkDevice` durch ein anderes Klassenfragment hinzugefügt werden. Diese leeren Methoden dienen als

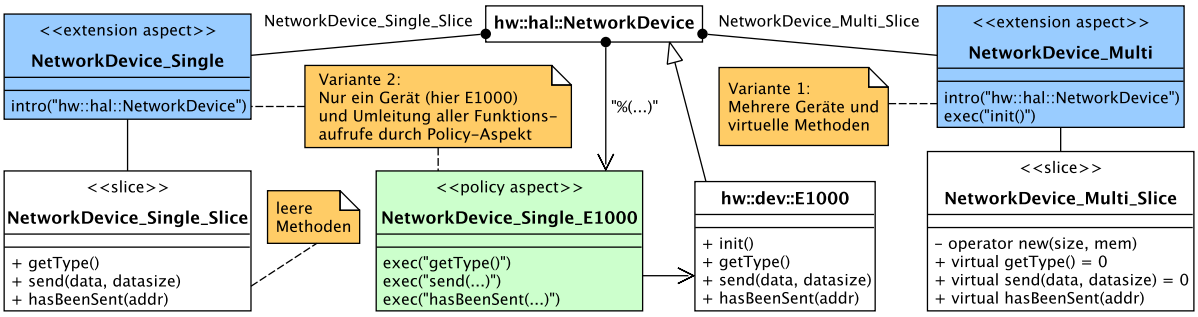

Abbildung 5.1: Kostenneutrale Implementierung der Netzzugangsschicht

explizite Join-Points für einen gerätespezifischen Policy-Aspekt, der alle Methodenaufrufe dieser Join-Points zu einer konkreten Gerätetreiberklasse umleitet. Die linke Seite der Abbildung 5.1 veranschaulicht das gerade beschriebene Prinzip. Damit wird das gleiche Verhalten wie mit virtuellen Methoden realisiert, wenn nur ein Gerätetyp verwendet wird. Durch *Inlining* der expliziten Join-Points verursacht diese Implementierung keine Speicher- und Laufzeitkosten.

5.2 Protokollimplementierung

Die Implementierung von Netzwerkprotokollen ist ein wesentlicher Bestandteil der wiederverwendbaren Softwarekomponenten der in dieser Diplomarbeit entwickelten Produktlinie. Abschnitt 5.2.1 beschreibt die Byte Order der Protokoll-Header, von der die Implementierungen aller Protokolle eines TCP/IP-Stacks betroffen sind. Im Anschluss an diesen querschneidenden Belang wird kurz darauf eingegangen, wie das Empfangen von fragmentierten IP-Paketen realisiert werden kann, bevor in Abschnitt 5.2.3 die Implementierung der Prüfsummenberechnung erläutert wird. Die Details der TCP-Implementierung werden daraufhin exemplarisch betrachtet (siehe 5.2.4).

5.2.1 Network Byte Order

„To keep the Internet Protocol independent of the machines on which it runs, the protocol standard specifies *network byte ordering*[1] for all integer quantities in the header“ [44, S. 73]. Die Network Byte Order ist als *Big-Endian* definiert. Enthält ein Protokoll-Header ein Feld, das größer als 8 Bit ist, so ist dieses als Big-Endian zu interpretieren. Das ist auf Maschinen unproblematisch, die ebenfalls diese Byte Order benutzen. Anders verhält es sich auf Architekturen wie *x86*, die *Little-Endian* einsetzen. Jeder Zugriff auf ein solches Feld eines Protokoll-Headers erfordert eine Konvertierung der Daten bzgl. der Byte Order. Davon sind alle Protokolle eines TCP/IP-Stacks betroffen, d. h. in dieser Diplomarbeit Ethernet, ARP, IPv4, UDP und TCP. Folglich stellt die Konvertierung der Byte Order einen querschneidenden Belang dar, der fast jede Komponente des TCP/IP-Stacks

[1] Hervorhebungen im Original

beeinflusst. Klassisch erfolgt der Zugriff auf die Protokoll-Header durch Präprozessor-Makros, die je nach Byte Order der verwendeten Maschine eine Konvertierung durchführen oder als leere Makros definiert sind. Eine solche Realisierung bietet zwar auf der einen Seite Konfigurierbarkeit, auf der anderen Seite muss bei jedem Zugriff auf einen Protokoll-Header die korrekte Benutzung der Makros beachtet werden. Das führt zu den in Abschnitt 2.4 beschriebenen Problemen bei der Nutzung des C-Präprozessors.
In dieser Diplomarbeit wird ein anderer Ansatz implementiert. Ein Datenpaket inklusive Protokoll-Header wird als Klasse umgesetzt und der Zugriff auf sämtliche Attribute der Klasse erfolgt über Zugriffsmethoden. Die Methoden zum Auslesen von Attributen beginnen mit dem Präfix `get` und die Methoden zum Schreiben mit dem Präfix `set`. Diese Namenskonvention entspricht der üblichen Vorgehensweise eines objektorientierten Entwurfs. Die Zugriffsmethoden bieten die nötigen Join-Points, um die Konvertierung der Byte Order in einem einzigen Aspekt zu implementieren. In Quelltext 5.1 ist ein Auszug eines solchen Aspektes gezeigt. Zu einer besseren Übersicht ist nur die Konvertierung von Zahlen der Länge 16 Bit aus den Headern der Protokolle IPv4, UDP und TCP dargestellt. Die Implementierung für weitere Protokolle und längere Zahlen unterscheidet sich davon kaum. Somit erfolgt die Konvertierung der Byte Order für die restlichen Komponenten des TCP/IP-Stacks völlig transparent, indem die o. g. Zugriffsmethoden

```
aspect Little_Endian {
  UInt16 htons(UInt16 n){
    return ((n & 0xff) << 8) | ((n & 0xff00) >> 8);
  }
  UInt16 ntohs(UInt16 n){ return htons(n); }

  pointcut Network_to_Host_Short() =
    "% ipstack::IPv4_Packet::get%()" ||
    "% ipstack::UDP_Packet::get%()"  ||
    "% ipstack::TCP_Segment::get%()";

  advice execution(Network_to_Host_Short()) && result(res) :
         after(UInt16 res){
    *tjp->result() = ntohs(res); // convert return value
  }

  pointcut Host_to_Network_Short() =
    "void ipstack::IPv4_Packet::set%(%)" ||
    "void ipstack::UDP_Packet::set%(%)"  ||
    "void ipstack::TCP_Segment::set%(%)";

  advice execution(Host_to_Network_Short()) && args(val) :
         before(UInt16 val){
    *((UInt16*)tjp->arg(0)) = htons(val); // convert 1st argument
  }
};
```

Quelltext 5.1: Konvertierung der Byte Order für Zahlen der Länge 16 Bit

benutzt werden, die implizit die Konvertierung durchführen. Auf einer Big-Endian Maschine ist selbstverständlich keine Konvertierung nötig, so dass auch kein Aspekt dafür erforderlich ist. Durch AOP kann dieser querschneidende Belang als eigenständige Komponente implementiert werden.

5.2.2 Fragment Reassembly

Das Empfangen von fragmentierten IPv4-Paketen wurde in der Domänenanalyse (siehe 3.3.3) als optionales Merkmal identifiziert. Das Zusammensetzen von einzelnen Fragmenten zu einem vollständigen IPv4-Paket, das wegen einer zu geringen MTU der verwendeten Netzwerkhardware nicht zusammenhängend übertragen werden konnte, wird als *Fragment Reassembly* [30, S. 24ff] bezeichnet. Aufgrund des modularen Entwurfs der Upcall-Dispatcher-Hierarchie (siehe 4.7.1) ist es möglich, dieses Merkmal als separaten Upcall-Dispatcher-Aspekt zu implementieren. Dieser Aspekt enthält einen Puffer für die einzelnen Fragmente eines IPv4-Paketes. Trifft ein IPv4-Paket ein, das als Fragment gekennzeichnet ist, wird es in diesem Puffer abgespeichert, bis alle Fragmente des kompletten IPv4-Paketes vorhanden sind. Anschließend wird das zugehörige IPv4-Paket rekonstruiert. Ist der Vorgang erfolgreich, kann es danach dem expliziten Join-Point `ipv4_demux(packet, len, interface)` (siehe 4.7.1) übergeben werden und von der Upcall-Dispatcher-Hierarchie wie ein herkömmliches IPv4-Paket behandelt werden. Die Implementierung dieses Aspektes umfasst neben dem Puffer für die Fragmente außerdem einen Timeout-Mechanismus. Dieser legt fest, wie groß die maximale Zeitdifferenz zwischen dem Empfang von zwei Fragmenten eines Paketes sein darf. Wird diese Zeit überschritten, so werden alle bisher empfangenen Fragmente gelöscht und der Puffer wird für ein anderes IPv4-Paket freigegeben.

5.2.3 Prüfsummenberechnung

Die Protokolle UDP und TCP verwenden Prüfsummen, um die Integrität von übertragenen Datenpaketen zu gewährleisten. Dabei wird neben den Nutzdaten auch der UDP- bzw. TCP-Header durch die Prüfsumme gesichert. Diese Prüfsummen sind zusätzlich abhängig von dem verwendeten Internetprotokoll, da bei der Berechnung ein *IP-Pseudo-Header* [28, S. 16f] berücksichtigt werden muss. Dieser Pseudo-Header umfasst u. a. die Absender- und Ziel-IP-Adressen. Damit unterscheidet sich die Berechnung der UDP- und TCP-Prüfsummen bei IPv4 von der bei IPv6. Der Entwurf der Referenzarchitektur des TCP/IP-Stacks sieht eine strikte Trennung der Protokolle vor (siehe 4.6). Eine direkte Berechnung der Prüfsummen innerhalb der Klassen `UDP_Socket` und `TCP_Socket` ist somit unmöglich. Darüber hinaus ist der Algorithmus zur Prüfsummenberechnung bei UDP und TCP identisch, so dass es eine gemeinsame Implementierung geben sollte. Die Berechnung der Prüfsummen lässt sich sowohl durch eine allgemeine Implementierung als auch durch spezielle Netzwerkhardware erreichen.

Allgemeine Implementierung

Diese *schichtenübergreifenden* Anforderungen an die Prüfsummenberechnung lassen sich durch Aspekte umsetzen, welche die Klasse `IPv4_Socket` (siehe 4.6) beeinflussen. Ist ein IPv4-Paket vor dem Versenden vollständig erstellt, lassen sich die Absender- und Ziel-IP-Adressen ablesen. Ein Aspekt kann nun damit die Prüfsummen berechnen und in die Header der Protokolle UDP und TCP einfügen. Für diese Berechnung kann eine gemeinsame Funktion benutzt werden, auf welche die Aspekte für UDP- und TCP-Prüfsummen zurückgreifen. Auf diese Weise wird zusätzlich die Konfigurierbarkeit der UDP-Prüfsumme erreicht. Da diese optional ist, kann eine Berechnung davon vermieden werden, indem der entsprechende Aspekt nicht ausgewählt wird. Die Verifikation von empfangenen Paketen gestaltet sich etwas einfacher als die o. g. Berechnung der Prüfsummen beim Senden. Die Upcall-Dispatcher-Aspekte der Upcall-Dispatcher-Hierarchie (siehe 4.7.1) stellen bereits eine Kombination der Protokolle IPv4 und UDP bzw. TCP dar. Der Aspekt `IPv4_TCP_Receive` enthält beispielsweise alle Informationen, die für eine Verifikation notwendig sind.

Berechnung durch Netzwerkhardware

In der Domänenanalyse wurde bereits das Merkmal der hardwarebeschleunigten Prüfsummenberechnung (siehe 3.3.3) vorgestellt. Einige Netzwerkgeräte verfügen über die Funktionen, beim Versenden von UDP- und TCP-Paketen die benötigen Prüfsummen selbstständig zu berechnen und in die Protokoll-Header einzufügen. Weiterhin können empfangene Pakete von diesen Netzwerkgeräten auf eine korrekte Prüfsumme hin untersucht und ggf. verworfen werden. Da der Entwurf der Netzzugangsschicht (siehe 4.4) bereits eine gleichzeitige Verwendung von unterschiedlichen Netzwerkgeräten vorsieht, muss die dort entworfene Schnittstelle erweitert werden, um Netzwerkgeräte mit hardwarebeschleunigter Prüfsummenberechnung identifizieren zu können. Dazu werden die Klassen `hw::hal::NetworkDevice` und `ipstack::Interface` um zwei Methoden mit den Signaturen `hasTransmitterHardwareChecksumming()` und `hasReceiverHardwareChecksumming()` ergänzt. Anhand dieser Methoden kann für jedes Netzwerkgerät überprüft werden, ob beim Senden eine vollständige Prüfsumme durch den TCP/IP-Stack berechnet werden soll und ob empfangene Pakete bereits verifiziert wurden. Die Überprüfung der Netzwerkgeräte ist nur dann sinnvoll, wenn mindestens ein Gerät mit hardwarebeschleunigter Prüfsummenberechnung vorhanden ist. Aus diesem Grund wird diese Überprüfung von optionalen Aspekten durchgeführt, die jeweils eine Prüfsummenberechnung durch die Netzwerkhardware beim Senden bzw. Empfangen realisieren. Somit können unnötige Fallunterscheidungen vermieden werden.

5.2.4 TCP

Die Implementierung des Protokolls TCP in dieser Diplomarbeit ist deutlich umfangreicher als die der anderen Protokolle. Aufgrund des enormen Umfangs der Implementierung

kann in diesem Kapitel lediglich auf einige ausgewählte Details eingegangen werden. Die grundlegende Funktionalität der Klasse `TCP_Socket` wird ohne Aspekte implementiert, da der minimale Funktionsumfang nicht konfigurierbar ist. Dabei wird versucht, möglichst viele Join-Points durch eine objektorientierte Implementierung mit vielen `get-` und `set-`Methoden anzubieten. Eine Erweiterung um optionale Merkmale durch Aspekte ist also in dieser Implementierung bereits eingeplant. Im Folgenden wird die Realisierung der optionalen Merkmale Sliding Window, Silly Window Syndrome Vermeidung, Round-Trip-Time Berechnung und Slow Start diskutiert.

5.2.4.1 Sliding Window

In der Domänenanalyse wurde bereits festgestellt, dass das Merkmal des Sliding Windows (siehe 3.3.3) in ein Sender- und ein Empfängermerkmal unterteilt werden kann. Die Implementierung dieser Merkmale findet also getrennt statt und wird nun kurz erläutert.

Senderseitiges Sliding Window

Wird ein TCP-Paket von diesem TCP/IP-Stack versendet, so wird es nach dem Sendevorgang in einer Klasse `TCP_History` gespeichert. Erst nach dem Eintreffen der Empfangsbestätigung kann der Speicherplatz für das Paket freigegeben werden. Andernfalls findet nach einer gewissen Zeit eine erneute Versendung des abgespeicherten Paketes statt, bis eine Empfangsbestätigung empfangen wird. Die Implementierung der Klasse `TCP_History` umfasst anfangs nur einen einzigen Speicherplatz für versendete TCP-Pakete. Ein neues Paket kann nur dann versendet werden, wenn dieser Speicherplatz frei ist. Das Merkmal des senderseitigen Sliding Windows wird als minimale Erweiterung durch einen optionalen Aspekt implementiert. Die o. g. Klasse wird mit einem Klassenfragment um weitere Speicherplätze ergänzt, so dass mehrere Pakete direkt hintereinander versendet werden können. Das Eintreffen einer Empfangsbestätigung führt in diesem Fall dazu, dass alle belegten Speicherplätze durchsucht und die bestätigten TCP-Pakete freigegeben werden.

Empfängerseitiges Sliding Window

Die Verarbeitung eines empfangenen TCP-Paketes führt in diesem TCP/IP-Stack dazu, dass es in einer Klasse `TCP_ReceiveBuffer` abgespeichert wird, bis es vollständig von der Anwendungsschicht ausgelesen wird. Diese Klasse enthält im Ausgangszustand nur genau einen Speicherplatz für ein TCP-Paket. Die Flusskontrolle von TCP stellt sicher, dass der Absender erst dann ein neues Paket versendet, wenn der Speicherplatz frei ist. Ebenso wie das senderseitige wird das empfängerseitige Sliding Window als minimale Erweiterung implementiert. Durch ein Klassenfragment wird die Klasse `TCP_ReceiveBuffer` um zusätzliche Speicherplätze erweitert. Des Weiteren wird die Flusskontrolle angepasst. Das Advertised Window (siehe 3.3.3) umfasst nun alle verfügbaren Speicherplätze. Damit ist es möglich, mehrere Pakete direkt hintereinander zu empfangen und mit einer kumulativen Empfangsbestätigung zu quittieren.

5.2.4.2 Silly Window Syndrome

Die Vermeidung des Silly Window Syndroms (SWS) (siehe 3.3.3) ist wesentlich für die Erzielung einer hohen TCP-Datenrate. Eine Vermeidung durch den Empfänger lässt sich als Aspekt in wenigen Zeilen Quelltext (siehe Quelltext 5.2) implementieren. Ausgangspunkt ist die Methode `getReceiveWindow()` der Klasse `TCP_Socket`, die den aktuellen Wert des Advertised Windows berechnet. Dieser Wert wird in der Regel zur Flusskontrolle direkt dem Sender übermittelt, der dann eine entsprechende Menge von Daten versenden darf. Um das SWS auszuschließen, kann durch den gezeigten Aspekt der o. g. Wert künstlich reduziert werden. Falls kein Paket maximaler Größe (MSS) aufgrund eines bereits gefüllten Empfangspuffers mehr empfangen werden kann, wird dem Sender ein geschlossenes Advertised Window mitgeteilt. Somit wird der Transfer von Paketen, die kleiner als eine MSS sind, effektiv unterbunden. Für den Fall, dass der gesamte Empfangspuffer geringer ist als eine MSS, wird zusätzlich das Versenden solange erlaubt, bis dieser zur Hälfte gefüllt ist. Werden die empfangenen Daten zur Anwendungsschicht mittels `copyData(...)` kopiert, wird erneut freier Platz im Empfangspuffer geschaffen. Sobald genügend Platz für ein Paket maximaler Größe frei ist oder der Empfangspuffer weniger als 50% gefüllt ist, wird dem Sender dies über ein *Window Update* mitgeteilt. Ein Window Update ist eine Empfangsbestätigung mit einem erhöhten Wert für das Advertised Window.

```
aspect SillyWindowSyndrome_Receiver_Avoidance {
  advice execution("% ipstack::TCP_Socket::getReceiveWindow()") &&
         that(socket) && result(window) :
         after(TCP_Socket& socket, UInt16 window){
    if(socket.state != SYNSENT && socket.state != CLOSED){
      if((window*2<socket.maxReceiveWindow) && (window<socket.mss)){
        //artificially close the advertised window until
        // 1) at least 1/2 buffer space is free, or
        // 2) one full-sized segment can be accepted
        *tjp->result() = 0;
      }
    }
  }
  advice call("void ipstack::TCP_ReceiveBuffer::copyData(...)") &&
         that(socket) && within("ipstack::TCP_Socket") :
         around(TCP_Socket& socket) {
    bool receiveWindowClosed = (socket.getReceiveWindow() == 0);
    //remember if receive window is closed before copying data
    tjp->proceed(); //remove data from ReceiveBuffer
    if(receiveWindowClosed && (socket.getReceiveWindow() != 0)){
      //receive window was closed, but we've made enough space
      socket.sendACK(); //send gratuitous ACK ('window update')
    }
  }
};
```

Quelltext 5.2: Vermeidung des SWS durch den Empfänger

Das SWS kann zusätzlich durch den Sender vermieden werden, falls der Empfänger keine solche Vermeidung vornimmt. Die Implementierung ist in diesem Fall etwas aufwändiger, kann aber trotzdem relativ kompakt als separater Aspekt realisiert werden. Vor dem Senden wird dabei überprüft, ob ein Paket maximaler Größe versendet werden kann und ob der maximale Empfangspuffer des Empfängers zu mehr als 50% frei ist. Sind beide Bedingungen nicht erfüllt, wird mit dem Senden gewartet. Erst wenn sich der Empfangspuffer des Empfängers wieder leert, werden neue Pakete versendet. Die Größe des maximalen Empfangspuffers des Empfängers muss dabei abgeschätzt werden, da der Sender diesen Wert nicht kennt.

5.2.4.3 Round-Trip-Time Berechnung

Die Zeit, die zwischen dem Versenden eines TCP-Paketes und dem Eintreffen einer zugehörigen Empfangsbestätigung vergeht, wird als *Round-Trip-Time* (RTT) bezeichnet. Um auf den Verlust eines TCP-Paketes frühzeitig reagieren zu können, kann eine durchschnittliche RTT pro Verbindung berechnet werden, anstatt eine konstante obere Schranke zu definieren. Trifft keine Empfangsbestätigung für ein versendetes Paket innerhalb dieser Zeit ein, kann von einem Paketverlust ausgegangen und das Paket erneut versendet werden. Jacobson [47, S. 323ff] definiert einen effizienten inkrementellen Algorithmus zur Approximation der durchschnittlichen RTT (A) sowie dessen mittlerer Abweichung (D), welche fortlaufend durch eine neue Messung (M) aktualisiert werden:

$$Err = M - A$$
$$A \leftarrow A + \frac{1}{8}Err$$
$$D \leftarrow D + \frac{1}{4}(|Err| - D)$$
$$RTO = A + 4D$$

Das *Retransmission Timeout* (RTO) bezeichnet die Zeitspanne, die vor einem wiederholten Senden eines Paketes gewartet werden muss und ergibt sich zu der approximierten RTT und dessen vierfacher mittlerer Abweichung. In der o. g. Literatur wird der Wert für das RTO zunächst als $A + 2D$ definiert und erst nachträglich auf den Wert $A + 4D$ von Jacobson [zitiert nach Stevens [33, S. 300]] geändert. Die Faktoren der Multiplikationen des Algorithmus' sind als Zweierpotenzen bzw. reziproke Zweierpotenzen gewählt, um eine effiziente Berechnung durch Schiebe-Operationen in Integer-Arithmetik zu erlauben. Dieses Verfahren hat sich in der Praxis bewährt und wird deshalb ebenfalls in der TCP-Implementierung dieser Diplomarbeit eingesetzt. Das Merkmal der RTT Berechnung kann durch einen Aspekt und ein Klassenfragment als eigenständige Komponente implementiert werden. In dem Aspekt wird außerdem der *Karn-Algorithmus* berücksichtigt. „That is, RTT samples MUST NOT[2] be made using segments that were retransmitted (and thus for which it is ambiguous whether the reply was for the first instance of the packet or a later instance)“ [55, S. 3].

[2]Hervorhebungen im Original

5.2.4.4 Slow Start

In dieser Diplomarbeit wird das Verfahren Slow Start (siehe 3.3.3) zur Vermeidung von Überlastsituationen durch TCP implementiert. Auch dieses Verfahren lässt sich in einem Aspekt und einem zugehörigen Klassenfragment separat realisieren. Die Klasse `TCP_Socket` enthält eine Methode `getSendWindow()`, deren Rückgabewert die Anzahl der Bytes darstellt, die aufgrund der TCP-Flusskontrolle versendet werden darf. Erkennt der Aspekt `TCP_SlowStart` eine Überlastsituation durch Paketverluste, wird der Rückgabewert von `getSendWindow()` manipuliert, so dass keine neuen Pakete mehr versendet werden dürfen. Erst nachdem ausreichend Empfangsbestätigungen eingetroffen sind, wird das Versenden von neuen Datenpaketen von diesem Aspekt wieder zugelassen.

5.3 Programmierschnittstelle

Die oberste Schicht der in dem Domänenentwurf entwickelten funktionalen Hierarchie (siehe 4.2) stellt die API dar. Dies ist die Programmierschnittstelle, die der Anwendungsschicht den Zugang zu dem TCP/IP-Stack ermöglicht. Alle für eine Anwendung relevanten Funktionen des TCP/IP-Stacks werden in der API zusammengefasst. In dieser Diplomarbeit wird eine Programmierschnittstelle implementiert, die der *Sockets Networking API* [37] konzeptionell ähnelt. Allerdings wird hier eine objektorientierte Implementierung realisiert, die zudem durch Templates konfigurierbar ist. In Quelltext 5.3 ist die Signatur der Schnittstellenklasse für eine TCP-Verbindung mit IPv4 gezeigt.

```
ipstack::api::IPv4_TCP_Socket<unsigned BLOCKSIZE_1,
                              unsigned COUNT_1,
                              unsigned BLOCKSIZE_2,
                              unsigned COUNT_2,
                              unsigned RINGBUFFERSIZE>
```

Quelltext 5.3: Schnittstellenklasse der API für TCP und IPv4

Die Instanziierung eines Objektes dieser Klasse ermöglicht einer Anwendung, einen Datentransfer über TCP abzuwickeln. Jedes Objekt dieser Klasse enthält eine eigene Speicherverwaltung, welche durch einen `BasicMempool` (siehe 4.3) realisiert wird. Die Konfiguration der Speicherverwaltung erfolgt über fünf optionale Template-Parameter, die mit Standardwerten vorbelegt sind. `BLOCKSIZE_1` und `BLOCKSIZE_2` legen zwei Paketgrößen fest, die von der Speicherverwaltung unterstützt werden sollen. Da eine TCP-Verbindung häufig gleichzeitig große und kleine Pakete überträgt (siehe 4.3), sind beide Parameter mit den Standardwerten 1514 bzw. 128 vorbelegt. Die Parameter `COUNT_1` und `COUNT_2` definieren, wie viele Pakete der zugehörigen Größe die Speicherverwaltung umfassen soll. Der letzte Template-Parameter `RINGBUFFERSIZE` stellt die Größe des Empfangspuffers der TCP-Verbindung ein, d. h. die Anzahl von Paketen, die empfangen werden können, ohne dass sie direkt verarbeitet werden müssen. Wird dieser Parameter nicht explizit spezifiziert, ergibt er sich zu `COUNT_1 + COUNT_2`. Die Konfigurierung der Klasse

`ipstack::api::IPv4_TCP_Socket` über Template-Parameter ist nur dann möglich, wenn die Speicherverwaltung (siehe 4.3) unterschiedliche Mempools unterstützt. Wird in der Speicherverwaltung global festgelegt, dass es nur eine Variante von Mempools gibt, werden die o. g. Template-Parameter ignoriert und stattdessen die genannten Standardwerte verwendet. Für einen Datentransfer über UDP gibt es eine äquivalente Schnittstellenklasse `ipstack::api::IPv4_UDP_Socket`, deren Template-Parameter und Instanziierung identisch sind.

Methode	**Aufgabe**
`void set_dst_ipv4_addr(a, b, c, d)`	setzt die IPv4-Adresse des Ziels
`void set_dport(port)`	setzt die Portnummer des Ziels
`void set_sport(port)`	setzt die eigene Portnummer
`bool send(data, datasize)`	Daten versenden
`bool connect()`	TCP-Verbindung aktiv aufbauen
`bool listen()`	TCP-Verbindung passiv aufbauen
`bool close()`	TCP-Verbindung terminieren
`int receive(buffer, buffer_len)`	TCP-Daten empfangen
`bool bind()`	UDP-Datenempfang registrieren
`void unbind()`	UDP-Datenempfang abmelden
`ipstack::IPv4_Packet* receive()`	UDP-Paket über IPv4 empfangen
`void free(packet)`	UDP-Paket freigeben

Tabelle 5.1: Methoden der Programmierschnittstelle

Die Methoden der Schnittstellenklassen, die zur Kommunikation nötig sind, werden in der Tabelle 5.1 zusammengefasst. Zu einer besseren Übersicht sind lediglich die wichtigsten Methoden aufgelistet. Die obersten vier sind sowohl für TCP als auch für UDP gültig. Das betrifft die Einstellung der IPv4-Zieladresse und der TCP- bzw. UDP-Portnummern. Die Signatur der Methode für das Versenden von Daten ist bei beiden Protokollen ebenfalls identisch. Allerdings gibt es je nach Protokoll eine unterschiedliche Semantik. Bei TCP werden die zu versendenden Daten automatisch segmentiert, d. h. in IPv4-Pakete optimaler Größe eingeteilt. Das Versenden von UDP-Paketen gelingt hingegen nur dann, wenn diese nicht größer als die MTU des verwendeten Netzwerkgerätes sind. Die Einhaltung dieser Bedingung wird bei UDP der Anwendungsschicht überlassen. Alle in der Tabelle aufgelisteten Methoden, die einen Rückgabewert vom Typ `bool` haben, zeigen den Erfolg der Methodenausführung mit dem Rückgabewert `true` an. Dies gilt z. B. für die Methoden zum TCP-Verbindungsaufbau und zum Terminieren einer TCP-Verbindung, die lediglich in der Klasse `ipstack::api::IPv4_TCP_Socket` vorhanden sind. Das Empfangen von Nutzdaten über eine TCP-Verbindung wird durch die Methode `receive(buffer, buffer_len)` realisiert, deren Argumente ein Zeiger auf einen Datenpuffer der Anwendung und die Länge dieses Puffers sind. Der Rückgabewert vom Typ `int` gibt die Anzahl der empfangenen Bytes zurück. Ein negativer Rückgabewert deutet auf einen Fehler beim Empfangsvorgang hin.

Die unteren vier Methoden der Tabelle 5.1 gelten nur für das Protokoll UDP. Mit `bind()` und `unbind()` ist eine Registrierung und Abmeldung für den Empfang von UDP-Paketen möglich (siehe 4.7.2). Damit wird festgelegt, dass alle UDP-Pakete mit den passenden Portnummern von dem registrierten Objekt entgegen genommen werden. Der Empfang eines UDP-Paketes erfolgt schließlich mit der Methode `receive()`. Deren Rückgabewert ist ein vollständiges IPv4-Paket inklusive IPv4-Header, UDP-Header und den Nutzdaten. Diese Header-Informationen sind für eine Kommunikation über UDP notwendig, da es keine explizite Verbindung zwischen zwei Kommunikationspartnern gibt, sondern mehrere Absender für ein Paket möglich sind. Die Freigabe eines empfangenen UDP-Paketes wird durch `free(packet)` erreicht.

5.4 CiAO Integration

Der in dieser Diplomarbeit entworfene und implementierte TCP/IP-Stack wird zur Evaluation in die CiAO Betriebssystemfamilie (siehe 2.5) integriert. Dieses Vorgehen ermöglicht eine praktische Anwendung des TCP/IP-Stacks in Kombination mit einem Betriebssystem für eingebettete Systeme. Die Integration erfolgt *nicht-invasiv* durch den Einsatz von Aspekten, d. h. weder der TCP/IP-Stack noch CiAO müssen modifiziert werden. Nachfolgend wird zuerst auf die Unterstützung des TCP/IP-Stacks durch das Betriebssystem eingegangen, um effizient auf externe Ereignisse reagieren zu können (siehe 5.4.1). Damit verbunden ist die Unterbrechungssynchronisation, die daran anschließend behandelt wird. In Abschnitt 5.4.3 wird beschrieben, wie das Variantenmanagement von CiAO für diesen TCP/IP-Stack übernommen wird.

5.4.1 Betriebssystemunterstützung

Die minimale Implementierung für das Empfangen von Datenpaketen mit dem in dieser Diplomarbeit entwickelten TCP/IP-Stack umfasst ein *aktives Warten* („*Polling*“). Dieses Prinzip ist lediglich in extrem seltenen Ausnahmefällen sinnvoll einsetzbar. Eine unterbrechungsgesteuerte Empfangsfunktion wird in der Regel bevorzugt, so dass die Hardware in einen Energiesparmodus versetzt werden kann, solange auf den Empfang eines Datenpaketes gewartet wird. Die dafür notwendige Funktionalität stellt ein Betriebssystem zur Verfügung. In CiAO wird dies mit dem Konzept eines *AUTOSAR OS Events* [21] erreicht, welches grundlegende Synchronisation ermöglicht. Ein Task kann auf ein Event warten und wird somit *blockiert*, d. h. er wird nicht mehr ausgeführt. Weiterhin kann ein Event gesetzt werden und damit ggf. ein blockierter Task wieder aktiviert werden. CiAO verfügt über zwei Funktionen `AS::WaitEvent(event)` und `AS::SetEvent(taskid, event)`, welche die beschriebene Funktionalität implementieren. Die Verwendung dieser Funktionen innerhalb des TCP/IP-Stacks wird durch einen Policy-Aspekt veranlasst, der zwei Methoden des TCP/IP-Stacks beeinflusst. Die Empfangsmethode `receive()` der Klasse `IPv4_Socket` (siehe 4.6) wird dahingehend erweitert, dass die Funktion `AS::WaitEvent` aufgerufen wird, sofern kein Datenpaket im Empfangspuffer vorhanden ist. Wird ein empfangenes Datenpaket von einer Unter-

brechungsbehandlung in den Empfangspuffer mittels `put(packet, len)` kopiert, kann der Join-Point dieser Methode dazu benutzt werden, um die Funktion `AS::SetEvent` aufzurufen. Auf diese Weise wird ein blockierender Empfangsvorgang implementiert. Die Verwendung der Betriebssystemfunktionen findet implizit durch einen Policy-Aspekt statt, der beliebig austauschbar ist. Bietet ein anderes Betriebssystem z. B. Semaphore zur Synchronisation an, können diese durch einen entsprechenden Policy-Aspekt ebenfalls benutzt werden, ohne dass der TCP/IP-Stack dafür angepasst werden muss.
Das Protokoll TCP benötigt darüber hinaus eine Funktion, die das Warten einer definierten Zeit ermöglicht. Das ist z. B. für eine wiederholte Versendung eines TCP-Paketes notwendig, welche nach etwa einer RTT erfolgt (siehe 5.2.4). Aus diesem Grund enthält die Klasse `TCP_Socket` einen expliziten Join-Point mit der Signatur `block(timeout)`, der eine leere Implementierung aufweist. Diese Methode wird immer dann aufgerufen, wenn für eine Zeit von `timeout` Millisekunden gewartet werden soll. Das Betriebssystem CiAO verfügt mit dem Konzept eines *AUTOSAR OS Alarms* [21] über genau diese Funktionalität. Ein Alarm kann in CiAO dazu benutzt werden, nach einer definierten Zeit einen Event auszulösen. Die Implementierung der o. g. Methode `block(timeout)` lässt sich somit durch einen Policy-Aspekt realisieren. Zu Beginn dieser Methode wird ein Alarm auf die zu wartende Zeit festgelegt. Mit `AS::WaitEvent` wird anschließend auf das Eintreten des Events gewartet, welches der Alarm auslöst. Diese Vorgehensweise ermöglicht eine flexible Integration des TCP/IP-Stacks in andere Betriebssysteme, die möglicherweise unterschiedliche Funktionen zum zeitgesteuerten Warten anbieten. Sämtliche Verbindungen zwischen dem TCP/IP-Stack und CiAO werden durch austauschbare Policy-Aspekte hergestellt.

5.4.2 Unterbrechungssynchronisation

Wird eine Datenstruktur sowohl von einer Anwendung als auch von einem Betriebssystemkern oder einer Unterbrechungsbehandlung verändert, ist eine Synchronisation notwendig. In dem bisher beschriebenen TCP/IP-Stack betrifft das z. B. die Speicherverwaltung (siehe 4.3). Empfangene Pakete werden von einer Unterbrechungsbehandlung in den Speicherbereich einer Anwendung kopiert, während versendete Pakete ebenfalls in dem Speicherbereich der Anwendung erstellt werden. Aus diesem Grund muss die Konsistenz der Speicherverwaltung durch eine Unterbrechungssynchronisation sichergestellt werden. Das Betriebssystem CiAO enthält für diesen Zweck die Funktionen `os::krn::enterKernel()` und `os::krn::leaveKernel()`. Die Unterbrechungssynchronisation des TCP/IP-Stacks kann somit durch einen separaten Aspekt implementiert werden. In diesem Aspekt wird ein Pointcut definiert, der alle relevanten Methoden umfasst, die synchronisiert werden sollen. Ein Around-Advice ermöglicht somit die Synchronisation, indem die o. g. Funktionen von CiAO zur Synchronisation vor bzw. nach der Ausführung der in dem Pointcut definierten Methoden platziert werden.
Des Weiteren muss der Zugriff auf die Netzwerkhardware synchronisiert werden. Ein ARP-Request veranlasst z. B. in einer Unterbrechungsbehandlung das Versenden eines zugehörigen ARP-Replies, während eine Anwendung ebenfalls Datenpakete versendet. Damit die Unterbrechungsbehandlung verzögert wird, werden wiederum die o. g. Funk-

tionen zur Synchronisation eingesetzt. Quelltext 5.4 zeigt den dafür verwendeten Advice. Die Verbindung zwischen dem TCP/IP-Stacks und CiAO besteht bei der Unterbrechungssynchronisation lediglich aus einem Aspekt. Die Integration des TCP/IP-Stacks in ein anderes Betriebssystem mit unterschiedlichen Synchronisationsmechanismen ist folglich problemlos möglich.

```
advice call("void ipstack::Interface::send(const void*, unsigned)") &&
       within("ipstack::IPv4_Socket") : around(){
  os::krn::enterKernel(); // lock (defer ISR)
  tjp->proceed(); // execute the original code
  os::krn::leaveKernel(); // unlock
}
```

Quelltext 5.4: Beispiel der Unterbrechungssynchronisation des TCP/IP-Stacks in CiAO

5.4.3 Variantenmanagement

Die Konfiguration der CiAO Betriebssystemfamilie wird durch das Werkzeug *pure::variants* [56] unterstützt. Dieses Werkzeug ermöglicht das Variantenmanagement einer Software-Produktlinie und verfügt über ein Plugin für die integrierte Entwicklungsumgebung *Eclipse*. Es lassen sich Merkmalmodelle (siehe 3.3.3) erstellen, welche die verfügbaren Merkmale einer Produktlinie definieren. Ein *Familienmodell* repräsentiert die wiederverwendbaren Softwarekomponenten der Produktlinie. Diese Softwarekomponenten lassen sich den Merkmalen der Merkmalmodelle zuordnen. Eine automatisierte Generierung einer Variante der Software-Produktline anhand einer Merkmalselektion wird somit ermöglicht. Das *Variantenmodell* legt die Merkmale fest, die in der zu generierenden Variante der Produktlinie vorhanden sein sollen. Eine Transformation überführt diese Merkmalselektion in ein konkretes Produkt, indem eine Auswahl der o. g. Softwarekomponenten getroffen wird.
Die Integration des in dieser Diplomarbeit entwickelten TCP/IP-Stacks in das Betriebssystem CiAO umfasst ebenfalls eine Erstellung von Modellen für pure::variants. Auf diese Weise wird eine durchgängige Konfigurierung von CiAO in Kombination mit dem TCP/IP-Stack durch lediglich ein Werkzeug realisiert. In Abbildung 5.2 wird ein Variantenmodell des TCP/IP-Stacks und eine exemplarische Merkmalselektion gezeigt. Die in dieser Abbildung dargestellten Merkmale sind vollständig implementiert, so dass sich eine enorme Anzahl von unterschiedlichen Produkten ergibt, je nachdem welche Merkmale ausgewählt sind. Mit pure::variants wird eine komfortable grafische Konfigurierung des TCP/IP-Stacks erreicht, so dass bei der Generierung einer Produktlinienvariante keine Quelltexte bearbeitet werden müssen.

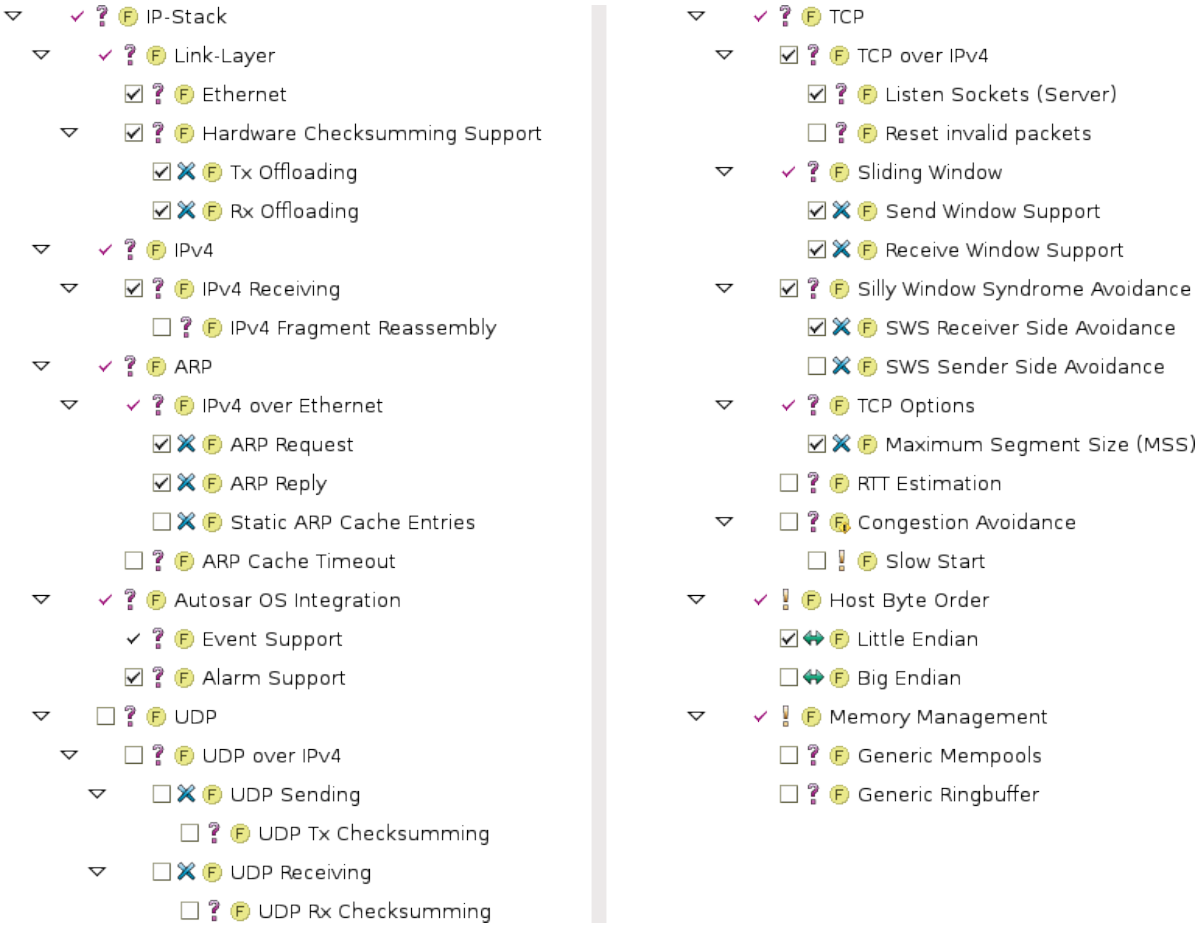

Abbildung 5.2: Variantenmodell des TCP/IP-Stacks und eine exemplarische Merkmalselektion

6 Diskussion und Bewertung

Nach der Analyse, dem Entwurf und der Implementierung erfolgt in diesem Kapitel die Evaluation des in dieser Diplomarbeit entwickelten TCP/IP-Stacks für eingebettete Systeme. Zuerst wird der Quelltext der implementierten Software-Produktlinie hinsichtlich softwaretechnischer Eigenschaften analysiert und bewertet (siehe 6.1). In Abschnitt 6.2 folgt die Betrachtung der Performanz des TCP/IP-Stacks unter verschiedenen Bedingungen. Es werden Vergleiche zu anderen Systemen gezogen und die Laufzeitkosten der unterschiedlichen Systeme gegenübergestellt. Abschließend wird der Speicherbedarf des TCP/IP-Stacks detailliert beschrieben (siehe 6.3). Die Überprüfung, ob sich dieser TCP/IP-Stack für eingebettete Systeme mit wenig Speicher eignet, wird am Ende dieses Kapitels vorgenommen.

6.1 Analyse des Quelltextes

Die nachfolgenden Abschnitte diskutieren die Eigenschaften des Quelltextes der in dieser Diplomarbeit entwickelten Software-Produktlinie. Zentraler Gesichtspunkt der Betrachtung ist der Nutzen, der aus AOP gewonnen werden kann. Zunächst wird in Abschnitt 6.1.1 die Trennung der Belange innerhalb der Implementierung untersucht, die bereits als hervorstechendes Merkmal von AOP aufgezeigt wurde (siehe 2.4.1). Anschließend wird auf eine Erweiterung der entwickelten Produktlinie um zusätzliche Funktionalität eingegangen. Dabei wird ermittelt, ob und wie die Erweiterbarkeit von Software durch AOP profitieren kann.

6.1.1 Trennung der Belange

AOP ermöglicht eine Trennung der Belange in der Software-Implementierung, die mit klassischen Methoden nicht erreicht werden kann. Die Realisierung von querschneidenden Belangen als eigenständige Aspekte innerhalb des TCP/IP-Stacks, der in dieser Diplomarbeit entwickelt wurde, wird im Folgenden exemplarisch anhand der Konvertierung der Byte Order erläutert. In Abschnitt 5.2.1 wurde bereits beschrieben, dass die Verarbeitung von Datenpaketen auf einem Little-Endian System eine Konvertierung der Byte Order voraussetzt. Diese Konvertierung beeinflusst viele Komponenten des TCP/IP-Stacks und ist somit ein querschneidender Belang. Die klassische Implementierungsweise dieses Belangs beruht auf der Verwendung des C-Präprozessors und wird z. B. in lwIP (siehe 3.2) eingesetzt. Es werden u. a. die Makros `htons` und `ntohs` angelegt, welche die Konvertierung von 16 Bit Werten durchführen. Der Zugriff auf einen Header eines Datenpaketes wird um die Benutzung dieser Makros erweitert. An jeder Stelle im Quelltext, an der die

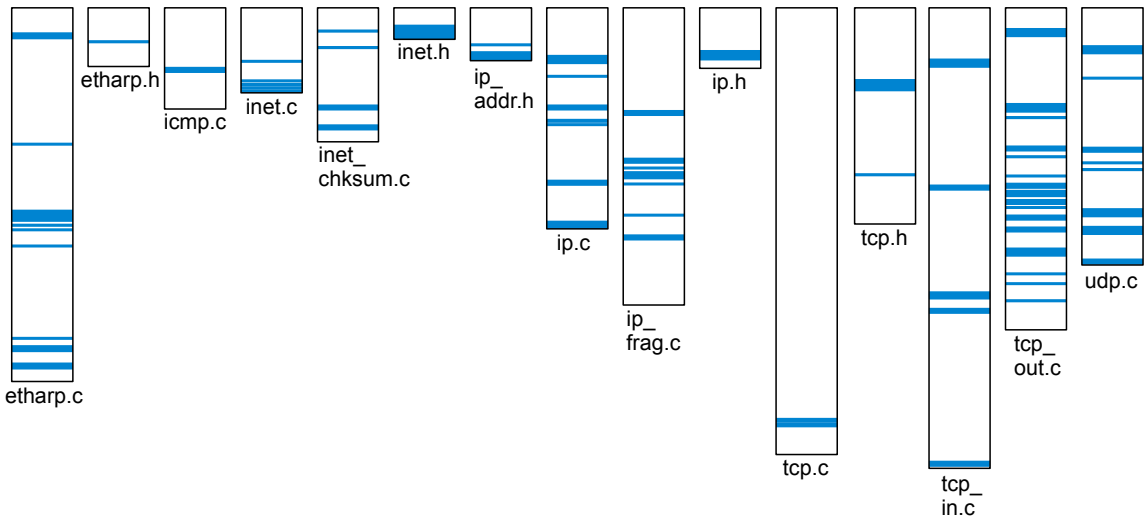

Abbildung 6.1: Konvertierung der Byte Order durch Makros in lwIP

Konvertierung bzgl. der Byte Order vorgenommen wird, muss ein Makro benutzt werden. Abbildung 6.1 veranschaulicht die Verwendung der Makros zur Konvertierung der Byte Order in lwIP[1]. Die vertikalen Rechtecke symbolisieren einzelne Dateien von lwIP, deren Höhen die Zeilenanzahl der jeweiligen Dateien widerspiegeln. Eine horizontale blaue Linie markiert eine Zeile in einer Datei, in der eines der o. g. Makros benutzt wird. Das aufeinander folgende Auftreten der Makros resultiert folglich in einer breiteren blauen Linie. Insgesamt werden die Makros zur Konvertierung der Byte Order innerhalb von lwIP 210 Mal in 15 Dateien verwendet. Die Implementierung des Belangs der Byte Order ist folglich über sehr viele Komponenten von lwIP verteilt.
Der aspektorientierte Entwurf des TCP/IP-Stacks dieser Diplomarbeit hingegen ermöglicht die Implementierung der o. g. Konvertierung als einzelne Komponente. Somit wird die vollständige Funktionalität der Konvertierung bzgl. der Byte Order in lediglich einem Aspekt bzw. einer Datei verkapselt (siehe 5.2.1). Daraus resultieren einige Vorteile für die anderen Komponenten des TCP/IP-Stacks. Zum einen wird die Lesbarkeit des Quelltextes durch den Verzicht auf Makros erhöht. Zum anderen wird die Fehleranfälligkeit reduziert, da nicht bei jedem Zugriff auf einen Paket-Header die korrekte Benutzung der Makros eingehalten werden muss. Die Wartbarkeit und Wiederverwendbarkeit des Quelltextes wird gesteigert, denn die Kapselung des Belangs in einem Aspekt spart dutzende Zeilen an Quellcode ein.

6.1.2 Erweiterbarkeit

Neben der Trennung der Belange kann durch AOP ein hohes Potenzial für die Erweiterbarkeit einer Software geschaffen werden. Optionale Merkmale können mit wenig Aufwand in ein bestehendes System integriert werden, ohne dass die Komponenten des Systems

[1] Version 1.32

selbst verändert werden müssen. Veranschaulicht werden kann die Erweiterbarkeit exemplarisch anhand eines Paketfilters. Der in dieser Diplomarbeit entwickelte TCP/IP-Stack verfügt nicht über die Funktionalität, absichtlich manipulierte Pakete zu erkennen und gesondert zu behandeln. Diese Funktion fällt üblicherweise in den Aufgabenbereich eines Paketfilters wie z. B. *pf* [57]. Es können Regeln definiert werden, die bestimmte Pakete abdecken und von einer Datenübertragung ausschließen. Ein Paketfilter könnte z. B. alle TCP-Pakete aussortieren, die eine ungültige Kombination der TCP-Flags aufweisen. Das sind u. a. Pakete, die einen Verbindungsaufbau sowie eine Terminierung der Verbindung gleichzeitig anfordern. Ein solches Paket enthält das Flag `SYN` zum Verbindungsaufbau und mindestens eines der Flags `FIN` oder `RST` zur Terminierung. Derartige Pakete deuten darauf hin, dass ein *Portscan* des Systems durchgeführt wird [58, S. 660f]. Die Erkennung solcher Pakete kann als separater Aspekt in wenigen Zeilen Quelltext implementiert werden. In Quelltext 6.1 ist die Implementierung gezeigt, die empfangene TCP-Pakete auf die o. g. Kombinationen der Flags überprüft und ggf. verwirft. Mit einem Order-Advice wird zunächst sichergestellt, dass der Paketfilter-Aspekt für die TCP-Pakete vor dem Upcall-Dispatcher-Aspekt `IPv4_TCP_Receive` ausgeführt wird. Ein Around-Advice greift direkt in die Upcall-Dispatcher-Hierarchie (siehe 4.7.1) ein und stoppt den Kontrollfluss durch diese Hierarchie mittels `return`, falls ungültige TCP-Flags erkannt werden. Andere Aktionen an dieser Stelle, wie z. B. das Ausgeben einer Warnmeldung oder das Abspeichern des manipulierten Paketes, sind ebenfalls denkbar. Die übrigen Pakete, die nicht von diesem Paketfilter-Aspekt aussortiert werden, gelangen über `tjp->proceed()` zurück in die Upcall-Dispatcher-Hierarchie. Die an diesem Beispiel demonstrierte Erweiterbarkeit wird vor allem durch den *aspektgewahren Entwurf* des TCP/IP-Stacks ermöglicht, in

```
aspect IPv4_TCP_Filter {
  //always execute this aspect before 'IPv4_TCP_Receive'
  advice execution("void ipstack::Demux::ipv4_demux(...)") :
         order("IPv4_TCP_Filter", "IPv4_TCP_Receive");

  advice execution("void ipstack::Demux::ipv4_demux(...)") &&
         args(packet, len, dev) :
         around(IPv4_Packet* packet, unsigned len, Interface* dev){

    if(packet->get_protocol() == TCP_Segment::IPV4_TYPE_TCP){
      TCP_Segment* tcps = (TCP_Segment*) packet->get_data();

      if(tcps->has_SYN() && (tcps->has_FIN() || tcps->has_RST())){
        //match invalid combination of SYN+FIN or SYN+RST flags
        return; //drop illegal packets
      }
    }
    tjp->proceed(); //flags are valid
  }
};
```

Quelltext 6.1: Filterung ungültiger TCP-Flags mit einem optionalen Aspekt

dem zukünftige Erweiterungen bereits eingeplant sind. Somit wird die Wartbarkeit und Wiederverwendbarkeit der Software deutlich verbessert, da zentrale Komponenten des Systems mit wenig Aufwand um zusätzliche Funktionalität erweitert werden können. Mit den anderen softwaretechnischen Methoden, wie z. B. dem C-Präprozessor oder Templates, die in Abschnitt 2.4.1 beschrieben wurden, kann demgegenüber keine Erweiterbarkeit mit derartiger Flexibilität erreicht werden.

6.2 Performanz

In diesem Abschnitt wird die Effizienz des in dieser Diplomarbeit entwickelten TCP/IP-Stacks untersucht. Dazu wird der TCP/IP-Stack zusammen mit der CiAO Betriebssystemfamilie (siehe 5.4) verwendet. Des Weiteren werden die TCP/IP-Stacks lwIP[2] und uIP[3] (siehe 3.2) ebenfalls in CiAO integriert, um Vergleichsmöglichkeiten zu schaffen. Im Nachfolgenden wird ausschließlich die Portierung von CiAO auf die IA-32 benutzt, da in dieser Architekturvariante von CiAO ein Treiber für einen Ethernet-Adapter vom Typ Intel Pro/1000 existiert. Die Konfigurationen aller TCP/IP-Stacks umfassen die Einstellungen zur Erreichung einer maximalen Datenrate und die Benutzung derselben Treiber für den Hardwarezugriff. Der in dieser Diplomarbeit entwickelte TCP/IP-Stack, der im Folgenden als *CiAO IP-Stack* bezeichnet wird, sowie lwIP werden mit identischen Speicherbereichen von jeweils 100 KByte zum Puffern von Datenpaketen ausgestattet. uIP hingegen unterstützt lediglich das Puffern von einem Datenpaket aufgrund des fehlenden Sliding Windows und wird deswegen mit einem Speicherbereich ausgestattet, der genau ein Paket maximaler Größe aufnehmen kann. Zusätzlich wird uIP um das Modul `uip_split` [59, S. 89] erweitert, um einen maximalen Durchsatz beim Senden zu erreichen. Das ist nötig, weil uIP jeweils nur ein Paket versendet und anschließend auf eine Empfangsbestätigung wartet. Wird im Empfänger der Algorithmus der *Delayed Acknowledgements* [27, S. 96f] eingesetzt, erfolgt das Bestätigen von Datenpaketen in der Regel erst nach 250 Millisekunden oder nach dem Empfang eines zweiten Datenpaketes. Mit `uip_split` wird jedes versendete Datenpaket in zwei Pakete aufgeteilt, so dass keine Delayed Acknowledgements entstehen.
Als Bewertungsmaßstab wird die Datenrate gewählt, die über eine TCP-Verbindung übertragen werden kann. Es werden lediglich die übertragenen Nutzdaten betrachtet und es wird zwischen der Performanz beim Senden und Empfangen unterschieden. Als Kommunikationspartner dient bei allen Messungen ein Linux[4]-System. Zuerst wird die Performanz einer Datenübertragung zwischen zwei Rechnern, die über ein Gigabit Ethernet miteinander verbunden sind, ermittelt (siehe 6.2.1). Anschließend finden in Abschnitt 6.2.2 Messungen in einer virtuellen Maschine statt, um Rückschlüsse auf die Prozessorauslastung durch die verschiedenen TCP/IP-Stacks zu ziehen.

[2]Version 1.32
[3]Version 1.0
[4]Version 2.6.32

6.2.1 Gigabit Ethernet

Zwei Desktop-Rechner vom Typ *Dell Optiplex 755* mit je einem *Intel Core 2 Quad 6600* Prozessor und je einem integrierten *Intel 82566DM* Gigabit Ethernet-Adapter, der zur Intel Pro/1000 Reihe zählt, dienen als Testsysteme. Beide Rechner sind direkt durch ein *Cat-5e-Kabel* miteinander verbunden. Einer der beiden Rechner ist mit Linux ausgestattet, während der andere Rechner das zu untersuchende System darstellt. Für die Ermittlung eines Referenzwertes wird auf dem Testsystem ebenfalls Linux eingesetzt, um die maximale praktisch erreichbare Datenrate zu bestimmen. Abbildung 6.2 stellt die Messergebnisse grafisch dar. Eine optimale Datenübertragung zwischen zwei Linux-Systemen ist auf 937 MBit/s limitiert. Dieser Wert entspricht fast der physikalisch möglichen Datenrate, wenn die Protokoll-Header berücksichtigt werden:

$$\underbrace{1460}_{\substack{\text{Nutzdaten}\\ \text{pro Paket}}} \Big/ \Big(1460 + \underbrace{20}_{\substack{\text{TCP-}\\ \text{Header}}} + \underbrace{20}_{\substack{\text{IP-}\\ \text{Header}}} + \underbrace{14}_{\substack{\text{Ethernet-}\\ \text{Header}}} + \underbrace{24}_{\substack{\text{Ethernet-}\\ \text{Präambel,}\\ \text{Trailer, IFG}}} \Big) = 94.9\%$$

Der in dieser Diplomarbeit entwickelte CiAO IP-Stack erreicht diesen Wert nahezu und weicht wie Linux nur geringfügig davon ab. lwIP bleibt hingegen deutlich hinter den beiden genannten Systemen zurück, während uIP aufgrund des fehlenden Sliding Windows nicht konkurrieren kann.

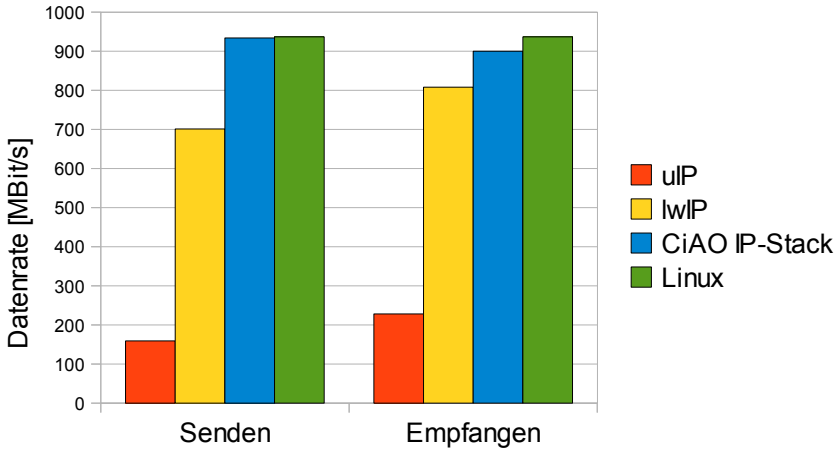

Abbildung 6.2: TCP-Durchsatz zwischen zwei Desktop-Rechnern

6.2.2 Virtuelle Maschinen

Die Messungen in einer virtuellen Maschine finden ausschließlich auf einem Rechner statt. Das Testsystem wird mittels *QEMU*[5] [60] und *KVM*[5] [61] jeweils als Gastsystem emuliert und verfügt dabei über eine virtuelle *Intel 82540EM* Ethernet-Schnittstelle,

[5]Version 0.12.3 (qemu-kvm)

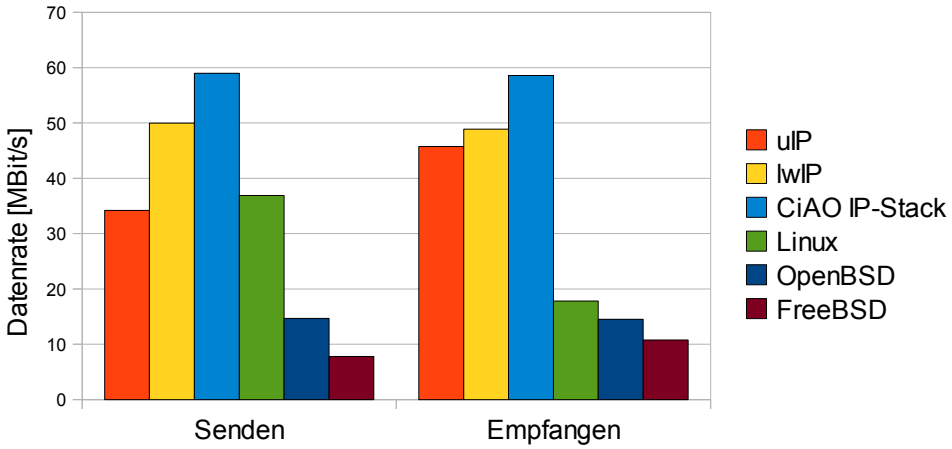

Abbildung 6.3: Virtuelle TCP-Verbindung auf einem Intel Atom N270 mit QEMU

die ebenfalls zur Intel Pro/1000 Reihe zählt. Ein *TAP-Adapter* [62, S. 94ff] verbindet das Linux-Hostsystem mit dem emulierten Gastsystem und stellt somit ein virtuelles Ethernet her. Gemessen wird die Datenrate, die über eine TCP-Verbindung zwischen dem Gast- und dem Hostsystem erreicht werden kann. Durch die Virtualisierung der Netzwerkverbindung werden die erzielten Datenraten lediglich durch den Prozessor limitiert. Eine hohe Datenrate resultiert aus einer geringen Prozessorauslastung durch den TCP/IP-Stack. In Abbildung 6.3 sind die Datenraten gezeigt, die auf einem *Intel Atom N270* Prozessor mit 1,6 GHz übertragen werden können. Für diese Messung wird lediglich QEMU eingesetzt, da der Prozessor über keine Hardwarevirtualisierungsfunktionen verfügt und somit KVM nicht unterstützt wird. Zusätzlich werden OpenBSD[6] und FreeBSD[7] als Testsysteme hinzugezogen. Der CiAO IP-Stack, lwIP und uIP schneiden in diesem Test durchschnittlich betrachtet am besten ab. Verursacht wird das vor allem durch die geringe Größe der Systeme und dadurch, dass keine Trennung zwischen User- und Supervisor-Modus besteht und somit kaum Kosten bei einem Kontextwechsel anfallen. Die Ergebnisse einer wiederholten Messung auf einem Rechner mit *AMD Phenom 9650* Prozessor, der vier Prozessorkerne mit je 2,3 GHz enthält und eine Benutzung von KVM erlaubt, zeigt Abbildung 6.4. Die Resultate unter Verwendung von QEMU sind tendenziell identisch zu der o. g. Messung auf dem Intel Atom N270 Prozessor. Der CiAO IP-Stack profitiert deutlich von dem schnelleren Prozessor. lwIP und uIP können sich hingegen lediglich beim Empfangen steigern. Der Einsatz von KVM, welche die Hardwarevirtualisierungsfunktionen des Prozessors nutzt, produziert stattdessen unterschiedliche Ergebnisse. Linux und FreeBSD werden beschleunigt, während die Performanz aller anderen Systeme durchschnittlich halbiert wird. Vermutlich ist KVM auf die Virtualisierung von Linux und FreeBSD hin optimiert, so dass lediglich ein Vergleich der in CiAO integrierten TCP/IP-Stacks untereinander aussagekräftig ist. In allen Messungen auf beiden Prozessoren, sowohl mit QEMU als auch mit KVM, erreicht

[6] Version 4.6
[7] Version 8.1

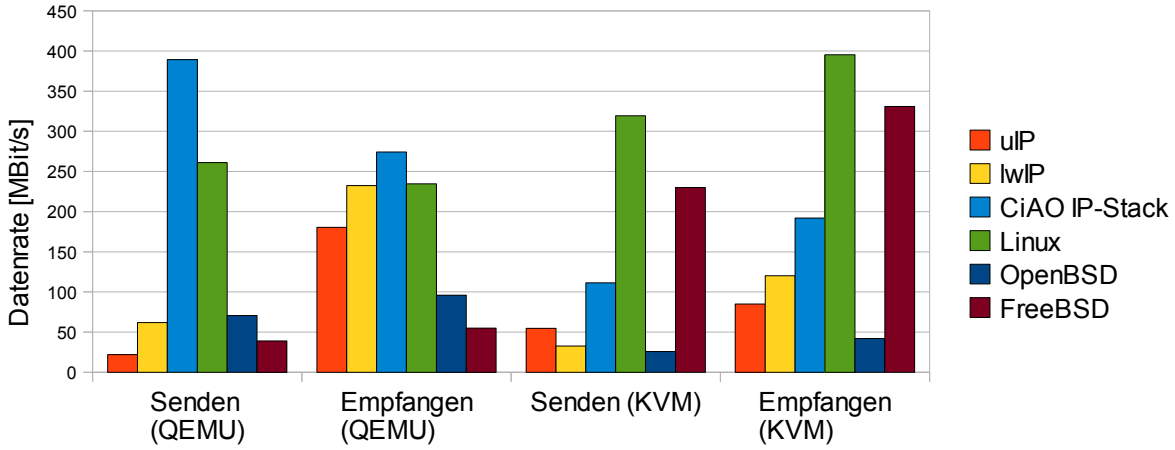

Abbildung 6.4: Virtualisierung mit QEMU und KVM auf einem AMD Phenom 9650

der CiAO IP-Stack eine höhere Datenrate als lwIP und uIP. Daraus folgt, dass die Implementierung des CiAO IP-Stacks effizient bzgl. der Prozessorauslastung ist und somit für eingebettete Systeme geeignet ist.

6.3 Speicherbedarf

Die Speichergröße einer Software ist entscheidend dafür, ob sie auf einem eingebetteten System verwendetet werden kann, da viele eingebettete System lediglich über sehr begrenzten Speicherplatz verfügen. Der in dieser Diplomarbeit entwickelte TCP/IP-Stack stellt eine hochgradig konfigurierbare Software-Produktlinie dar, deren Variationspunkte in Abbildung 5.2 bereits veranschaulicht wurden. Jedem der konfigurierbaren Merkmale des TCP/IP-Stacks kann ein Speicherbedarf zugeordnet werden. Somit lässt sich abschätzen, wie viel Speicher eine bestimmte Merkmalselektion benötigt. Eine Zuordnung des Speicherbedarfs zu jedem Merkmal ist in Tabelle 6.1 aufgelistet. Die Kosten betreffen die `text`-Sektion der Software bei einer Kompilierung für die IA-32 mit *g++*[8] und der Optimierung `-Os` bzgl. der Programmgröße. Diese Kosten spiegeln die Codegröße der Software in Byte wider, die ein eingebettetes System als ROM zur Verfügung stellen muss. Die Größen der `data`- und `bss`-Sektionen, die dem benötigten RAM entsprechen, werden allein durch die Speicherverwaltung (siehe 5.3) dominiert und von der Merkmalselektion nur geringfügig beeinflusst. Die ersten vier Merkmale auf der linken Seite der Tabelle betreffen die Unterstützung von Netzwerkgeräten. Das Merkmal *Polymorphic NetworkDevice* ermöglicht beispielsweise eine gleichzeitige Verwendung von verschiedenen Netzwerkgeräten. Wird nur ein Gerät benutzt, können durch die Implementierung des kostenneutralen Strategie-Entwurfsmusters (siehe 5.1) 546 Byte eingespart werden. Die darauf folgenden vier Merkmale stellen die Kosten für die Implementierung des ARP-Caches dar (siehe 3.3.3). Ein statischer Cache verursacht lediglich 131 Byte an Speicherkosten, während

[8] Version 4.3.2

Merkmal	**Kosten**
Polymorphic NetworkDevice	546
Ethernet Support	219
Tx Checksum Offloading	530
Rx Checksum Offloading	33
Static ARP Cache	131
ARP Reply	368
ARP Request	747
ARP Cache Timeout	97
IPv4 Sending	606
IPv4 Receiving	427
IPv4 Fragment Reassembly	747
AUTOSAR OS Event Support	98
AUTOSAR OS Alarm Support	1888
Generic Mempools	-13
Generic Ringbuffer	104

Merkmal	**Kosten**
UDP Sending	584
UDP Receiving	716
UDP Tx Checksumming	222
UDP Rx Checksumming	198
TCP Support	9692
TCP Client Sockets	1710
TCP Listen Sockets (Server)	2397
Sliding Window (Sending)	1340
Sliding Window (Receiving)	693
TCP MSS Option	547
RTT Estimation	760
Slow Start	300
Reset Invalid TCP Packets	788
SWS Avoidance (Receiver)	496
SWS Avoidance (Sender)	332

Tabelle 6.1: Speicherkosten der Merkmale des CiAO IP-Stacks für die IA-32

das dynamische Senden und Empfangen von ARP-Paketen 368 bzw. 747 Byte benötigt. Die Kosten für den AUTOSAR OS Alarm Support erscheinen mit 1888 Byte zunächst relativ hoch. Diese Kosten umfassen jedoch ebenfalls die vollständige Implementierung der Alarme in CiAO. Das einzige Merkmal, dessen Aktivierung Speicherplatz einspart, ist als *Generic Mempools* verzeichnet. Der Einsatz eines Mempools mit virtuellen Methoden (siehe 4.3) verbraucht 13 Byte weniger als ein statisch festgelegter Mempool. Dieses Resultat wird durch Compileroptimierungen erzeugt und ist in den Optimierungsstufen `-O2` und `-O3` des g++ nahezu identisch. Auf der rechten Seite der Tabelle sind die Kosten für die Protokolle UDP und TCP aufgelistet. Die dort gezeigten Merkmale entsprechen denen der in Abschnitt 3.3.3 analysierten Merkmalmodelle. Lediglich das Merkmal *Reset Invalid TCP Packets* ist hinzugekommen, das empfangene TCP-Pakete, die zu keiner TCP-Verbindung passen, mit einem RST-Flag versieht und zurückschickt. Anhand der Tabelle 6.1 ist zu erkennen, dass die hochgradige Konfigurierbarkeit der entwickelten Software-Produktlinie zu einem enormen Einsparungspotenzial führt. Der Speicherplatz, der durch das Deaktivieren eines Merkmals frei bleibt, kann direkt abgelesen werden.
Im Nachfolgenden wird der Speicherbedarf des CiAO IP-Stacks mit lwIP und uIP verglichen, indem die TCP/IP-Stacks unter denselben Bedingungen kompiliert und vermessen werden. Abbildung 6.5 stellt die minimalen Codegrößen der `text`-Sektion von lwIP grafisch dar. Die *Basisfunktionen* von lwIP umfassen das Speichermanagement sowie die Unterstützung für Ethernet und ARP und benötigen 5260 Byte der `text`-Sektion. Hinzu kommen, neben einem Pool für das Speichermanagement, 25673 Byte in der `bss`-Sektion, die lwIP intern anlegt. Wird in lwIP das Protokoll TCP verwendet, erhöht sich die

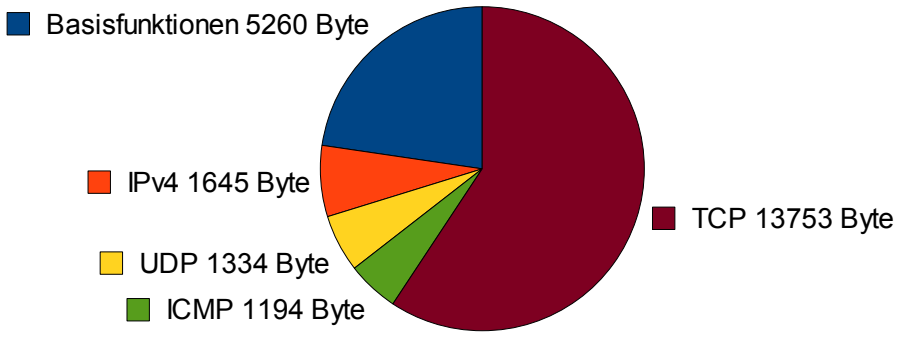

Abbildung 6.5: Speicherkosten von lwIP (Version 1.32) für die IA-32

`bss`-Sektion um weitere 9749 Byte, um u. a. die TCP-Verbindungen zu verwalten. Die Konfiguration von lwIP ist in dieser Messung auf ein absolutes Minimum beschränkt. Beispielsweise ist die API nur durch ereignisgesteuerte Callback-Funktionen realisiert, um möglichst wenig Speicherplatz zu beanspruchen. Damit wird von lwIP trotz des geringeren Funktionsumfangs, wie der fehlenden Socket-API, deutlich mehr Speicherplatz verwendet als beim CiAO IP-Stack. Besonders hervorzuheben ist die enorme Größe der `bss`-Sektion.
uIP hingegen bietet kaum Konfigurierungsmöglichkeiten, sondern enthält immer die Funktionalität für Ethernet, ARP, IPv4, TCP und eine ereignisgesteuerte API. Die Codegröße von uIP beträgt lediglich 7036 Byte. Werden genau die genannten Protokolle benötigt, ist uIP kleiner als der CiAO IP-Stack und lwIP. Die geringe Größe wird allerdings durch einige Vereinfachungen erreicht (siehe 3.2), wie z. B. die fehlende Reaktion auf ein verlorenes TCP-Paket, die dann von der Anwendungsschicht übernommen werden muss. Da bei uIP das Protokoll TCP nicht deaktivierbar ist, werden hohe Speicherkosten für ein System verursacht, das lediglich UDP benötigt. In diesem Fall ist der CiAO IP-Stack deutlich speichereffizienter als uIP.
Zusammenfassend kann festgestellt werden, dass der Speicherbedarf des CiAO IP-Stacks sehr gut mit der Anzahl von ausgewählten Merkmalen skaliert (siehe Tabelle 6.1) und mit lwIP und uIP vergleichbar ist. Der Speicherbedarf kann durch eine Merkmalselektion effektiv minimiert werden, wodurch ein Einsatz in vielen eingebetteten Systemen möglich ist.

7 Fazit und Ausblick

Innerhalb dieses Kapitels werden die zentralen Ergebnisse der vorliegenden Diplomarbeit zusammenfassend dargestellt. Anschließend erfolgt ein Ausblick über weiterführende Forschungsmöglichkeiten und Anwendungsgebiete.

Die Zielsetzung dieser Diplomarbeit war es, einen hochgradig konfigurierbaren TCP/IP-Stack für eingebettete Systeme unter Anwendung von AOP zu entwickeln. Dabei sollte die Komplexität der Software durch AOP reduziert und die daraus resultierenden Auswirkungen auf die Laufzeiteffizienz und den Speicherbedarf untersucht werden. Die Vorgehensweise anhand des Referenzprozesses der Software-Produktlinienentwicklung führte bei der Umsetzung der o. g. Ziele zu einer klaren Strukturierung. Der aspektgewahre Systementwurf der Produktlinie ermöglichte die Realisierung einer konfigurierbaren Softwarearchitektur, welche durch die Anwendung von aspektorientierten Entwurfsprinzipien vereinfacht werden konnte. Diese Entwurfsprinzipien wurden um die im Rahmen dieser Diplomarbeit entstandene Upcall-Dispatcher-Hierarchie ergänzt. Mit den Upcall-Dispatcher-Aspekten konnte eine differenziertere Taxonomie der Aspekttypen als bisher stattfinden. Die Analyse der softwaretechnischen Eigenschaften des entwickelten TCP/IP-Stacks zeigte, dass durch den Einsatz von AOP die Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit des Quelltextes gesteigert und eine Reduktion der Komplexität erzielt werden können. Darüber hinaus wurde durch den Vergleich zu anderen Systemen festgestellt, dass der Einsatz von AOP in der Praxis zu einer optimalen Performanz der Software führt. Der Speicherbedarf der entwickelten Software-Produktlinie konnte gering gehalten werden. Ebenso wurde eine sehr gute Skalierung mit der Merkmalselektion erreicht. Als abschließendes Fazit lässt sich festhalten, dass mit AOP eine effiziente Software-Produktlinie entworfen und implementiert werden kann, die sich für eingebettete Systeme eignet. Somit stellt AOP eine wertvolle Bereicherung der Programmierparadigmen dar.

Die in dieser Diplomarbeit gewonnenen Erkenntnisse bieten weitere Forschungsansätze. Eine Auseinandersetzung mit der Erweiterung des TCP/IP-Stacks um IPv6 könnte Aufschluss darüber geben, ob sich die zukünftige Version des Internetprotokolls ebenfalls für eingebettete Systeme eignet. Ein weiterer Forschungsansatz könnte die Implementierung von Routingprotokollen und die damit verbundene Evaluation des TCP/IP-Stacks als Internet-Router sein. Des Weiteren wäre die Portierung eines umfangreichen HTTP- und FTP-Servers auf diesen TCP/IP-Stack in Kombination mit CiAO interessant, um die Performanz in diesen Anwendungsfällen zu ermitteln. Eine vertiefende Bestimmung der Performanz und des Speicherbedarfs dieser Software-Produktlinie auf zusätzlichen Architekturen, wie z. B. ARM, AVR und MIPS, könnte ebenfalls eine weiterführende Forschungsmöglichkeit darstellen.

Literaturverzeichnis

[1] Lohmann, Daniel ; Scheler, Fabian ; Tartler, Reinhard ; Spinczyk, Olaf ; Schröder-Preikschat, Wolfgang: A Quantitative Analysis of Aspects in the eCos Kernel. In: *Proceedings of the EuroSys 2006 Conference (EuroSys '06)*. New York, NY, USA : ACM Press, April 2006, S. 191–204

[2] Kurose, James F. ; Ross, Keith W.: *Computer Networking. A Top-Down Approach Featuring the Internet*. 3. Auflage. Amsterdam : Pearson, Addison-Wesley, 2005. – ISBN 0–321–26976–4

[3] International Telecommunication Union (ITU): *The World in 2010. ICT Facts and Figures*. http://www.itu.int/ITU-D/ict/material/FactsFigures2010.pdf. Version: Oktober 2010

[4] Weiser, Mark ; Brown, John S.: The Coming Age of Calm Technology. In: Denning, Peter J. (Hrsg.) ; Metcalfe, Robert M. (Hrsg.): *Beyond Calculation. The Next Fifty Years of Computing*. New York, NY, USA : Copernicus, Springer, 1997. – ISBN 0–387–98588–3, S. 75–85

[5] Marwedel, Peter: *Eingebettete Systeme. Übersetzt aus dem Englischen von Lars Wehmeyer*. Korrigierter Nachdruck. Berlin, Heidelberg : Springer, 2008. – ISBN 978–3–540–34048–5

[6] Elrad, Tzilla ; Filman, Robert E. ; Bader, Atef: Aspect-oriented programming: Introduction. In: *Commun. ACM* 44 (2001), Nr. 10, S. 29–32

[7] Müller, Hans-Erich: *Unternehmensführung. Strategien – Konzepte – Praxisbeispiele*. München : Oldenbourg Wissenschaftsverlag GmbH, 2010. – ISBN 978–3–486–59729–5

[8] Böckle, Günter (Hrsg.) ; Knauber, Peter (Hrsg.) ; Pohl, Klaus (Hrsg.) ; Schmid, Klaus (Hrsg.): *Software-Produktlinien. Methoden, Einführung und Praxis*. Heidelberg : dpunkt.verlag, 2004. – ISBN 978–3–89864–257–6

[9] Clements, Paul C. ; Northrop, Linda: *Software Product Lines. Practices and Patterns*. 6. Auflage. Addison-Wesley, 2007 (SEI Series in Software Engineering). – ISBN 978–0–201–70332–0

[10] Czarnecki, Krzysztof ; Eisenecker, Ulrich W.: *Generative Programming. Methods, Tools, and Applications*. 6. Auflage. Indianapolis : Addison-Wesley, 2000. – ISBN 0–201–30977–7

[11] Buschmann, Frank ; Henney, Kevlin ; Schmidt, Douglas C.: *Pattern-Oriented Software Architecture. On Patterns and Pattern Languages. Volume 5.* Chichester, UK : Wiley, 2007 (Wiley Series in Software Desing Patterns). – ISBN 978–0471486480

[12] Ernst, Michael D. ; Badros, Greg J. ; Notkin, David: An empirical analysis of c preprocessor use. In: *IEEE Transactions on Software Engineering* 28 (2002), Nr. 12, S. 1146–1170

[13] Bacon, David F. ; Graham, Susan L. ; Sharp, Oliver J.: Compiler transformations for high-performance computing. In: *ACM Computing Surveys* 26 (1994), Nr. 4, S. 345–420

[14] Dehnert, James C. ; Stepanov, Alexander A.: Fundamentals of Generic Programming. In: Jazayeri, Mehdi (Hrsg.) ; Loss, Rüdiger G. K. (Hrsg.) ; Musser, David R. (Hrsg.): *Generic Programming. International Seminar on Generic Programming. Selected Papers.* Dagstuhl Castle : Springer, 1998. – ISBN 3–540–41090–2, S. 1–11

[15] Dijkstra, Edsger W.: *A Discipline of Programming.* New Jersey, USA : Prentice-Hall, 1976. – ISBN 0–13–215871–X

[16] Spinczyk, Olaf ; Lohmann, Daniel: The Design and Implementation of AspectC++. In: *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software* 20 (2007), Nr. 7, S. 636–651

[17] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; John, Vlissides: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software.* München : Addison-Wesley, 2009. – ISBN 978–3–8273–2824–3

[18] Engel, Michael: *Advancing Operating Systems via Aspect-Oriented Programming*, Universität Siegen, Fachbereich Elektrotechnik und Informatik, Dissertation, Juli 2005

[19] Tartler, Reinhard ; Lohmann, Daniel ; Schröder-Preikschat, Wolfgang ; Spinczyk, Olaf: Dynamic AspectC++: Generic Advice at Any Time. In: Fujita, Hamido (Hrsg.) ; Marík, Vladimír (Hrsg.): *The 8th International Conference on Software Methodologies, Tools and Techniques.* Prag, Czech Republic : IOS Press, September 2009 (Frontiers in Artificial Intelligence and Applications), S. 165–186

[20] AUTOSAR GbR: *Specification of Operating System (Version 3.1.1).* http://www.autosar.org/download/R3.1/AUTOSAR_SWS_OS.pdf. Version: Februar 2009

[21] Lohmann, Daniel ; Hofer, Wanja ; Schröder-Preikschat, Wolfgang ; Streicher, Jochen ; Spinczyk, Olaf: CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems. In: *Proceedings of the 2009 USENIX Annual Technical Conference.* Berkeley, CA, USA : USENIX Association, Juni 2009, S. 215–228

[22] Lohmann, Daniel: *Aspect-Awareness in the Development of Configurable System Software*, Friedrich-Alexander Universität Erlangen-Nürnberg, Technischen Fakultät, Dissertation, Oktober 2008

[23] Lohmann, Daniel ; Streicher, Jochen ; Spinczyk, Olaf ; Schröder-Preikschat, Wolfgang: Interrupt synchronization in the CiAO operating system. In: *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*. New York, NY, USA : ACM Press, 2007. – ISBN 1–59593–657–8

[24] Parnas, David L.: Designing Software for Ease of Extension and Contraction. In: *IEEE Transaction on Software Engineering* SE-5 (1979), Nr. 2, S. 128–137

[25] Habermann, A. N. ; Flon, Lawrence ; Cooprider, Lee: Modularization and hierarchy in a family of operating systems. In: *Communications of the ACM* 19 (1976), Nr. 5, S. 266–272

[26] Norm ISO/IEC 7498-1:1994(E) . *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*

[27] Braden, R.: *Requirements for Internet Hosts - Communication Layers*. RFC 1122 (Standard), Oktober 1989 (Request for Comments)

[28] Postel, J.: *Transmission Control Protocol*. RFC 793 (Standard), September 1981 (Request for Comments)

[29] Postel, J.: *User Datagram Protocol*. RFC 768 (Standard), August 1980 (Request for Comments)

[30] Postel, J.: *Internet Protocol*. RFC 791 (Standard), September 1981 (Request for Comments)

[31] Deering, S. ; Hinden, R.: *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard), Dezember 1998 (Request for Comments)

[32] Spurgeon, Charles E.: *Ethernet. The Definitive Guide*. Sebastopol, CA, USA : O'Reilly Media, Inc., 2000. – ISBN 1–56592–660–9

[33] Stevens, W. R.: *TCP/IP Illustrated, Volume 1. The Protocols*. 2. Auflage. Boston, MA, USA : Addison-Wesley, 1994. – ISBN 0–201–63346–9

[34] Wright, Gary R. ; Stevens, W. R.: *TCP/IP Illustrated, Volume 2. The Implementation*. Addison-Wesley, 1995. – ISBN 0–201–63354–X

[35] Hagino, Jun-ichiro i.: Implementing IPv6: experiences at KAME project. In: *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on* (2003), Januar, S. 218–221

[36] Dunkels, Adam: *Design and Implementation of the lwIP TCP/IP Stack*. Swedish Institute of Computer Science, Februar 2001. http://www.sics.se/~adam/lwip/doc/lwip.pdf

[37] Stevens, W. R. ; Fenner, Bill ; Rudoff, Andrew M.: *UNIX Network Programming. The Sockets Networking API. Volume 1*. 3. Auflage. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2004. – ISBN 0–13–141155–1

[38] Bentham, Jeremy: *TCP/IP-Lean. Web Servers for Embedded Systems*. 2. Auflage. Lawrence, KS, USA : CMP Books, 2002. – ISBN 1–57820–108–X

[39] Dunkels, Adam: Full TCP/IP for 8-bit architectures. In: *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*. New York, NY, USA : ACM, 2003, S. 85–98

[40] Tanenbaum, Andrew: *Computer Networks*. 4. Auflage. New Jersey, USA : Prentice Hall Professional Technical Reference, 2003. – ISBN 0–13–066102–3

[41] Plummer, David C.: *Ethernet Address Resolution Protocol – or – Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826 (Standard), November 1982 (Request for Comments)

[42] Harel, David: Statecharts: A visual formalism for complex systems. In: *Science of Computer Programming* 8 (1987), Nr. 3, S. 231–274

[43] Larzon, L-A. ; Degermark, M. ; Pink, S. ; Jonsson, L-E. ; Fairhurst, G.: *The Lightweight User Datagram Protocol (UDP-Lite)*. RFC 3828 (Proposed Standard), Juli 2004 (Request for Comments)

[44] Comer, Douglas E. ; Stevens, David L.: *Internetworking with TCP/IP. Volume II. Design, Implementation, and Internals*. New Jersey, USA : Prentice Hall Inc., 1991. – ISBN 0–13–472242–6

[45] Mathis, M. ; Mahdavi, J. ; Floyd, S. ; Romanow, A.: *TCP Selective Acknowledgment Options*. RFC 2018 (Proposed Standard), Oktober 1996 (Request for Comments)

[46] Clark, David D.: *Window and Acknowledgement Strategy in TCP*. RFC 813, Juli 1982 (Request for Comments)

[47] Jacobson, Van: Congestion Avoidance and Control. In: *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*. New York, NY, USA : ACM, 1988. – ISBN 0–89791–279–9, S. 314–329

[48] Jacobson, V. ; Braden, R. ; Borman, D.: *TCP Extensions for High Performance*. RFC 1323 (Proposed Standard), Mai 1992 (Request for Comments)

[49] Cox, Alan: Network Buffers and Memory Management. In: *Linux Journal* (1996), Nr. 29. http://www.linuxjournal.com/article/1312

[50] Kopetz, Hermann: *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Norwell, MA, USA : Kluwer Academic Publishers, 1997. – ISBN 0–7923–9894–7

[51] Jacobson, Van ; Felderman, Bob: *Speeding up Networking*. http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf. Version: 2006. – Linux.conf.au

[52] Milekic, Bosko: Network Buffer Allocation in the FreeBSD Operating System. Ottawa, ON, Canada, Mai 2004

[53] Smaragdakis, Yannis ; Batory, Don: Implementing Layered Designs with Mixin Layers. In: Jul, Eric (Hrsg.): *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. London, UK : Springer, 1998, S. 550–570

[54] The Apache Software Foundation: *Apache HTTP Server 2.2. Official Documentation. Apache Modules (A-H). Volume III*. Fultus Corporation, 2010. – ISBN 1–59682–193–0

[55] Paxson, V. ; Allman, M.: *Computing TCP's Retransmission Timer*. RFC 2988 (Proposed Standard), November 2000 (Request for Comments)

[56] pure-systems GmbH: *Variantenmanagement mit pure::variants*. http://www.pure-systems.com. Version: 2006. – Technical White Paper

[57] Hartmeier, Daniel: Design and Performance of the OpenBSD Stateful Packet Filter (pf). In: *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2002. – ISBN 1–880446–01–4, S. 171–180

[58] Spenneberg, Ralf: *Intrusion Detection und Prevention mit Snort 2 & Co.* München : Addison-Wesley, 2005. – ISBN 3–8273–2134–4

[59] Dunkels, Adam: *The uIP Embedded TCP/IP Stack. The uIP 1.0 Reference Manual*. Swedish Institute of Computer Science, 2006. http://www.sics.se/~adam/download/?f=uip-1.0-refman.pdf

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

[61] Habib, Irfan: Virtualization with KVM. In: *Linux Journal* (2008), Nr. 166, S. 8

[62] Warnke, Robert ; Ritzau, Thomas: *qemu-kvm & libvirt*. Norderstedt : Books on Demand GmbH, 2010. – ISBN 978–38370087600

Abbildungsverzeichnis

Quelltextverzeichnis